EXPERT INSIGHT

C# 9 и .NET 5

Разработка и оптимизация

Пятое издание

Марк Прайс

Packt

C# 9 and .NET 5 – Modern Cross-Platform Development

Fifth Edition

Build intelligent apps, websites, and services with Blazor, ASP.NET Core, and Entity Framework Core using Visual Studio Code

Mark J. Price

BIRMINGHAM - MUMBAI

Марк Прайс

C# 9 и .NET 5

Разработка и оптимизация

Пятое издание

Санкт-Петербург • Москва • Минск

2022

Марк Прайс

C# 9 и .NET 5. Разработка и оптимизация

Серия «Для профессионалов»

Перевел с английского *С. Черников*

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Художественный редактор	*В. Мостипан*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

ББК 32.973.2-018.1
УДК 004.43

Прайс Марк

П68 C# 9 и .NET 5. Разработка и оптимизация. — СПб.: Питер, 2022. — 832 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-2921-8

В этой книге опытный преподаватель Марк Прайс дает все необходимое для разработки приложений на C#. В пятом издании для работы со всеми основными операционными системами используется популярный редактор кода Visual Studio Code. Издание полностью обновлено и дополнено новой главой, касающейся Microsoft Blazor.

В первой части книги рассмотрены основы C#, включая объектно-ориентированное программирование и новые возможности C# 9, такие как создание экземпляров новых объектов с целевым типом и работа с неизменяемыми типами с использованием ключевого слова record. Во второй части рассматриваются API .NET для выполнения таких задач, как управление данными и запросы к ним, мониторинг и повышение производительности, а также работа с файловой системой, асинхронными потоками, сериализацией и шифрованием. В третьей части на примерах кросс-платформенных приложений вы сможете собрать и развернуть собственные: например, веб-приложения с использованием ASP.NET Core или мобильные приложения на Xamarin Forms.

Вы приобретете знания и навыки, необходимые для использования C# 9 и .NET 5 для разработки сервисов, веб- и мобильных приложений.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1800568105 англ. © Packt Publishing 2020
First published in the English language under the title ‘C# 9 and .NET 5 – Modern Cross-Platform Development - Fifth Edition – (9781800568105)’
ISBN 978-5-4461-2921-8 © Перевод на русский язык ООО Издательство «Питер», 2022
© Издание на русском языке, оформление ООО Издательство «Питер», 2022
© Серия «Для профессионалов», 2022

Права на издание получены по соглашению с Packt Publishing. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 10.2021. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 30.09.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 67,080. Тираж 500. Заказ 0000.

Краткое содержание

Об авторе ... 27

О научном редакторе ... 29

Предисловие ... 30

Глава 1. Привет, C#! Здравствуй, .NET Core! ... 36

Глава 2. Говорим на языке C# ... 66

Глава 3. Управление потоком исполнения и преобразование типов ... 115

Глава 4. Разработка, отладка и тестирование функций ... 152

Глава 5. Создание пользовательских типов с помощью объектно-ориентированного программирования ... 186

Глава 6. Реализация интерфейсов и наследование классов ... 229

Глава 7. Описание и упаковка типов .NET ... 272

Глава 8. Работа с распространенными типами .NET ... 307

Глава 9. Работа с файлами, потоками и сериализация ... 348

Глава 10. Защита данных и приложений ... 384

Глава 11. Работа с базами данных с помощью Entity Framework Core ... 412

Глава 12. Создание запросов и управление данными с помощью LINQ ... 457

Глава 13. Улучшение производительности и масштабируемости с помощью многозадачности ... 491

Глава 14. Практическое применение C# и .NET ... 520

Глава 15. Разработка сайтов с помощью ASP.NET Core Razor Pages ... 542

Глава 16. Разработка сайтов с использованием паттерна MVC ... 586

Глава 17. Разработка сайтов с помощью системы управления контентом (CMS)..... 630

Глава 18. Разработка и использование веб-сервисов 681

Глава 19. Разработка интеллектуальных приложений с помощью алгоритмов машинного обучения 729

Глава 20. Создание пользовательских веб-интерфейсов с помощью Blazor 759

Глава 21. Разработка кросс-платформенных мобильных приложений 797

Послесловие 832

Оглавление

Об авторе ... 27

О научном редакторе ... 29

Предисловие ... 30

Структура книги ... 30

Необходимое программное обеспечение ... 34

Условные обозначения ... 34

От издательства ... 35

Глава 1. Привет, C#! Здравствуй, .NET Core! ... 36

Настройка среды разработки ... 37

Использование Visual Studio Code для разработки кросс-платформенных приложений ... 37

Использование GitHub Codespaces для разработки в облаке ... 38

Использование Visual Studio 2019 для разработки Windows-приложений ... 38

Использование Visual Studio на Mac для разработки мобильных приложений ... 39

Рекомендуемые инструменты и операционные системы ... 39

Кросс-платформенное развертывание ... 40

Знакомство с версиями Microsoft Visual Studio Code ... 40

Скачивание и установка среды Visual Studio Code ... 42

Установка расширений ... 43

Знакомство с .NET ... 43

Обзор .NET Framework ... 43

Проекты Mono и Xamarin ... 44

Обзор .NET Core ... 45

Обзор .NET 5 и последующих версий .NET ... 45

Поддержка .NET Core ... 46
Версии .NET Runtime и .NET SDK ... 47
Удаление старых версий .NET ... 48
В чем особенность .NET Core ... 49
Обзор .NET Standard ... 50
Платформы .NET в изданиях этой книги ... 51
Знакомство с промежуточным языком ... 52
Сравнение технологий .NET ... 53
Разработка консольных приложений с использованием Visual Studio Code ... 53
Написание кода с помощью Visual Studio Code ... 53
Компиляция и запуск кода с использованием инструмента командной строки dotnet ... 56
Написание программ верхнего уровня ... 56
Скачивание кода решения из репозитория GitHub ... 57
Использование системы Git в Visual Studio Code ... 58
Клонирование репозитория с примерами из книги ... 58
Поиск справочной информации ... 59
Знакомство с Microsoft Docs ... 59
Получение справки для инструмента dotnet ... 59
Получение определений типов и их элементов ... 60
Ищем ответы на Stack Overflow ... 62
Поисковая система Google ... 62
Подписка на официальный блог .NET ... 63
Видео от Скотта Хансельмана ... 63
Практические задания ... 63
Упражнение 1.1. Проверочные вопросы ... 63
Упражнение 1.2. Практическое задание ... 64
Упражнение 1.3. Дополнительные ресурсы ... 64
Резюме ... 65

Глава 2. Говорим на языке C# ... 66
Введение в C# ... 66
Обзор версий языка и их функций ... 67
Версии компилятора C# ... 71
Включение версии компилятора на определенном языке ... 72
Основы языка C# ... 74
Грамматика языка C# ... 75
Терминология языка C# ... 76
Изменение цветовой схемы синтаксиса ... 77
Сравнение языков программирования с естественными языками ... 77

Работа с переменными ... 82
Присвоение переменным имен ... 82
Хранение текста ... 83
Хранение чисел ... 84
Хранение логических значений ... 90
Использование рабочих областей Visual Studio Code ... 90
Хранение объектов любого типа ... 91
Хранение данных динамического типа ... 93
Локальные переменные ... 93
Использование целевого типа выражения new для создания экземпляров объектов ... 95
Получение значений по умолчанию для типов ... 95
Хранение нескольких значений ... 96
Работа со значениями null ... 97
Создание значимого типа, допускающего значение null ... 97
Включение ссылочных типов, допускающих и не допускающих значение null ... 99
Объявление переменных и параметров, не допускающих значение null ... 100
Проверка на null ... 102
Дальнейшее изучение консольных приложений ... 103
Отображение вывода пользователю ... 103
Форматирующие строки ... 104
Получение пользовательского ввода ... 106
Импорт пространства имен ... 106
Упрощение работы с командной строкой ... 107
Получение клавиатурного ввода от пользователя ... 107
Чтение аргументов ... 108
Настройка параметров с помощью аргументов ... 110
Работа с платформами, не поддерживающими некоторые API ... 111
Практические задания ... 112
Упражнение 2.1. Проверочные вопросы ... 112
Упражнение 2.2. Практическое задание — числовые размеры и диапазоны ... 112
Упражнение 2.3. Дополнительные ресурсы ... 113
Резюме ... 114

Глава 3. Управление потоком исполнения и преобразование типов ... 115
Работа с переменными ... 115
Унарные операции ... 116
Арифметические бинарные операции ... 117
Операция присваивания ... 119

Логические операции ... 119
Условные логические операции ... 120
Побитовые операции и операции побитового сдвига ... 122
Прочие операции ... 123
Операторы выбора ... 123
Ветвление с помощью оператора if ... 124
Почему в операторах if необходимы фигурные скобки ... 125
Сопоставление шаблонов с помощью операторов if ... 125
Ветвление с помощью оператора switch ... 126
Сопоставление шаблонов с помощью оператора switch ... 127
Упрощение операторов switch с помощью выражений switch ... 129
Операторы цикла ... 130
Оператор while ... 130
Оператор do ... 131
Оператор for ... 132
Оператор foreach ... 132
Приведение и преобразование типов ... 133
Явное и неявное приведение типов ... 134
Использование типа System.Convert ... 136
Округление чисел ... 137
Преобразование значения любого типа в строку ... 138
Преобразование двоичного (бинарного) объекта в строку ... 139
Разбор строк для преобразования в числа или значения даты и времени ... 140
Обработка исключений при преобразовании типов ... 142
Проверка переполнения ... 145
Практические задания ... 148
Упражнение 3.1. Проверочные вопросы ... 148
Упражнение 3.2. Циклы и переполнение ... 149
Упражнение 3.3. Циклы и операторы ... 149
Упражнение 3.4. Обработка исключений ... 150
Упражнение 3.5. Проверка знания операций ... 150
Упражнение 3.6. Дополнительные ресурсы ... 151
Резюме ... 151

Глава 4. Разработка, отладка и тестирование функций ... 152
Написание функций в языке C# ... 152
Написание функции для таблицы умножения ... 153
Функции, возвращающие значение ... 155
Написание математических функций ... 157
Документирование функций с помощью XML-комментариев ... 161

Использование лямбда-выражений в реализациях функций 163
Отладка в процессе разработки 166
Преднамеренное добавление ошибок в код 166
Установка точек останова 167
Навигация с помощью панели средств отладки 169
Панели отладки 169
Пошаговое выполнение кода 170
Настройка точек останова 171
Регистрация событий во время разработки и выполнения проекта 173
Работа с типами Debug и Trace 173
Прослушиватель трассировки 174
Настройка прослушивателей трассировки 175
Переключение уровней трассировки 176
Модульное тестирование функций 179
Создание библиотеки классов, требующей тестирования 180
Разработка модульных тестов 181
Выполнение модульных тестов 182
Практические задания 183
Упражнение 4.1. Проверочные вопросы 183
Упражнение 4.2. Функции, отладка и модульное тестирование 184
Упражнение 4.3. Дополнительные ресурсы 184
Резюме 185

Глава 5. Создание пользовательских типов с помощью объектно-ориентированного программирования 186
Коротко об объектно-ориентированном программировании 186
Разработка библиотек классов 187
Создание библиотек классов 188
Определение классов 189
Создание экземпляров классов 190
Управление несколькими файлами 192
Работа с объектами 192
Хранение данных в полях 193
Определение полей 193
Модификаторы доступа 194
Установка и вывод значений полей 195
Хранение значения с помощью типа-перечисления 196
Хранение группы значений с помощью типа enum 197
Хранение нескольких значений с помощью коллекций 199
Создание статического поля 200

Создание константного поля ...201
Создание поля только для чтения...202
Инициализация полей с помощью конструкторов ...203
Установка значения поля с использованием литерала для значения по умолчанию...204
Запись и вызов методов...206
Возвращение значений из методов...206
Возвращение нескольких значений с помощью кортежей...207
Определение и передача параметров в методы ...210
Перегрузка методов...211
Передача необязательных параметров и именованных аргументов ...211
Управление передачей параметров ...213
Ключевое слово ref...215
Разделение классов с помощью ключевого слова partial ...215
Управление доступом с помощью свойств и индексаторов...216
Определение свойств только для чтения...216
Определение изменяемых свойств ...217
Определение индексаторов ...219
Сопоставление шаблонов с объектами ...220
Создание и работа с библиотеками классов .NET 5...220
Определение пассажиров при полете ...221
Изменения сопоставления с шаблоном в C# 9...222
Работа с записями...223
Свойства только для инициализации ...223
Записи ...224
Упрощение членов данных ...225
Позиционирование записей ...226
Практические задания ...227
Упражнение 5.1. Проверочные вопросы ...227
Упражнение 5.2. Дополнительные ресурсы ...227
Резюме ...228

Глава 6. Реализация интерфейсов и наследование классов ...229
Настройка библиотеки классов и консольного приложения...229
Упрощение методов ...231
Реализация функционала с помощью методов ...232
Реализация функционала с помощью операций...234
Реализация функционала с помощью локальных функций...235
Вызов и обработка событий...236
Вызов методов с помощью делегатов...236

Определение и обработка делегатов ... 237
Определение и обработка событий ... 239
Реализация интерфейсов ... 240
Универсальные интерфейсы ... 240
Сравнение объектов при сортировке ... 241
Сравнение объектов с помощью отдельных классов ... 243
Определение интерфейсов с реализациями по умолчанию ... 245
Обеспечение безопасности многократного использования типов с помощью дженериков ... 247
Работа с типами-дженериками ... 248
Работа с методами-дженериками ... 250
Управление памятью с помощью ссылочных типов и типов значений ... 251
Работа со структурами ... 252
Освобождение неуправляемых ресурсов ... 253
Обеспечение вызова метода Dispose ... 256
Наследование классов ... 256
Расширение классов ... 257
Сокрытие членов класса ... 257
Переопределение членов ... 259
Предотвращение наследования и переопределения ... 260
Полиморфизм ... 260
Приведение в иерархиях наследования ... 262
Неявное приведение ... 262
Явное приведение ... 262
Обработка исключений приведения ... 263
Наследование и расширение типов .NET ... 264
Наследование исключений ... 265
Расширение типов при невозможности наследования ... 266
Практические задания ... 269
Упражнение 6.1. Проверочные вопросы ... 269
Упражнение 6.2. Создание иерархии наследования ... 269
Упражнение 6.3. Дополнительные ресурсы ... 270
Резюме ... 271

Глава 7. Описание и упаковка типов .NET ... 272
Введение в .NET 5 ... 272
.NET Core 1.0 ... 273
.NET Core 1.1 ... 274
.NET Core 2.0 ... 274
.NET Core 2.1 ... 274

.NET Core 2.2 275
.NET Core 3.0 275
.NET 5.0 276
Повышение производительности с .NET Core 2.0 до .NET 5 276
Использование компонентов .NET 277
Сборки, пакеты и пространства имен 277
Импорт пространства имен для использования типа 281
Связь ключевых слов языка C# с типами .NET 281
Создание кросс-платформенных библиотек классов при помощи .NET Standard 283
Создание библиотеки классов .NET Standard 2.0 284
Публикация и развертывание ваших приложений 285
Разработка консольного приложения для публикации 286
Команды dotnet 287
Публикация автономного приложения 288
Публикация однофайлового приложения 289
Уменьшение размера приложений с помощью обрезки приложений 290
Декомпиляция сборок 291
Упаковка библиотек для распространения через NuGet 295
Ссылка на пакет NuGet 295
Упаковка библиотеки для NuGet 297
Тестирование пакета 299
Перенос приложений с .NET Framework на .NET 5 301
Что означает перенос 301
Стоит ли переносить 302
Сравнение .NET Framework и .NET 5 302
Анализатор переносимости .NET 303
Использование библиотек, не скомпилированных для .NET Standard 303
Практические задания 305
Упражнение 7.1. Проверочные вопросы 305
Упражнение 7.2. Дополнительные ресурсы 305
Резюме 306

Глава 8. Работа с распространенными типами .NET 307
Работа с числами 307
Большие целые числа 308
Комплексные числа 309
Работа с текстом 310
Извлечение длины строки 310
Извлечение символов строки 310

Разделение строк 311
Извлечение фрагмента строки 311
Поиск содержимого в строках 312
Конкатенация строк, форматирование и прочие члены типа string 313
Эффективное создание строк 314
Сопоставление шаблонов с использованием регулярных выражений 314
Проверка цифр, введенных как текст 315
Синтаксис регулярных выражений 316
Примеры регулярных выражений 317
Разбивка сложных строк, разделенных запятыми 318
Улучшение производительности регулярных выражений 319
Хранение данных с помощью коллекций 319
Общие свойства коллекций 320
Выбор коллекции 322
Работа со списками 324
Работа со словарями 325
Сортировка коллекций 326
Использование специализированных коллекций 327
Использование неизменяемых коллекций 327
Работа с интервалами, индексами и диапазонами 328
Управление памятью с помощью интервалов 328
Идентификация позиций с типом Index 329
Идентификация диапазонов с помощью типа Range 329
Использование индексов и диапазонов 330
Работа с сетевыми ресурсами 331
Работа с URI, DNS и IP-адресами 331
Проверка соединения с сервером 332
Работа с типами и атрибутами 334
Версии сборок 334
Чтение метаданных сборки 335
Создание пользовательских атрибутов 337
Еще немного об отражении 339
Работа с изображениями 340
Интернационализация кода 342
Обнаружение и изменение региональных настроек 342
Обработка часовых поясов 344
Практические задания 345
Упражнение 8.1. Проверочные вопросы 345
Упражнение 8.2. Регулярные выражения 345

Упражнение 8.3. Методы расширения 346
Упражнение 8.4. Дополнительные ресурсы 346
Резюме 347

Глава 9. Работа с файлами, потоками и сериализация 348
Управление файловой системой 348
Работа с различными платформами и файловыми системами 348
Управление дисками 350
Управление каталогами 351
Управление файлами 353
Управление путями 355
Извлечение информации о файле 356
Контроль работы с файлами 357
Чтение и запись с помощью потоков 358
Запись в текстовые потоки 360
Запись в XML-потоки 361
Освобождение файловых ресурсов 363
Сжатие потоков 365
Сжатие с помощью алгоритма Бротли 367
Высокопроизводительные потоки с использованием конвейеров 369
Асинхронные потоки 369
Кодирование и декодирование текста 369
Преобразование строк в последовательности байтов 370
Кодирование и декодирование текста в файлах 372
Сериализация графов объектов 373
XML-сериализация 373
Генерация компактного XML 376
XML-десериализация 377
JSON-сериализация 378
Высокопроизводительный процессор JSON 379
Практические задания 381
Упражнение 9.1. Проверочные вопросы 381
Упражнение 9.2. XML-сериализация 382
Упражнение 9.3. Дополнительные ресурсы 382
Резюме 383

Глава 10. Защита данных и приложений 384
Терминология безопасности 385
Ключи и их размеры 385
Векторы инициализации и размеры блоков 386

Соль....386
Генерация ключей и векторов инициализации....387
Шифрование и дешифрование данных....387
Симметричное шифрование с помощью алгоритма AES....388
Хеширование данных....393
Хеширование с помощью алгоритма SHA256....393
Подписывание данных....396
Подписывание с помощью алгоритмов SHA256 и RSA....397
Генерация случайных чисел....400
Генерация случайных чисел для игр....400
Генерация случайных чисел для криптографии....401
Криптография: что нового....402
Аутентификация и авторизация пользователей....403
Реализация аутентификации и авторизации....405
Защита приложения....408
Практические задания....409
Упражнение 10.1. Проверочные вопросы....409
Упражнение 10.2. Защита данных с помощью шифрования и хеширования....410
Упражнение 10.3. Дешифрование данных....410
Упражнение 10.4. Дополнительные ресурсы....410
Резюме....411

Глава 11. Работа с базами данных с помощью Entity Framework Core....412
Современные базы данных....412
Введение в Entity Framework....413
Entity Framework Core....414
Использование образца реляционной базы данных....414
Создание образца базы данных Northwind для SQLite....416
Управление образцом базы данных Northwind в SQLiteStudio....417
Настройка EF Core....418
Выбор поставщика данных Entity Framework Core....418
Настройка инструмента dotnet-ef....419
Подключение к базе данных....420
Определение моделей EF Core....420
Соглашения EF Core....421
Атрибуты аннотаций Entity Framework Core....421
Entity Framework Core Fluent API....423
Заполнение таблиц базы данных....423
Создание модели Entity Framework Core....423

Создание моделей с использованием существующей базы данных 428
Запрос данных из моделей EF Core 433
Фильтрация включенных сущностей 434
Фильтрация и сортировка товаров 436
Получение сгенерированного SQL-кода 437
Логирование в EF Core 438
Теги запросов 442
Сопоставление с образцом с помощью оператора Like 442
Определение глобальных фильтров 443
Схемы загрузки данных при использовании EF Core 444
Жадная загрузка элементов 444
Использование ленивой загрузки 445
Явная загрузка элементов 446
Управление данными с помощью EF Core 449
Добавление элементов 449
Обновление элементов 450
Удаление элементов 451
Пулы соединений с базами данных 452
Транзакции 453
Определение явной транзакции 454
Практические задания 455
Упражнение 11.1. Проверочные вопросы 455
Упражнение 11.2. Экспорт данных с помощью различных форматов сериализации 456
Упражнение 11.3. Изучение документации EF Core 456
Резюме 456

Глава 12. Создание запросов и управление данными с помощью LINQ 457
Написание запросов LINQ 457
Расширение последовательностей с помощью класса Enumerable 458
Фильтрация элементов с помощью метода Where 459
Сортировка элементов 463
Фильтрация по типу 465
Работа с множествами с помощью LINQ 466
Использование LINQ с EF Core 468
Создание модели EF Core 468
Фильтрация и сортировка последовательностей 471
Проецирование последовательностей в новые типы 472
Объединение и группировка 473
Агрегирование последовательностей 477

Подслащение синтаксиса LINQ с помощью синтаксического сахара 478
Использование нескольких потоков и параллельного LINQ 480
Разработка приложения с помощью нескольких потоков....................480
Создание собственных методов расширения LINQ 483
Работа с LINQ to XML....................486
Генерация XML с помощью LINQ to XML....................487
Чтение XML с помощью LINQ to XML....................487
Практические задания 488
Упражнение 12.1. Проверочные вопросы 489
Упражнение 12.2. Создание запросов LINQ....................489
Упражнение 12.3. Дополнительные ресурсы....................490
Резюме 490

Глава 13. Улучшение производительности и масштабируемости с помощью многозадачности....................491
Процессы, потоки и задачи....................491
Мониторинг производительности и использования ресурсов....................493
Оценка эффективности типов....................493
Мониторинг производительности и использования памяти....................494
Реализация класса Recorder 495
Асинхронное выполнение задач 499
Синхронное выполнение нескольких действий....................499
Асинхронное выполнение нескольких действий с помощью задач 501
Ожидание выполнения задач....................502
Задачи продолжения....................503
Вложенные и дочерние задачи 505
Синхронизация доступа к общим ресурсам....................506
Доступ к ресурсу из нескольких потоков 507
Применение к ресурсу взаимоисключающей блокировки....................508
Оператор блокировки и избежание взаимоблокировок....................509
Синхронизация событий 511
Выполнение атомарных операций 512
Использование других типов синхронизации 513
Ключевые слова async и await 513
Увеличение скорости отклика консольных приложений 514
Увеличение скорости отклика GUI-приложений....................515
Улучшение масштабируемости клиент-серверных приложений....................515
Распространенные типы, поддерживающие многозадачность 516
Ключевое слово await в блоках catch 516
Работа с асинхронными потоками 516

Практические задания 518
Упражнение 13.1. Проверочные вопросы 518
Упражнение 13.2. Дополнительные ресурсы 518
Резюме 519

Глава 14. Практическое применение C# и .NET 520
Модели приложений для C# и .NET 520
Разработка сайтов с помощью ASP.NET Core 521
Разработка сайтов с использованием системы управления веб-контентом (веб-содержимым) 521
Веб-приложения 522
Создание и использование веб-сервисов 523
Разработка интеллектуальных приложений 523
Нововведения ASP.NET Core 524
ASP.NET Core 1.0 524
ASP.NET Core 1.1 524
ASP.NET Core 2.0 524
ASP.NET Core 2.1 525
ASP.NET Core 2.2 525
ASP.NET Core 3.0 526
ASP.NET Core 3.1 526
Blazor WebAssembly 3.2 527
ASP.NET Core 5.0 527
SignalR 528
Blazor 529
JavaScript и другие технологии 530
Silverlight — использование C# и .NET через плагин 530
WebAssembly — база для Blazor 530
Blazor — на стороне сервера или клиента 531
Дополнительные материалы 531
Разработка кросс-платформенных мобильных и настольных Windows-приложений 532
Разработка настольных Windows-приложений с использованием устаревших технологий 533
Разработка модели данных объекта для Northwind 534
Разработка библиотеки классов для сущностных моделей Northwind 535
Создание моделей сущностей с использованием dotnet-ef 535
Улучшение сопоставления классов и таблиц вручную 537
Создание библиотеки классов для контекста базы данных Northwind 538
Резюме 541

Глава 15. Разработка сайтов с помощью ASP.NET Core Razor Pages 542
Веб-разработка 542
Протокол передачи гипертекста 542
Клиентская веб-разработка 546
Обзор ASP.NET Core 546
Классический ASP.NET против современного ASP.NET Core 548
Разработка проекта ASP.NET Core 548
Тестирование и защита сайта 550
Управление средой хостинга 554
Включение статических файлов и файлов по умолчанию 555
Технология Razor Pages 558
Включение Razor Pages 558
Определение Razor Pages 558
Использование общих макетов в Razor Pages 560
Использование файлов с выделенным кодом в Razor Pages 563
Использование Entity Framework Core совместно с ASP.NET Core 565
Настройка Entity Framework Core в виде сервиса 565
Управление данными с помощью страниц Razor 567
Применение библиотек классов Razor 569
Создание библиотеки классов Razor 569
Отключение компактных папок 569
Реализация функции сотрудников с помощью EF Core 571
Реализация частичного представления для отображения одного сотрудника 573
Использование библиотек классов Razor 574
Настройка служб и конвейера HTTP-запросов 575
Регистрация служб 576
Конфигурация конвейера HTTP-запросов 578
Простой проект сайта ASP.NET Core 582
Практические задания 583
Упражнение 15.1. Проверочные вопросы 583
Упражнение 15.2. Веб-приложение, управляемое данными 584
Упражнение 15.3. Создание веб-страниц для консольных приложений 584
Упражнение 15.4. Дополнительные ресурсы 584
Резюме 585

Глава 16. Разработка сайтов с использованием паттерна MVC 586
Настройка сайта ASP.NET Core MVC 586
Создание и изучение сайтов ASP.NET Core MVC 587
Обзор сайта ASP.NET Core MVC 590
Обзор базы данных ASP.NET Core Identity 592

Изучение сайта ASP.NET Core MVC 592
Запуск ASP.NET Core MVC 592
Маршрутизация по умолчанию 595
Контроллеры и действия 596
Соглашение о пути поиска представлений 597
Модульное тестирование MVC 598
Фильтры 598
Сущности и модели представлений 600
Представления 602
Добавление собственного функционала на сайт ASP.NET Core MVC 605
Определение пользовательских стилей 606
Настройка категории изображений 606
Синтаксис Razor 606
Определение типизированного представления 607
Тестирование измененной главной страницы 610
Передача параметров с помощью значения маршрута 611
Привязка моделей 613
Проверка модели 617
Методы класса-помощника для представления 620
Отправка запросов в базу данных и использование шаблонов отображения 621
Улучшение масштабируемости с помощью асинхронных задач 623
Использование других шаблонов проектов 625
Установка дополнительных шаблонов 626
Практические задания 627
Упражнение 16.1. Проверочные вопросы 627
Упражнение 16.2. Реализация MVC для страницы, содержащей сведения о категориях 628
Упражнение 16.3. Улучшение масштабируемости за счет понимания и реализации асинхронных методов действий 628
Упражнение 16.4. Дополнительные ресурсы 628
Резюме 629

Глава 17. Разработка сайтов с помощью системы управления контентом (CMS) 630
Преимущества CMS 630
Основные функции CMS 631
Возможности корпоративной CMS 632
Платформы CMS 632
Piranha CMS 633
Библиотеки с открытым исходным кодом и лицензирование 633
Создание веб-приложения с помощью Piranha CMS 634

Изучение сайта Piranha CMS 635
Редактирование содержимого сайта и страницы....636
Создание новой страницы верхнего уровня 641
Создание новой дочерней страницы 642
Обзор архива блога....643
Комментирование постов и страниц....644
Аутентификация и авторизация 646
Изучение конфигурации 648
Тестирование нового контента 649
Маршрутизация....650
Мультимедиа 652
Сервис приложения 653
Типы контента....654
Стандартные блоки 660
Типы компонентов и стандартных блоков....660
Определение компонентов, типов контента и шаблонов 662
Определение пользовательских шаблонов контента для типов контента....667
Настройка запуска и импорта из базы данных....670
Создание контента с использованием шаблона проекта....673
Тестирование сайта Northwind CMS 674
Загрузка изображений и создание корня каталога 674
Импорт контента (содержимого) страниц категорий и товаров....675
Управление контентом (содержимым) каталога 676
Хранение контента в системе Piranha....678
Практические задания 679
Упражнение 17.1. Проверочные вопросы 679
Упражнение 17.2. Определение типа блока для отображения видео с YouTube....680
Упражнение 17.3. Дополнительные ресурсы....680
Резюме....680

Глава 18. Разработка и использование веб-сервисов....681
Разработка веб-сервисов с помощью Web API в ASP.NET Core 681
Аббревиатуры, типичные для веб-сервисов 681
Разработка проекта Web API в ASP.NET Core 683
Функциональность веб-сервисов....686
Создание веб-сервиса для базы данных Northwind 687
Создание хранилищ данных для сущностей 689
Реализация контроллера Web API....692
Настройка хранилища данных клиентов и контроллера Web API....694

Спецификация деталей проблемы ... 698
Управление сериализацией XML ... 699
Документирование и тестирование веб-сервисов ... 700
Тестирование GET-запросов в браузерах ... 700
Тестирование HTTP-запросов с помощью расширения REST Client ... 702
Swagger ... 705
Тестирование запросов с помощью Swagger UI ... 707
Обращение к сервисам с помощью HTTP-клиентов ... 711
HttpClient ... 712
Настройка HTTP-клиентов с помощью HttpClientFactory ... 712
Получение списка клиентов в контроллере в формате JSON ... 713
Включение совместного использования ресурсов между источниками ... 716
Реализация расширенных функций ... 718
Мониторинг работоспособности — HealthCheck API ... 718
Реализация анализаторов и соглашений Open API ... 719
Обработка проходных отказов ... 720
Система маршрутизации на основе конечных точек ... 720
Настройки маршрутизации на основе конечных точек ... 721
Добавление HTTP-заголовков для безопасности ... 723
Защита веб-сервисов ... 725
Прочие коммуникационные технологии ... 725
Windows Communication Foundation (WCF) ... 725
gRPC ... 726
Практические задания ... 726
Упражнение 18.1. Проверочные вопросы ... 726
Упражнение 18.2. Создание и удаление клиентов с помощью HttpClient ... 727
Упражнение 18.3. Дополнительные ресурсы ... 727
Резюме ... 728

Глава 19. Разработка интеллектуальных приложений с помощью алгоритмов машинного обучения ... 729
Общие сведения о машинном обучении ... 729
Жизненный цикл систем машинного обучения ... 730
Наборы данных для обучения и тестирования ... 731
Задачи машинного обучения ... 732
Машинное обучение Microsoft Azure ... 733
Знакомство с ML.NET ... 734
Знакомство с Infer.NET ... 734
Обучающие конвейеры ML.NET ... 735

Концепции обучения моделей 736
Пропущенные значения и типы ключей 737
Характеристики и метки 737
Рекомендация товаров пользователю 738
Анализ проблем 738
Сбор и обработка данных 739
Разработка проекта сайта NorthwindML 740
Тестирование сайта с рекомендациями по использованию товара 753
Практические задания 755
Упражнение 19.1. Проверочные вопросы 755
Упражнение 19.2. Примеры 756
Упражнение 19.3. Дополнительные ресурсы 757
Резюме 757

Глава 20. Создание пользовательских веб-интерфейсов с помощью Blazor 759
Знакомство с Blazor 759
Модели хостинга Blazor 759
Компоненты Blazor 760
Blazor и Razor 761
Сравнение шаблонов проектов Blazor 761
Обзор шаблона проекта Blazor Server 762
CSS-изоляция 766
Запуск шаблона проекта Blazor Server 767
Обзор шаблона проекта Blazor WebAssembly 768
Сборка компонентов с помощью Blazor Server 771
Определение и тестирование простого компонента 771
Получение сущностей и их компонентов 773
Абстрагирование службы для компонента Blazor 776
Использование форм Blazor 778
Определение форм с помощью компонента EditForm 778
Создание и использование компонента формы клиента 779
Создание компонентов с использованием Blazor WebAssembly 784
Настройки сервера для Blazor WebAssembly 785
Настройка клиента для Blazor WebAssembly 788
Поддержка прогрессивных веб-приложений 792
Практические задания 794
Упражнение 20.1. Проверочные вопросы 794
Упражнение 20.2. Упражнения по созданию компонента 795
Упражнение 20.3. Дополнительные ресурсы 795
Резюме 796

Глава 21. Разработка кросс-платформенных мобильных приложений 797
Знакомство с XAML 798
Упрощение кода с помощью XAML 798
Выбор общих элементов управления 799
Расширения разметки 800
Знакомство с Xamarin и Xamarin.Forms 800
Xamarin.Forms в качестве расширения Xamarin 801
Мобильные стратегии 801
Доля рынка мобильных платформ 802
Дополнительная функциональность 802
Компоненты пользовательского интерфейса Xamarin.Forms 804
Разработка мобильных приложений с помощью Xamarin.Forms 805
Добавление Android SDK 806
Создание решения Xamarin.Forms 806
Создание модели объекта с двусторонней привязкой данных 808
Создание компонента для набора телефонных номеров 812
Создание представлений для списка клиентов и подробной информации о клиенте 815
Тестирование мобильного приложения в среде iOS 821
Взаимодействие мобильных приложений с веб-сервисами 824
Настройка веб-сервиса в целях разрешения небезопасных запросов 824
Настройка приложения для iOS в целях разрешения небезопасных подключений 825
Настройка приложения для Android в целях разрешения небезопасных подключений 826
Добавление NuGet-пакетов для потребления веб-сервиса 827
Получение данных о клиентах с помощью сервиса 827
Практические задания 829
Упражнение 21.1. Проверочные вопросы 830
Упражнение 21.2. Дополнительные ресурсы 830
Резюме 831

Послесловие 832

Об авторе

Марк Дж. Прайс — обладатель сертификатов Microsoft Certified Trainer (MCT), Microsoft Specialist: Programming in C# и Microsoft Specialist: Architecting Microsoft Azure Infrastructure Solutions, имеющий более 20 лет практики в области обучения и программирования.

С 1993 года Марк сдал свыше 80 экзаменов корпорации Microsoft по программированию и специализируется на подготовке других людей к успешному прохождению тестирования. Его студенты — как 16-летние новички, так и профессионалы с многолетним опытом. Марк ведет эффективные тренинги, сочетая теоретические материалы с реальным опытом консультирования и разработки систем для корпораций по всему миру.

В период с 2001 по 2003 год Марк посвящал все свое время разработке официального обучающего программного обеспечения в штаб-квартире Microsoft в американском городе Редмонд. В составе команды он написал первый обучающий курс по C#, когда была только выпущена ранняя альфа-версия языка. Во время сотрудничества с Microsoft он работал инструктором по повышению квалификации сертифицированных корпорацией специалистов на специальных тренингах, посвященных C# и .NET.

Microsoft CERTIFIED
Solutions Developer
App Builder

Microsoft Specialist
Programming in C#

Episerver CMS
Certified Developer

В настоящее время Марк разрабатывает и поддерживает обучающие курсы для системы Digital Experience Platform, включая Content Cloud, Commerce Cloud и Intelligence Cloud.

В 2010 году Марк получил свидетельство об окончании последипломной программы обучения, дающее право на преподавание. Он преподает старшеклассникам математику в двух средних школах в Лондоне. Кроме того, Марк получил степень бакалавра компьютерных наук с отличием в Бристольском университете (Великобритания).

Спасибо моим родителям, Памеле и Иану, за мое воспитание, за то, что привили мне трудолюбие и любопытство к миру. Спасибо моим сестрам, Эмили и Джульетте, за то, что полюбили меня, неуклюжего старшего братца. Спасибо моим друзьям и коллегам, которые вдохновляют меня технически и творчески. Наконец, спасибо всем ученикам, на протяжении многих лет слушавшим меня, за то, что побудили меня стать лучшим преподавателем, чем я есть.

О научном редакторе

Дамир Арх много лет разрабатывает и сопровождает программное обеспечение; от сложных корпоративных проектов до современных мобильных приложений. Он работал с различными языками, однако его любимым языком остается C#. Стремясь к совершенствованию процессов разработки, Дамир является сторонником CI/CD, а также разработки, основанной на тестировании. Он делится своими знаниями на курсах для пользователей и на конференциях, ведет блоги и пишет статьи. Дамир Арх девять раз подряд получил престижную премию Microsoft MVP в категории Developer Technologiees. В свободное время он всегда в движении: любит пеший туризм, геокэшинг, бег и скалолазание.

Я хотел бы поблагодарить мою семью и друзей за их терпение и понимание во время выходных и вечеров, которые я провел, работая за компьютером и помогая сделать эту книгу лучше для всех.

Предисловие

Существуют исчерпывающие справочники по платформе .NET и программированию на C# объемом в тысячи страниц. Эта книга другая. Ее содержание лаконичное и увлекательное, она наполнена практическими упражнениями по каждой теме. Широта всеобъемлющего повествования достигается за счет несколько меньшей глубины, но при желании вы найдете здесь множество указателей на материалы для дальнейшего изучения.

Данная книга одновременно представляет собой пошаговое руководство по изучению современных проверенных практик на языке C# с использованием кросс-платформенного .NET и краткое введение в основные типы приложений, которые можно создавать с их помощью. Книга лучше всего подходит новичкам в C# и .NET или программистам, которые уже ранее работали с C#, но желают совершенствоваться.

Если вы уже имеете опыт работы с C# или более поздней версией, то в первом разделе главы 2, касающейся C#, можете ознакомиться со сравнительными таблицами новых языковых функций и перейти к ним. Если уже имеете опыт работы с .NET или более поздней версией, то в первом разделе главы 7 можете ознакомиться с таблицами функций новой платформы и сразу перейти к ним.

Я расскажу о крутых фишках и секретах языка C# и .NET, чтобы вы могли впечатлить коллег и потенциальных работодателей и быстро начать зарабатывать деньги. Вместо того чтобы тоскливо обсуждать каждую деталь, я буду пользоваться принципом «не знаете термин — лезьте в Google».

Файлы примеров для выполнения упражнений из данной книги вы можете бесплатно скачать со страницы репозитория GitHub (https://github.com/markjprice/cs9dotnet5). Инструкции о том, как это сделать, я предоставлю в конце главы 1.

Структура книги

Глава 1 «Привет, C#! Здравствуй, .NET Core!» посвящена настройке среды разработки и использованию Visual Studio Code для разработки простейшего приложения с помощью языка C# и .NET. Вы узнаете, как писать и компилировать код в любой из операционных систем: Windows, macOS и Linux. Для упрощенных консольных приложений вы изучите использование программной функции верх-

него уровня, написанной на C# 9. Вдобавок вы найдете сведения о том, где лучше всего искать справочную информацию.

Глава 2 «Говорим на языке C#» знакомит нас с версиями языка C# и приводит таблицы, помогающие понять, в какой версии были представлены те или иные функциональные особенности языка. Также в этой главе происходит знакомство с грамматикой и лексикой C#, которыми вы будете пользоваться каждый день, создавая исходный код своих приложений. В частности, вы узнаете, как объявлять переменные разных типов и работать с ними, а также о значительных изменениях в C# 8, возникших благодаря введению ссылочных типов, допускающих значение `null` (nullable).

Глава 3 «Управление потоком исполнения и преобразование типов» охватывает использование операторов для выполнения простых действий с переменными, включая сравнения, написание кода, который принимает решения, шаблоны, соответствующие версиям C# 7 — C# 9, повторяет блок операторов и выполняет преобразование между типами. Глава также освещает написание кода в целях защиты от ошибок при их неизбежном возникновении.

Глава 4 «Разработка, отладка и тестирование функций» посвящена соблюдению принципа программирования *DRY* (Do not repeat yourself — «Не повторяйся») при создании многократно используемых функций. Кроме того, вы узнаете, как с помощью инструментов отладки отслеживать и устранять ошибки, мониторить код в процессе его выполнения для диагностики проблем и тщательно тестировать код, чтобы устранять ошибки и обеспечивать его стабильность и надежность, прежде чем развертывать его в производственной среде.

Глава 5 «Создание пользовательских типов с помощью объектно-ориентированного программирования» знакомит с различными категориями элементов, которые может иметь тип, в том числе полями для хранения данных и методами для выполнения действий. Вы будете использовать концепции *объектно-ориентированного программирования (ООП)*, такие как агрегирование и инкапсуляция. Вы изучите языковые функции, такие как поддержка синтаксиса кортежей и переменные `out`, литералы для значений по умолчанию и автоматически определяемые имена элементов кортежей. Также научитесь определять и работать с неизменяемыми типами с помощью ключевого слова `new record`, свойств только для инициализации и выражений, представленных в C# 9.

Глава 6 «Реализация интерфейсов и наследование классов» посвящена созданию новых типов из существующих с использованием объектно-ориентированного программирования (ООП). Вы узнаете, как определять операторы и локальные функции, делегаты и события, как реализовывать интерфейсы с базовыми и производными классами, как переопределять элементы типа, а также изучите концепции полиморфизма, научитесь создавать методы расширения и поймете, как выполнять приведение классов в иерархии наследования.

Глава 7 «Описание и упаковка типов .NET» представляет версии .NET и содержит таблицы, в которых приведены новые функции, а также типы .NET, соответствующие стандарту .NET, и их отношение к языку C#. Вы узнаете, как разворачивать и упаковывать собственные приложения и библиотеки.

Глава 8 «Работа с распространенными типами .NET» описывает типы, позволяющие вашему коду выполнять типовые практические задачи, такие как управление числами и текстом, хранение элементов в коллекциях и реализация интернационализации.

Глава 9 «Работа с файлами, потоками и сериализация» касается взаимодействия с файловой системой, чтения и записи в файлы и потоки, кодирования текста и форматов сериализации, таких как JSON и XML, включая улучшенную функциональность и производительность классов `System.Text.Json` в .NET 5.

Глава 10 «Защита данных и приложений» научит защите данных с помощью шифрования от просмотра злоумышленниками, а также применению хеширования и цифровых подписей в целях защиты от вмешательства или искажения данных. Вы также узнаете об аутентификации и авторизации для защиты приложений от неавторизованных пользователей.

Глава 11 «Работа с базами данных с помощью Entity Framework Core» научит чтению и записи данных в базы данных, такие как Microsoft SQL Server и SQLite, с использованием технологии *объектно-реляционного отображения данных* Entity Framework Core.

Глава 12 «Создание запросов и управление данными с помощью LINQ» рассказывает о языковых расширениях *Language INtegrated Queries (LINQ)*, которые позволяют работать с последовательностями элементов с их последующей фильтрацией, сортировкой и проецированием на различные программные выходы.

Глава 13 «Улучшение производительности и масштабируемости с помощью многозадачности» рассказывает, как одновременно выполнять несколько действий, чтобы повысить эти показатели. Вы узнаете о функции `async Main` и о том, как использовать типы в пространстве имен `System.Diagnostics` в целях мониторинга вашего кода для измерения производительности и эффективности.

Глава 14 «Практическое применение C# и .NET» знакомит с типами кросс-платформенных приложений, которые можно создавать с помощью C# и .NET. Вы также создадите модель сущности для представления базы данных `Northwind`, с которой вы ознакомитесь в главах с 15-й по 21-ю.

Глава 15 «Разработка сайтов с помощью ASP.NET Core Razor Pages» посвящена основам создания сайтов с использованием современной HTTP-архитектуры на стороне сервера с применением ASP.NET Core. Вы научитесь проектировать простые сайты, задействуя функцию ASP.NET Core, известную как Razor Pages, кото-

рая упрощает создание динамических веб-страниц для небольших сайтов. Также вы научитесь создавать запросы и получать ответы HTTP.

Глава 16 «Разработка сайтов с использованием паттерна MVC» посвящена созданию сложных сайтов таким образом, чтобы можно было легко их тестировать и управлять ими с помощью команд, использующих ASP.NET Core MVC. В данной главе также описаны такие темы, как конфигурации запуска, аутентификация, маршруты, модели, представления и контроллеры.

Глава 17 «Разработка сайтов с помощью системы управления контентом (CMS)» описывает, как *система управления контентом* (Content Management System, CMS) позволяет разработчикам быстро создавать сайты с настраиваемым пользовательским интерфейсом администратора, с которым могут работать даже не специалисты, для создания и управления собственным контентом. В качестве примера вы познакомитесь с простой CMS с доступным исходным программным кодом на .NET Core под названием Piranha CMS.

Глава 18 «Разработка и использование веб-сервисов» описывает, как создавать поддерживающие REST веб-сервисы с помощью ASP.NET Core Web API и как правильно потреблять их, задействуя созданные фабрикой HTTP-клиенты.

Глава 19 «Разработка интеллектуальных приложений с помощью алгоритмов машинного обучения» посвящена алгоритмам машинного обучения Microsoft с доступным исходным кодом ML.NET, который можно применять для встраивания адаптивного интеллекта в любое кросс-платформенное приложение .NET, такое как сайт цифровой коммерции, предоставляющий посетителям рекомендации по продукту, который они могут добавить в корзину.

Глава 20 «Создание пользовательских веб-интерфейсов с помощью Blazor» посвящена созданию компонентов пользовательского веб-интерфейса с помощью Blazor, выполняющихся либо на стороне сервера, либо внутри браузера на стороне клиента. Вы проанализируете различия между Blazor Server и Blazor WebAssembly и узнаете, как создавать компоненты, упрощающие переключение между двумя моделями размещения.

Глава 21 «Разработка кросс-платформенных мобильных приложений» посвящена разработке кросс-платформенных веб-приложений для платформ iOS и Android. Приложение, описанное в данной главе, будет разработано с помощью Visual Studio 2019 для Mac с операционной системой macOS.

Приложение А «Ответы на вопросы» содержит ответы на вопросы тестов, приведенных в конце каждой главы.

Приложение Б «Создание настольных приложений для Windows» знакомит вас с работой .NET 5 и его Windows Desktop Pack и с тем, как они позволяют приложениям Windows Forms и WPF эффективно работать на .NET 5. Вдобавок вы изучите основы XAML, который позволяет определить пользовательский

интерфейс для графического приложения для *Windows Presentation Foundation (WPF)* или *Universal Windows Platform (UWP)*. Вы будете применять принципы и функции Fluent Design в целях изучения приложения UWP. Приложения, описанные в этой главе, должны быть разработаны с помощью Visual Studio 2019 в Windows 10.

С приложениями вы можете ознакомиться по следующей ссылке: https://clck.ru/XtWkK.

Необходимое программное обеспечение

Вы можете разрабатывать и разворачивать приложения C# и .NET с использованием Visual Studio Code на многих платформах, включая Windows, macOS и большинство модификаций Linux. Вам необходима операционная система, поддерживающая Visual Studio Code, и подключение к Интернету (главы 1–20).

Вам понадобится операционная система macOS для разработки приложений, описанных в главе 21, поскольку для компиляции приложений iOS вам потребуются macOS и Xcode.

Вам понадобится операционная система Windows 10 для разработки приложений, описанных в приложении Б.

Условные обозначения

В книге вы увидите текст, оформленный различными стилями. Ниже я привел их примеры и объяснил, что означает это форматирование.

Фрагменты кода в тексте, имена таблиц баз данных, файлов, расширения файлов, пути, ввод пользователя, а также учетные записи в Twitter оформляются следующим образом: «Папки Controllers, Models и Views содержат классы ASP.NET Core и файлы `.cshtml` для выполнения на сервере».

Блок кода выглядит так:

```
// хранение элементов в индексированных позициях
names[0] = "Kate";
names[1] = "Jack";
names[2] = "Rebecca";
names[3] = "Tom";
```

Если нужно обратить ваше внимание на определенную часть приведенного кода, то соответствующие строки или элементы будут выделены **полужирным моноширинным** шрифтом:

```
// хранение элементов в индексированных позициях
names[0] = "Kate";
names[1] = "Jack";
names[2] = "Rebecca";
names[3] = "Tom";
```

Весь ввод/вывод в командной строке дается так:

```
dotnet new console
```

URL и имена папок оформляются шрифтом без засечек. Слова, которые вы видите на экране, например в меню или диалоговых окнах, отображаются в тексте так: «Нажмите кнопку Next (Далее) для перехода к следующему экрану».

Ссылки на внешние источники для более детального ознакомления оформлены таким образом.

Советы и рекомендации экспертов по разработке оформлены так.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Привет, C#! Здравствуй, .NET Core!

Эта глава посвящена настройке среды разработки; сходствам и различиям .NET 5, .NET Core, .NET Framework и .NET Standard; а также тому, как использовать различные инструменты для создания простейших приложений с помощью C# 9 и .NET 5, используя Microsoft Visual Studio Code.

После первой главы эту книгу можно разделить на три части: первая знакомит вас с грамматикой и терминологией языка C#; вторая содержит описание типов, доступных в .NET для создания функций приложения; третья включает примеры распространенных кросс-платформенных приложений, которые вы можете создавать с использованием C# и .NET.

Изучение сложных тем многим людям дается проще методом имитации и повторения, а не чтения детальных теоретических объяснений. Поэтому я тоже не буду перегружать книгу детальным объяснением каждого шага. Идея в том, чтобы дать вам задание написать некий код, собрать приложение и посмотреть, что происходит при запуске.

Вам не нужно будет разбираться, как все работает. Вы поймете это в процессе создания собственных приложений и выйдя за рамки того, чему может научить книга.

Выражаясь словами Самюэля Джонсона, составившего в 1755 году толковый словарь английского языка, я, вероятно, допустил «несколько диких промахов и забавных несуразиц, от которых не может быть свободна ни одна из работ подобной сложности». Я принимаю на себя полную ответственность за них и надеюсь, что вы оцените мою попытку «поймать ветер» и написать книгу о таких быстро развивающихся технологиях, как C# и .NET, и приложениях, которые можно разработать с их помощью.

В этой главе:

- настройка среды разработки;
- знакомство с .NET;
- разработка консольных приложений с использованием Visual Studio Code;
- управление исходным кодом с помощью GitHub;
- поиск справочной информации.

Настройка среды разработки

Прежде чем приступать к программированию, вам нужно настроить редактор кода для C#. Microsoft представляет целое семейство редакторов и *интегрированных сред разработки* (Integrated Development Environment, IDE):

- Visual Studio Code;
- GitHub Codespaces;
- Visual Studio 2019;
- Visual Studio 2019 для Mac.

Использование Visual Studio Code для разработки кросс-платформенных приложений

Microsoft Visual Studio Code — самая современная и упрощенная кросс-платформенная среда разработки, созданная корпорацией Microsoft. Ее можно запустить во всех распространенных операционных системах, включая Windows, macOS и множество разновидностей Linux, таких как *Red Hat Enterprise Linux (RHEL)* и Ubuntu.

Visual Studio Code — хороший выбор для современной кросс-платформенной разработки, поскольку имеет обширный и развивающийся набор расширений для поддержки многих языков, помимо C#. И, будучи кросс-платформенной и простой средой разработки, Visual Studio Code может быть установлена на всех платформах, на которых будут установлены ваши приложения, для выполнения быстрых правок и прочих задач.

На сегодняшний день Visual Studio Code — наиболее популярная среда разработки. По результатам опроса, проведенного в Stack Overflow 2019 (не включая 2020 год), большинство разработчиков пользуются данной средой (рис. 1.1).

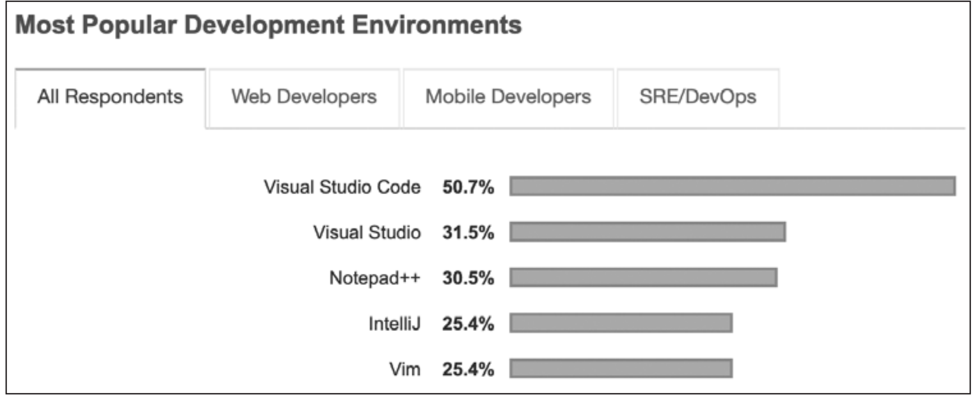


Рис. 1.1. Наиболее популярные среды разработки

Результаты опроса опубликованы на сайте https://insights.stackoverflow.com/survey/2019#developmentenvironments-and-tools.

Visual Studio Code — редактор кода для разработки кросс-платформенных приложений. Поэтому я решил использовать данную программу во всех главах книги, кроме двух последних, где требуются недоступные в Visual Studio Code специальные функции для разработки приложений в операционной системе Windows и мобильных приложений.

О намерениях Microsoft в отношении Visual Studio Code можно прочитать на сайте https://github.com/Microsoft/vscode/wiki/Roadmap.

Если вы предпочитаете использовать Visual Studio 2019 или Visual Studio для Mac вместо Visual Studio Code, то, конечно, это возможно, но я предполагаю, что вы уже знакомы с данными приложениями, и потому не буду давать в книге пошаговые инструкции по их применению.

Информацию, касающуюся Visual Studio Code и Visual Studio 2019, можно найти на сайте https://www.itworld.com/article/3403683/visual-studio-code-stepping-on-visual-studiostoes.html.

Использование GitHub Codespaces для разработки в облаке

GitHub Codespaces — это полностью сконфигурированная среда разработки, основанная на Visual Studio Code, которая может быть развернута в среде, размещенной в облаке, и доступна через любой браузер. GitHub Codespaces поддерживает репозитории Git, расширения и встроенный интерфейс командной строки, поэтому вы можете редактировать, запускать и тестировать код с любого устройства.

Информацию, касающуюся GitHub Codespaces, можно найти на сайте https://docs.github.com/en/github/developing-online-with-codespaces/about-codespaces.

Использование Visual Studio 2019 для разработки Windows-приложений

Программа Microsoft Visual Studio 2019 работает только в операционной системе Windows версии 7 SP1 или более поздней. Для создания приложений *универсальной платформы Windows* (Universal Windows Platform, UWP) вам необходимо

запустить приложение в операционной системе Windows 10. Платформу можно установить из Windows Store и запустить в песочнице для защиты вашего компьютера. Это единственный инструмент Microsoft для разработчиков, который может создавать приложения для Windows, поэтому я буду ссылаться на него в приложении Б, которое доступно в виде PDF-документа по следующей ссылке: https://clck.ru/XtWkK.

Использование Visual Studio на Mac для разработки мобильных приложений

Разработка приложений для устройств под управлением операционной системы iOS и iPadOS требует наличия компьютера с установленной системой macOS и средой Xcode. Вы можете использовать Visual Studio 2019 на Windows с установленными расширениями под Xamarin для написания кросс-платформенного мобильного приложения, но для его компиляции все равно понадобятся macOS и Xcode.

Итак, мы будем использовать программу Visual Studio 2019 для Mac в операционной системе macOS в главе 21.

Рекомендуемые инструменты и операционные системы

В табл. 1.1 приведены инструменты и операционные системы, которые я рекомендую использовать для каждой из глав в этой книге.

Таблица 1.1. Рекомендуемые инструменты и операционные системы

Глава	Среда разработки	Операционная система
1–20	Visual Studio Code	Windows, macOS или Linux
21	Visual Studio 2019 для Mac	macOS
Приложение Б	Visual Studio 2019	Windows 10

Работая над этой книгой, я использовал MacBook Pro и следующее программное обеспечение:

- Visual Studio Code на macOS как мой основной редактор кода;
- Visual Studio Code в операционной системе Windows 10 (на виртуальной машине) для тестирования поведения, специфичного для операционной системы, например, работы с файловой системой;
- Visual Studio 2019 в операционной системе Windows 10 (на виртуальной машине);

- Visual Studio 2019 для Mac в операционной системе macOS для создания мобильных приложений.

Google и Amazon — сторонники среды Visual Studio Code, о чем можно прочитать на сайте https://www.cnbc.com/2018/12/20/microsoft-cmo-capossela-says-google-employees-use-visual-studio-code.html.

Кросс-платформенное развертывание

Выбранные вами среда разработки и используемая операционная система не влияют на то, где могут быть развернуты ваши программы.

Сейчас .NET 5 поддерживает следующие платформы для развертывания:

- *Windows* — Windows 7 SP1 или версии выше; Windows 10 версии 1607 или выше; Windows Server 2012 R2 SP1 или версии выше; Nano Server версии 1809 или выше;
- *Mac* — macOS High Sierra (версия 10.13) или версии выше;
- *Linux* — Alpine Linux 3.11 или версии выше; CentOS или версии выше; Debian 9 или версии выше; Linux Mint 18 или версии выше; openSUSE 15 или версии выше; Red Hat Enterprise Linux (RHEL) 7 или версии выше; SUSE Enterprise Linux 12 SP2 или версии выше; Ubuntu 18.04, 19.10, 20.04 или версии выше.

С официальным списком поддерживаемых операционных систем можно ознакомиться на сайте https://github.com/dotnet/core/blob/master/release-notes/5.0/5.0-supported-os.md.

Поддержка Windows ARM64 в .NET 5 и более поздних версиях означает, что теперь вы можете разрабатывать и развертывать ПО на устройствах Windows ARM, например Microsoft Surface Pro X.

Информацию, касающуюся Windows ARM64, можно найти на сайте https://github.com/dotnet/runtime/issues/36699.

Знакомство с версиями Microsoft Visual Studio Code

Компания Microsoft почти каждый месяц выпускает релиз и обновляет программу Visual Studio Code. Например:

- Version 1.49, август 2020 года, функциональная версия;
- Version 1.49.1, август 2020 года, исправленная версия.

С последними версиями можно ознакомиться на сайте https://code.visualstudio.com/updates.

В книге используется версия 1.49, выпущенная 10 сентября 2020 года. Однако версия Microsoft Visual Studio Code менее важна, чем версия расширения *C# для Visual Studio Code*, которое вы установите позже.

Расширение C# не требуется, однако предоставляет технологию IntelliSense при вводе, навигации по коду и отладке, поэтому его весьма желательно установить. Для поддержки C# 9 вам необходимо установить расширение C# версии 1.23 или выше.

В этой книге я познакомлю вас с сочетаниями клавиш и интерфейсом программы Visual Studio Code в операционной системе macOS. Среды Visual Studio Code для операционных систем Windows и Linux практически идентичны, однако комбинации клавиш, скорее всего, будут отличаться.

В табл. 1.2 перечислены некоторые распространенные сочетания клавиш.

Таблица 1.2. Некоторые распространенные сочетания клавиш

Действие	В macOS	В Windows
Отобразить панель команд	Cmd+Shift+P, F1	Ctrl+Shift+P, F1
Переход к определению	F12	F12
Назад	Ctrl+–	Alt+←
Далее	Ctrl+Shift+–	Alt+→
Отобразить терминал	Ctrl+`	Ctrl+'
Новый терминал	Ctrl+Shift+`	Ctrl+Shift+'
Переключить на строчный комментарий	Ctrl+/	Ctrl+/
Переключить на блочный комментарий	Shift+Option+A	Shift+Alt+A

Я рекомендую вам скачать PDF-файл, в котором описаны сочетания клавиш для вашей операционной системы:

- Windows: https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf;
- macOS: https://code.visualstudio.com/shortcuts/keyboard-shortcuts-macos.pdf;
- Linux: https://code.visualstudio.com/shortcuts/keyboard-shortcuts-linux.pdf.

Узнать о привязках клавиш по умолчанию для Visual Studio Code и о том, как их настроить, можно на сайте https://code.visualstudio.com/docs/getstarted/keybindings.

Среда разработки Visual Studio Code совершенствовалась последние несколько лет благодаря компании Microsoft. Существует также версия Insiders, которая постоянно обновляется.

Скачивание и установка среды Visual Studio Code

Теперь вы готовы скачать и установить среду Visual Studio Code, ее расширение C#, среду .NET 5 и пакет SDK.

1. Скачайте и установите либо стабильную версию Stable, либо версию Insiders среды Visual Studio Code, перейдя по ссылке https://code.visualstudio.com/.
2. Скачайте и установите среду .NET 5 SDK, перейдя по ссылке https://www.microsoft.com/net/download.
3. Для установки расширения C# вам необходимо сначала запустить приложение Visual Studio Code.
4. В Visual Studio Code нажмите значок Extensions (Расширения) или выберите View ▸ Extensions (Вид ▸ Расширения).
5. C# — одно из самых популярных доступных расширений, поэтому вы должны увидеть его в верхней части списка или можете ввести C# в поле поиска (рис. 1.2).
6. Нажмите кнопку Install (Установить) и дождитесь скачивания и установки пакета.

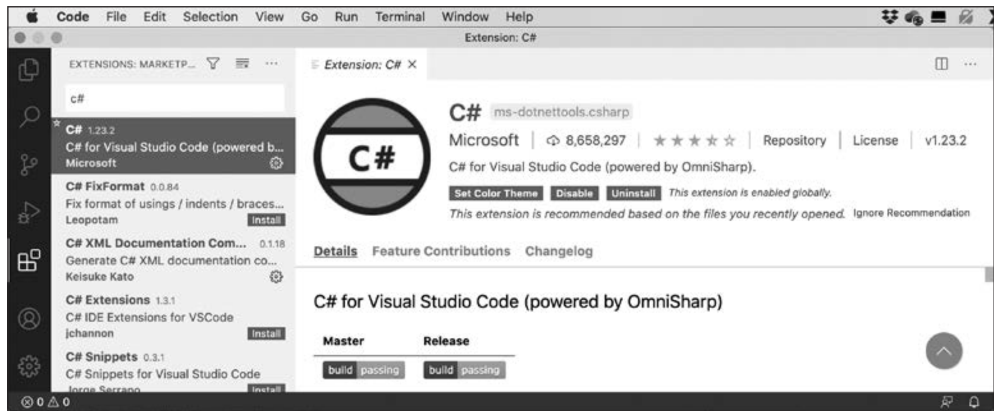


Рис. 1.2. Расширения C#

Более подробную информацию о поддержке средой разработки Visual Studio Code языка C# можно найти на сайте https://code.visualstudio.com/docs/languages/csharp.

Установка расширений

В последующих главах этой книги вы будете использовать дополнительные расширения. В табл. 1.3 представлен их перечень.

Таблица 1.3. Перечень дополнительных расширений

Расширение	**Описание**
C# для Visual Studio Code (работает на OmniSharp) мс-vscode.csharp	Поддержка редактирования C#, включая подсветку синтаксиса, IntelliSense, переход к определению, поиск ссылок, поддержка отладки для .NET и проектов csproj для операционных систем Windows, macOS и Linux
MSBuild tinytoy.msbuild-project-tools	Предоставляет файлы проектов IntelliSense для MSBuild, включая автозаполнение для элементов <PackageReference>
Документация XML в языке C# k--kato.docomment	Создание комментариев документации XML.
REST-клиент humao.rest-client	Отправка HTTP-запроса и просмотр ответа непосредственно в Visual Studio Code
Декомпилятор ILSpy .NET icsharpcode.ilspy-vscode	Декомпиляция сборки MSIL — поддержка среды .NET Framework, .NET Core и .NET Standard

Знакомство с .NET

.NET 5, .NET Framework, .NET Core и Xamarin— связанные и зависимые друг от друга платформы для разработки приложений и сервисов. В данном разделе я познакомлю вас с каждой из этих платформ .NET.

Обзор .NET Framework

.NET Framework — платформа для разработки, включающая *общеязыковую исполняющую среду* (Common Language Runtime, CLR), которая управляет выполнением кода, и обширную *библиотеку базовых классов* (Base Class Library, BCL) для создания приложений. Платформа .NET Framework изначально проектировалась как кросс-платформенная, но впоследствии компания Microsoft сконцентрировалась на внедрении платформы и обеспечении максимально эффективной работы в операционной системе Windows.

Платформа версии .NET Framework 4.5.2 и более поздних версий — официальный компонент операционной системы Windows. .NET Framework установлена на более чем одном миллиарде компьютеров, ввиду чего должна обновляться как можно реже. Даже исправление неполадок может вызвать проблемы, поэтому данная платформа обновляется нечасто.

Все приложения на компьютере, разработанные для .NET Framework, используют одну и ту же версию CLR и библиотек, хранящихся в *глобальном кэше сборок* (Global Assembly Cache, GAC), что может привести к неполадкам, если некоторым из них потребуется определенная версия для совместимости.

В сущности, платформа .NET Framework работает только в среде Windows и считается устаревшей. Не рекомендуется использовать данную платформу для создания новых приложений.

Проекты Mono и Xamarin

Сторонние разработчики создали .NET Framework под названием *Mono*. Хоть он и был кросс-платформенным, но значительно отставал от официальной реализации .NET Framework.

Подробнее о проекте Mono можно прочитать на сайте http://www.mono-project.com.

Проект Mono занял нишу в качестве основы мобильной платформы *Xamarin*, а также кросс-платформенных платформ для разработки игр, таких как *Unity*.

Более подробную информацию о Unity можно получить на сайте https://docs.unity3d.com/.

В 2016 году корпорация Microsoft приобрела компанию Xamarin и теперь бесплатно предоставляет пользователям дорогостоящее решение Xamarin в качестве расширения для среды разработки Visual Studio 2019. Кроме того, корпорация переименовала инструментарий для разработчика *Xamarin Studio* в *Visual Studio для Mac* и внедрила возможность создавать другие типы приложений. В Visual Studio 2019 для Mac корпорация Microsoft заменила части редактора Xamarin Studio на части из Visual Studio для Windows, чтобы обеспечить более знакомый пользовательский опыт и производительность.

Обзор .NET Core

Сегодня мы живем в реально кросс-платформенном мире. Современные методы мобильной и облачной разработки уменьшили прежнюю значимость операционной системы Windows. Поэтому Microsoft активно работает над отделением платформы .NET от Windows, прерыванием их тесных связей. Переписывая код .NET Framework для обеспечения истинной кросс-платформенности, сотрудники корпорации воспользовались моментом, чтобы реорганизовать компоненты .NET и удалить те из них, которые не считаются ядром.

В итоге мир увидел новый продукт — *.NET Core*, включающий кросс-платформенную реализацию рабочей общеязыковой исполняющей среды под названием *CoreCLR* и набор библиотек классов *CoreFX*.

Скотт Хантер, директор .NET-подразделения Microsoft, утверждает: «Сорок процентов пользователей .NET Core — новые разработчики. Мы хотим привлекать новых людей».

Платформа .NET Core быстро развивается и ввиду возможности ее развертывания рядом с приложением может часто меняться, и эти изменения не повлияют на другие приложения .NET Core на той же машине. Обновления, которые Microsoft может внести в .NET Core, нельзя добавить в .NET Framework.

Узнать больше о позиционировании Microsoft .NET Core и .NET Framework можно на сайте https://devblogs.microsoft.com/dotnet/update-on-net-core-3-0-and-net-framework-4-8/.

Обзор .NET 5 и последующих версий .NET

На конференции разработчиков Microsoft Build в мае 2020 года команда .NET объявила, что их планы по унификации .NET отложены. Было объявлено, что версия .NET 5 будет выпущена 10 ноября 2020 года и объединит различные платформы .NET, кроме мобильных. Единая платформа .NET начнет поддерживать мобильные устройства только в .NET 6 в ноябре 2021 года.

Платформа .NET Core будет переименована в .NET, а в номере основной версии будет пропущен номер четыре, чтобы избежать путаницы с NET Framework 4.x. корпорация Microsoft планирует выпускать ежегодные основные версии каждый ноябрь, подобно тому как Apple выпускает основные версии iOS каждый сентябрь.

Более подробную информацию о планах корпорации Microsoft, касающихся версий .NET, можно найти на сайте https://devblogs.microsoft.com/dotnet/announcing-net-5-preview-4-and-our-journey-to-one-net/.

В табл. 1.4 показано, когда были выпущены основные версии .NET, когда запланированы будущие выпуски и какая версия используется в этой книге.

Таблица 1.4. Основные версии .NET

Версия	Дата выпуска	Издание	Выпуск
.NET Core RC1	Ноябрь 2015 года	Первое	Март 2016 года
.NET Core 1.0	Июнь 2016 года		
.NET Core 1.1	Ноябрь 2016 года		
.NET Core 1.0.4 и .NET Core 1.1.1	Март 2017 года	Второе	Март 2017 года
.NET Core 2.0	Август 2017 года		
Обновление .NET Core для UWP в Windows 10 Fall Creators	Октябрь 2017 года	Третье	Ноябрь 2017 года
.NET Core 2.1 (LTS)	Май 2018 года		
.NET Core 2.2 (текущая)	Декабрь 2018 года		
.NET Core 3.0 (текущая)	Сентябрь 2019 года	Четвертое	Октябрь 2019 года
.NET Core 3.1 (LTS)	Декабрь 2019 года		
.NET 5.0 (текущая)	Ноябрь 2020 года	Пятое	Ноябрь 2020 года
.NET 6.0 (LTS)	Ноябрь 2021 года	Шестое	Ноябрь 2021 года

Поддержка .NET Core

Версии .NET сопровождаются либо долгосрочной поддержкой (long term support, LTS), либо текущей (current), как описано ниже.

- Релизы *LTS* стабильны и требуют меньше обновлений в течение срока их службы. Это хороший выбор для приложений, не требующих частых обновлений. Релизы LTS будут поддерживаться в течение трех лет после их выпуска.
- *Текущие* релизы включают функции, которые могут меняться в зависимости от обратной связи. Это хороший выбор для активно разрабатываемых приложений, поскольку предоставляют доступ к последним обновлениям. После трех месяцев обслуживания предыдущая вспомогательная версия больше не будет поддерживаться.

В целях обеспечения безопасности и надежности критические обновления безопасности и надежности поставляются для релизов независимо от режима поддержки. Для получения поддержки вы всегда должны быть в курсе последних обновлений.

Например, если система работает под версией 1.0, а версия 1.0.1 уже выпущена, то, значит, необходимо установить версию 1.0.1 (рис. 1.3).

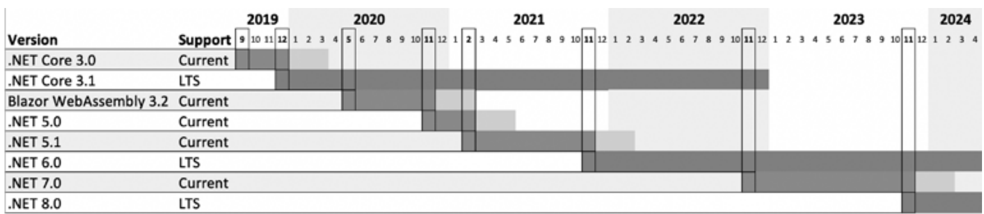


Рис. 1.3. Поддержка различных версий

Например, если вы создаете проект с использованием .NET 5.0 и Microsoft выпускает .NET 5.1 в феврале 2021 года, вам к концу мая 2021 года необходимо будет обновить свой проект до версии .NET 5.1.

При необходимости долгосрочной поддержки со стороны Microsoft в настоящее время выбирайте .NET Core 3.1, а не .NET 5.0. После выпуска .NET 6.0 в ноябре 2021 года у вас будет еще более года поддержки, прежде чем вам придется обновить свой проект до версии .NET 6.0.

Все версии .NET Core достигли конца срока службы, за исключением версий LTS, срок службы которых истек согласно следующему списку:

- срок службы .NET Core 2.1 истекает 21 августа 2021 года;
- срок службы .NET Core 3.1 истекает 3 декабря 2022 года;
- срок службы .NET 6.0 истекает в ноябре 2024 года (при условии, что выйдет в ноябре 2021 года).

Информацию о планах корпорации Microsoft, касающихся версий .NET, можно найти на сайте https://devblogs.microsoft.com/dotnet/announcing-net-5-preview-4-and-our-journey-to-one-net/.

Версии .NET Runtime и .NET SDK

Управление версиями .NET Runtime следует за семантическим управлением версиями, то есть большое приращение указывает на критические изменения, незначительное приращение указывает на новые функции, а приращение исправлений указывает на исправления ошибок.

Управление версиями .NET SDK не следует за семантическим управлением версиями. Старший и дополнительный номера версий привязаны к версии среды выполнения, с которой они совпадают. Номер патча соответствует соглашению,

указывающему основную и вспомогательную версии SDK. Пример приведен в табл. 1.5.

Таблица 1.5. Версии .NET Runtime и .NET SDK

Изменения	Runtime	SDK
Первоначальная версия	5.0.0	5.0.100
SDK, исправленная версия	5.0.0	5.0.101
Runtime и SDK, исправленная версия	5.0.1	5.0.102
SDK, обновленная версия	5.0.1	5.0.200

Информацию, касающуюся версий, можно изучить на сайте https://docs.microsoft.com/ru-ru/dotnet/fundamentals/.

Удаление старых версий .NET

Обновления среды выполнения .NET совместимы с основной версией, такой как 5.x, и обновленные выпуски .NET SDK поддерживают возможность создания приложений, ориентированных на предыдущие версии среды выполнения, что позволяет безопасно удалять старые версии.

Используя следующие команды, вы можете увидеть, какие SDK и среды выполнения в настоящее время установлены:

- `dotnet --list-sdks`;
- `dotnet --list-runtimes`.

Для удаления пакетов SDK для .NET в Windows используйте раздел App & features (Приложение и функции).

В macOS или Windows используйте инструмент dotnet-core-uninstall.

Информацию, касающуюся удаления .NET, можно изучить на сайте https://docs.microsoft.com/ru-ru/dotnet/core/additional-tools/uninstall-tool.

Например, при написании четвертого издания я каждый месяц использовал следующую команду:

```
dotnet-core-uninstall --all-previews-but-latest --sdk
```

Информацию, касающуюся удаления .NET SDK, можно изучить на сайте https://docs.microsoft.com/ru-ru/dotnet/core/install/remove-runtime-sdk-versions.

В чем особенность .NET Core

Современная версия .NET менее объемна, чем текущая версия .NET Framework, из-за того, что устаревшие технологии были удалены. Например, *Windows Forms* и *Windows Presentation Foundation (WPF)* можно использовать для создания приложений с графическим интерфейсом пользователя (graphical user interface, GUI), однако они тесно связаны с экосистемой операционной системы Windows, поэтому были удалены из .NET Core в операционных системах macOS и Linux.

Одна из новых функций .NET 5 — поддержка запуска старых приложений Windows Forms и WPF с помощью пакета *Windows Desktop Pack*, входящего в состав версии. NET 5 для операционной системы Windows, поэтому он считается более полным, чем пакет SDK для операционных систем macOS и Linux. При необходимости вы можете внести небольшие изменения в устаревшее приложение Windows, а затем перестроить его для .NET Core, чтобы воспользоваться новыми функциями и обновлениями. О поддержке разработки таких типов приложений Windows вы узнаете в приложении Б.

Компоненты *ASP.NET Web Forms* и *Windows Communication Foundation (WCF)* представляют собой устаревшие технологии для создания веб-приложений и сервисов, используемые сегодня лишь некоторыми разработчиками, ввиду чего эти компоненты также были удалены из .NET 5. Вместо этого разработчики предпочитают задействовать компоненты ASP.NET MVC и ASP.NET Web-API. Обе технологии были реорганизованы и объединены в новый продукт, *ASP.NET Core*, работающий на платформе .NET 5. Вы узнаете о них в главах 15, 16 и 18.

Некоторые разработчики .NET Framework озабочены тем, что в .NET 5 отсутствуют веб-формы ASP.NET, WCF и Windows Workflow (WF) и надеются, что корпорация Microsoft пересмотрит данный вопрос. Существуют проекты с открытым исходным кодом, позволяющие WCF и WF перейти на .NET 5. Более подробную информацию можно получить на сайте https://devblogs.microsoft.com/dotnet/supporting-the-community-with-wf-and-wcf-oss-projects/. По следующей ссылке для компонентов Blazor Web Forms вы можете найти проект с открытым исходным кодом: https://github.com/FritzAndFriends/BlazorWebFormsComponents.

Entity Framework (EF) 6 — технология объектно-реляционного отображения для работы с информацией, хранящейся в реляционных базах данных, таких как Oracle и Microsoft SQL Server. За годы развития данная технология погрязла в различных доработках, поэтому ее новый кросс-платформенный API был оптимизирован, будет поддерживать нереляционные базы данных, такие как Microsoft Azure Cosmos DB, и называется теперь *Entity Framework Core*. Об этом вы узнаете в главе 11.

Если у вас есть приложения, использующие устаревший EF, то его версия 6.3 поддерживается в .NET Core 3.0 или более поздней.

Несмотря на то что в версии .NET 5 отсутствует слово Core, в ASP.NET Core и Entity Framework Core оно по-прежнему используется, что позволяет отличить технологии от более старых версий. Данную информацию вы можете найти по следующей ссылке: https://docs. microsoft.com/ru-ru/dotnet/core/dotnet-five.

Помимо удаления значительных частей из .NET Framework для создания .NET Core, Microsoft разделила .NET Core на пакеты NuGet, представляющие собой небольшие функциональные части, которые можно развернуть независимо.

Основная цель корпорации Microsoft — не делать ядро .NET меньше, чем .NET Framework. Цель состоит в том, чтобы разбить .NET Core на компоненты для поддержки современных технологий и уменьшения числа зависимостей, чтобы для развертывания нужны были только те пакеты, которые необходимы вашему приложению.

Обзор .NET Standard

В 2019 году с платформой .NET сложилась следующая ситуация: существует три ветви платформы .NET, все из которых разрабатываются корпорацией Microsoft:

- .NET Core — для кросс-платформенных и новых приложений;
- .NET Framework — для устаревших приложений;
- Xamarin — для мобильных приложений.

Все они имеют свои достоинства и недостатки, поскольку предназначены для разных ситуаций. Это привело к тому, что разработчик должен изучить три платформы, каждая из которых раздражает своими странностями и ограничениями. По этой причине Microsoft работает над *.NET Standard* — спецификацией для набора API, которая может быть реализована на всех платформах .NET. Например, указанием на наличие базовой поддержки является совместимость платформы с .NET Standard 1.4.

В рамках .NET Standard 2.0 и более поздних версий корпорация Microsoft привела все три платформы в соответствие с современным минимальным стандартом, что значительно упростило разработчикам совместное использование кода с любой разновидностью .NET.

Это позволило добавить в .NET Core 2.0 и более поздниии версии ряд недостающих API, необходимых разработчикам для переноса старого кода, написанного для

.NET Framework, в кросс-платформенный .NET Core. Однако некоторые API уже реализованы, но при работе выдают исключение о том, что фактически не должны использоваться! Обычно это происходит из-за различий в операционной системе, в которой вы запускаете .NET. Как обрабатывать эти исключения, вы узнаете в главе 2.

Важно понимать, что .NET Standard — это просто стандарт. Вы не можете установить .NET Standard так же, как не можете установить HTML5. Чтобы использовать HTML5, вам необходимо установить браузер, который реализует стандарт HTML5.

Чтобы использовать .NET Standard, необходимо установить платформу .NET, которая реализует спецификацию .NET Standard. .NET Standard 2.0 реализована в последних версиях .NET Framework, .NET Core и Xamarin.

Последняя версия .NET Standard 2.1 реализована только в .NET Core 3.0, Mono и Xamarin. Для некоторых функций C# 8.0 требуется .NET Standard 2.1. Она не реализована в .NET Framework 4.8, поэтому мы рассматриваем .NET Framework как устаревшую.

После выпуска .NET 6 в ноябре 2021 года потребность в .NET Standard значительно уменьшится, поскольку для всех платформ, включая мобильные, будет единая среда разработки .NET. Даже в этом случае приложения и сайты, созданные для .NET Framework, должны будут поддерживаться, поэтому важно понимать, что вы можете создавать библиотеки классов .NET Standard 2.0, обратно совместимые с устаревшими платформами .NET.

Версии .NET Standard и поддерживаемые ими платформы .NET перечислены на сайте https://github.com/dotnet/standard/blob/master/docs/versions.md.

К концу 2021 года корпорация Microsoft обещает создать единую платформу .NET. Планируется, что .NET 6 будет иметь один BCL и две среды выполнения: первую — оптимизированную для серверных или настольных сценариев, таких как сайты и настольные приложения Windows на основе среды выполнения .NET Core, и вторую — оптимизированную для мобильных приложений на основе среды выполнения Xamarin.

Платформы .NET в изданиях этой книги

В первом издании этой книги, написанном в марте 2016 года, я сосредоточился на функциональности .NET Core, но использовал .NET Framework, когда важные или полезные функции еще не были реализованы в .NET Core, поскольку это было еще до окончательного выпуска .NET Core 1.0. Visual Studio 2015 использовался для большинства примеров, Visual Studio Code описывался кратко.

Во втором издании все примеры кода .NET Framework были (почти) полностью убраны, чтобы читатели могли сосредоточиться на примерах .NET Core, которые действительно работают кросс-платформенно.

Третье издание завершило переход. Оно было переписано так, чтобы весь код был чистым .NET Core. Но предоставление пошаговых инструкций как для Visual Studio Code, так и для Visual Studio 2019 привело к ненужному усложнению.

В четвертом издании мы продолжили тенденцию, показывая только примеры кодирования с использованием Visual Studio Code для всех глав, кроме двух последних. Для главы 20 вам нужно было использовать Visual Studio 2019, работающую в операционной системе Windows 10, а в приложении Б — Visual Studio 2019 для Mac.

В этом, пятом издании глава 20 была перенесена в приложение Б, чтобы освободить место для новой главы. Проекты Blazor можно создавать с помощью Visual Studio Code.

В будущем, шестом издании глава 21 будет полностью переписана, чтобы можно было показать создание мобильных и настольных кросс-платформенных приложений с использованием кода Visual Studio и расширения для поддержки .NET MAUI (мультипользовательский интерфейс приложения платформы). Однако следует дождаться шестого издания, так как Microsoft выпустит .NET MAUI с .NET 6 только в ноябре 2021 года. На данном этапе для всех примеров в книге будет использоваться код Visual Studio.

Знакомство с промежуточным языком

Компилятор C# (под названием *Roslyn*), используемый инструментом командной строки `dotnet`, конвертирует ваш исходный код на языке C# в код на *промежуточном языке* (intermediate language, IL) и сохраняет его в *сборке* (DLL- или EXE-файле). Операторы кода на промежуточном языке (IL) похожи на код ассемблера, только выполняются с помощью виртуальной машины CoreCLR в .NET.

В процессе работы код IL загружается CoreCLR из сборки, динамически (just-in-time, JIT) компилируется в инструкции CPU, а затем исполняется с помощью CPU на вашем компьютере. Преимущество такого трехэтапного процесса компиляции заключается в том, что Microsoft может создавать CLR не только для Windows, но и для Linux и macOS. Один и тот же код IL запускается в любой среде благодаря второму процессу компиляции, который генерирует код для конкретной операционной системы и набора команд CPU.

Независимо от того, на каком языке написан исходный код, например C#, Visual Basic или F#, все приложения .NET используют код IL для своих инструкций, хранящихся в сборке. Корпорация Microsoft и другие предоставляют инструменты

дизассемблера, которые могут открывать сборку и раскрывать данный код IL, например расширение декомпилятора ILSpy .NET.

Сравнение технологий .NET

В табл. 1.6 перечисляются и сравниваются технологии .NET 2020 года.

Таблица 1.6. Сравнение технологий .NET

Технология	Возможности	Хостовая ОС
.NET 5	Современный набор функций, полная поддержка C# 9, портирование существующих и создание новых приложений для Windows и веб-приложений/сервисов	Windows, macOS, Linux
.NET Framework	Устаревший набор функций, ограниченная поддержка C# 8.0, отсутствие поддержки C# 9, поддержка существующих приложений	Только Windows
Xamarin	Только для мобильных и настольных приложений	Android, iOS, macOS

Разработка консольных приложений с использованием Visual Studio Code

Цель данного раздела — разработка консольных приложений с помощью среды разработки Visual Studio Code. Инструкции и снимки экрана в этом разделе относятся к операционной системе macOS, но те же действия будут работать с Visual Studio Code в операционных системах Windows и Linux.

Основными отличиями послужат собственные действия командной строки, такие как удаление файла: как команда, так и путь, вероятно, будут разными для операционных систем Windows или macOS и Linux. Однако инструмент командной строки `dotnet` будет одинаковым на всех платформах.

Написание кода с помощью Visual Studio Code

Начнем писать код!

1. Запустите среду разработки Visual Studio Code.
2. В операционной системе macOS выберите File ▸ Open (Файл ▸ Открыть) или нажмите сочетание клавиш Cmd+O. В операционной системе Windows выберите

File ▸ Open Folder (Файл ▸ Открыть папку) или нажмите сочетание клавиш Ctrl+K Ctrl+O. В обеих операционных системах вы можете нажать кнопку Open Folder (Открыть папку) на панели EXPLORER (Проводник) или щелкнуть кнопкой мыши на ссылке Open Folder (Открыть папку) на странице приветствия (рис. 1.4).

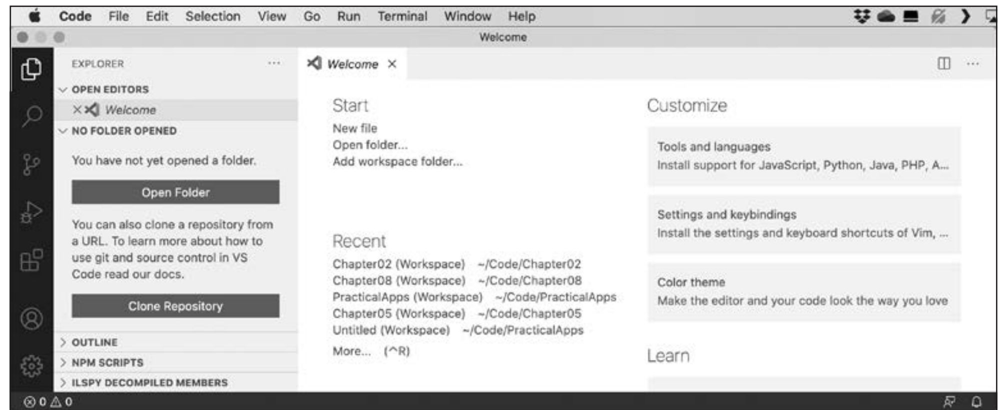


Рис. 1.4. Вкладка Welcome в программе Visual Studio Code

3. В открывшемся диалоговом окне перейдите к своей пользовательской папке в macOS (моя папка называется markjprice), к папке Documents в операционной системе Windows или к любому каталогу или диску, на котором хотите сохранить свои проекты.
4. Нажмите кнопку New Folder (Новая папка) и назовите папку Code.
5. В папке Code создайте подпапку Chapter01.
6. В папке Chapter01 создайте подпапку HelloCS.
7. Выберите папку HelloCS и нажмите кнопку Open (Открыть) в системе macOS или Select Folder (Выбрать папку) в системе Windows.
8. Выберите View ▸ Terminal (Вид ▸ Терминал) или в системе macOS нажмите сочетание клавиш Ctrl+` (обратный апостроф) и в системе Windows нажмите Ctrl+' (одинарная кавычка). В операционной системе Windows вводит в заблуждение комбинация клавиш Ctrl+` (обратный апостроф), которая разделяет текущее окно!
9. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new console
```

10. Вы увидите, что инструмент командной строки `dotnet` создает в текущей папке проект `Console Application`, а на панели EXPLORER (Проводник) отображаются два созданных файла, `HelloCS.proj` и `Program.cs` (рис. 1.5).

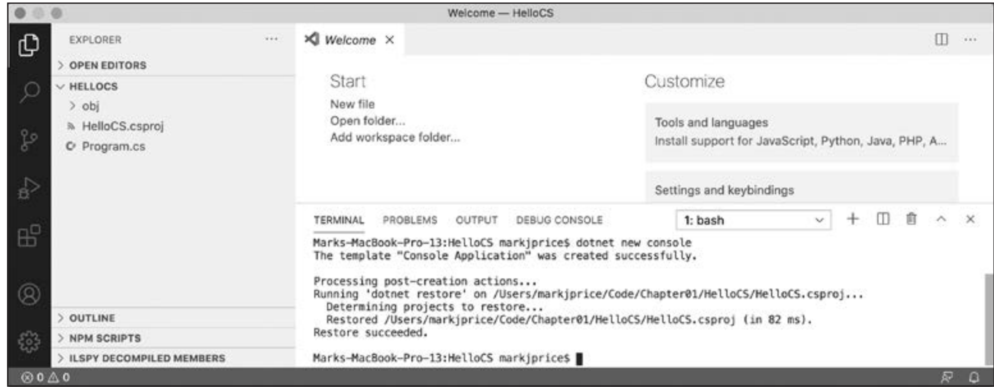


Рис. 1.5. На панели EXPLORER (Проводник) показано, что оба файла созданы.

11. На панели EXPLORER (Проводник) выберите файл `Program.cs`, чтобы открыть его в окне редактора. При первом выполнении Visual Studio Code, возможно, придется загружать и устанавливать зависимые объекты C#, такие как OmniSharp, Razor Language Server и отладчик .NET Core, если это не было сделано при установке расширения C#.
12. Если вы видите предупреждение о том, что необходимые ресурсы отсутствуют, то нажмите кнопку Yes (Да) (рис. 1.6).

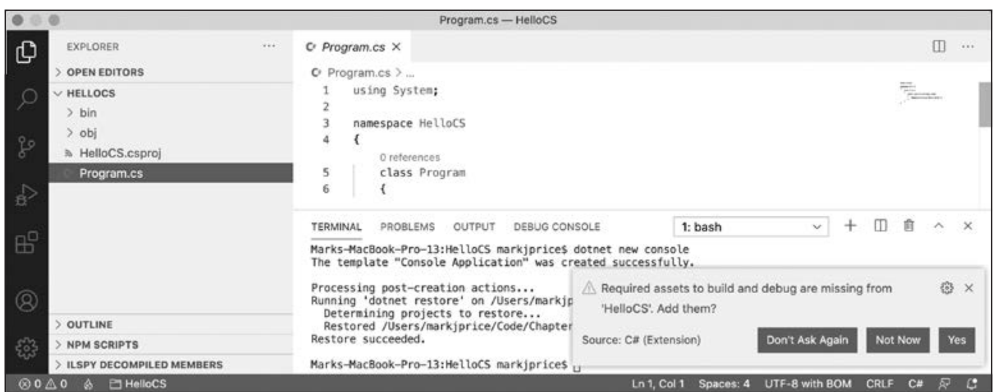


Рис. 1.6. Предупреждающее сообщение для добавления необходимых ресурсов сборки и отладки

13. Через несколько секунд на панели EXPLORER (Проводник) появится папка .vscode. Более подробную информацию вы получите в главе 4.
14. В файле `Program.cs` измените строку 9 так, чтобы текст, который выводится в консоль, сообщал: Hello, C#!.

15. Выберите команду меню File ▸ Auto Save (Файл ▸ Автосохранение). Она избавит вас от необходимости каждый раз вспоминать о сохранении перед повторным обновлением приложения.

Компиляция и запуск кода с использованием инструмента командной строки dotnet

Следующая задача — это компиляция и запуск кода.

1. Выберите View ▸ Terminal (Вид ▸ Терминал) и введите следующую команду:

```
dotnet run
```

2. На панели TERMINAL (Терминал) отобразится результат запуска вашего приложения (рис. 1.7).

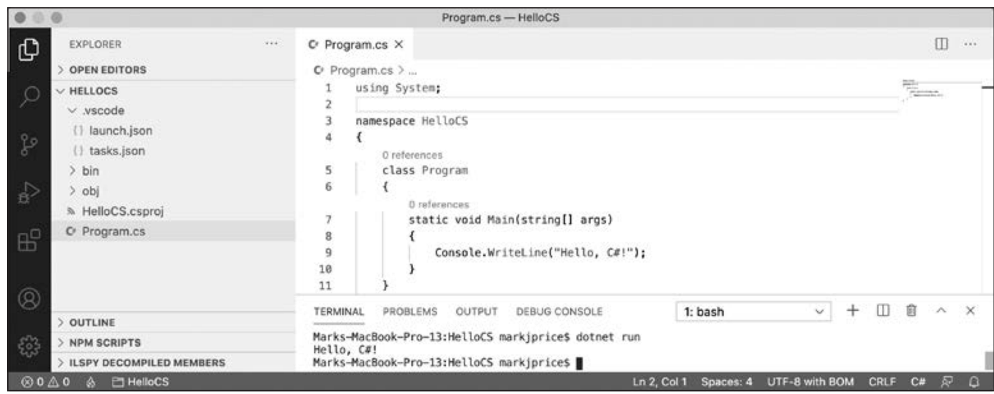


Рис. 1.7. Результат запуска приложения

Написание программ верхнего уровня

Сразу можно подумать, что это слишком большой объем кода только для вывода сообщения Hello, C#!, хотя шаблонный код написан с помощью шаблона проекта. Существует ли более простой способ?

В C# 9 существует, и он известен как *программы верхнего уровня*.

Давайте сравним традиционное минимальное консольное приложение, как показано в следующем коде:

```
using System;

class Program
```

```
{
  static void Main(string[] args)
  {
    Console.WriteLine("Hello World!");
  }
}
```

с новым минимальным консольным приложением программы верхнего уровня, как показано в следующем коде:

```
using System;

Console.WriteLine("Hello World!");
```

Это намного проще, правда? Целесообразнее начать с пустого файла и писать все утверждения самостоятельно.

Во время компиляции весь шаблонный код для определения класса `Program` и его метода `Main` генерируется и выполняется вместе с написанными вами операторами. Любые операторы `using` по-прежнему должны располагаться в начале файла. В проекте может быть только один такой файл.

Лично я, особенно при обучении C#, планирую продолжать использовать традиционный шаблон проекта, поскольку он соответствует действительности. Мне не нравится скрытый код по той же причине, что мне не нравятся графические пользовательские интерфейсы, которые скрывают элементы в попытке упростить работу, но расстраивают пользователей, так как они не могут обнаружить необходимые им функции.

Например, аргументы можно передать в консольное приложение. В программе верхнего уровня вам необходимо знать, что параметр `args` существует, даже если вы его не видите.

Скачивание кода решения из репозитория GitHub

Git — широкоиспользуемая система управления исходным кодом. GitHub — компания, сайт и настольное приложение, которое облегчает работу с Git. Корпорация Microsoft в 2018 году приобрела GitHub, поэтому интеграция GitHub с инструментами Microsoft будет усиливаться.

Я использовал веб-сервис GitHub для хранения ответов ко всем упражнениям, приведенным в конце каждой главы этой книги. Получить к ним доступ можно по ссылке https://github.com/markjprice/cs9dotnet5.

Рекомендую вам добавить предыдущую ссылку в ваши любимые закладки, так как я использую репозиторий GitHub в работе над этой книгой для публикации исправлений и других полезных ссылок.

Использование системы Git в Visual Studio Code

Среда разработки Visual Studio Code имеет встроенную поддержку системы Git, но требует установки Git в используемой операционной системе, поэтому вам необходимо установить Git 2.0 или более позднюю версию, прежде чем получить доступ к этим функциям.

Вы можете скачать дистрибутив Git по ссылке https://git-scm.com/download.

Если хотите использовать графический интерфейс, то можете скачать GitHub Desktop по ссылке https://desktop.github.com.

Клонирование репозитория с примерами из книги

Клонируем репозиторий с примерами из книги.

1. Создайте папку Repos в пользовательской папке, или папке Documents, или там, где вы хотите хранить свои репозитории Git.
2. В программе Visual Studio Code откройте папку Repos.
3. Выберите View ▸ Terminal (Вид ▸ Терминал) и введите следующую команду:

```
git clone https://github.com/markjprice/cs8dotnetcore3.git
```

4. Клонирование всех решений для всех глав займет примерно минуту (рис. 1.8).

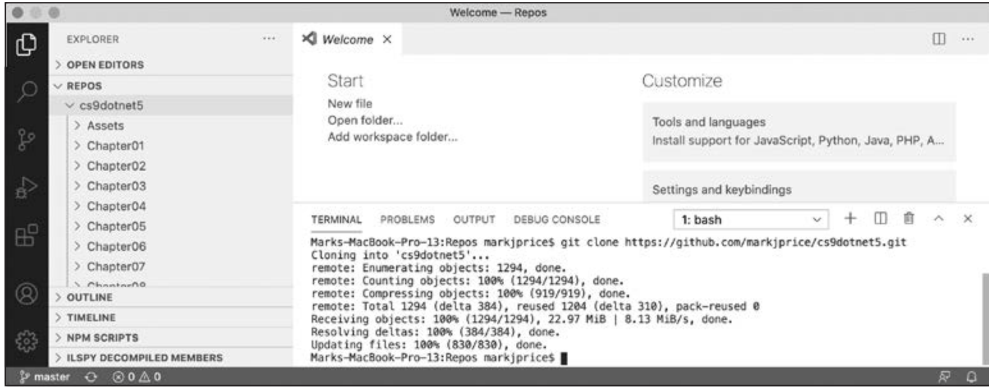


Рис. 1.8. Клонирование репозитория с примерами из книги

Дополнительную информацию об управлении версиями исходного кода с помощью Visual Studio Code можно получить на сайте https://code.visualstudio.com/Docs/editor/versioncontrol.

Поиск справочной информации

Этот раздел главы посвящен тому, как найти достоверную информацию о программировании в Интернете.

Знакомство с Microsoft Docs

Главным ресурсом, позволяющим получить справочные сведения об инструментах и платформах Microsoft, ранее был *Microsoft Developer Network (MSDN)*. Теперь это *Microsoft Docs*, и вы можете найти его по адресу docs.microsoft.com.

Получение справки для инструмента dotnet

В командной строке вы можете запросить справочную информацию об инструменте `dotnet`.

1. Чтобы открыть в окне браузера официальную документацию для команды `dotnet new`, введите в командной строке или на панели TERMINAL (Терминал) программы Visual Studio Code следующую команду:

```
dotnet help new
```

2. Чтобы вывести справочную информацию в командной строке, используйте ключ `-h` или `--help`:

```
dotnet new console -h
```

3. Вы увидите следующий вывод (фрагмент):

```
Console Application (C#)
Author: Microsoft
Description: A project for creating a command-line application that can
run on .NET Core on Windows, Linux and macOS
Options:
  -f|--framework  The target framework for the project.
                      net5.0           - Target net5.0
                      netcoreapp3.1    - Target netcoreapp3.1
                      netcoreapp3.0    - Target netcoreapp3.0
                  Default: net5.0

  --langVersion   Sets langVersion in the created project file
                  text - Optional

  --no-restore    If specified, skips the automatic restore of the project
                  on create.
```

```
                    bool - Optional
                    Default: false / (*) true

  * Indicates the value used if the switch is provided without a value.
```

Получение определений типов и их элементов

Одна из наиболее полезных клавиш в Visual Studio Code — F12 для *перехода к определению*. С помощью данной клавиши вы увидите, как выглядит общедоступное определение типа или элемента, полученное чтением метаданных в скомпилированной сборке. Некоторые инструменты, такие как ILSpy .NET Decompiler, могут даже выполнить реверс-инжиниринг метаданных и кода IL в C#.

Рассмотрим пример использования функции Go To Definition (Переход к определению).

1. В программе Visual Studio Code откройте папку HelloCS.
2. В файле `Program.cs` в методе `Main` введите следующий код, чтобы объявить целочисленную переменную `z`:

   ```
   int z;
   ```

3. Щелкните на слове `int`, а затем нажмите клавишу `F12` или щелкните правой кнопкой мыши и выберите пункт Go To Definition (Перейти к определению) в контекстном меню. В появившемся окне вы можете увидеть определение типа данных `int` (рис. 1.9).

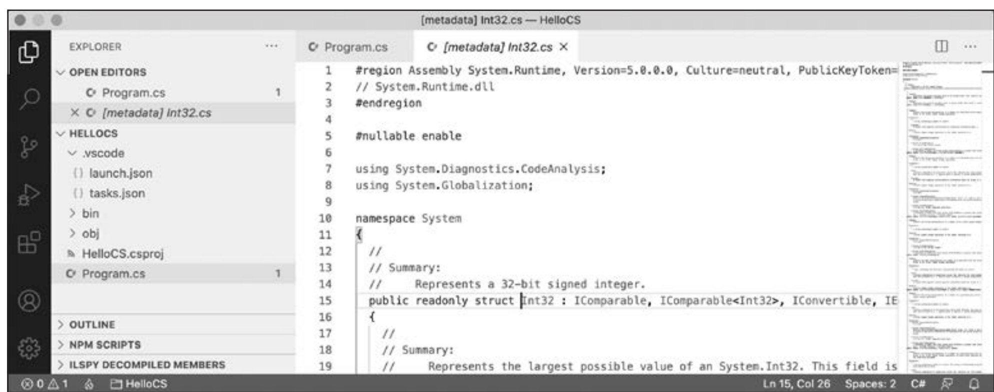


Рис. 1.9. Тип данных int

Вы можете видеть, что `int`:

- определяется с помощью ключевого слова `struct`;
- находится в сборке `System.Runtime`;

- находится в пространстве имен `System`;
- называется `Int32`;
- является, таким образом, псевдонимом для типа `System.Int32`;
- реализует интерфейсы, такие как `IComparable`;
- имеет постоянные значения для своих максимальных и минимальных значений;
- имеет такие методы, как `Parse`.

При попытке использовать команду Go To Definition (Перейти к определению) иногда появляется сообщение об ошибке No definition found (Определение не найдено). Это связано с тем, что расширение C# «не знает» о текущем проекте. Выберите View ▸ Command Palette (Вид ▸ Палитра команд), найдите пункт OmniSharp:Select Project и выберите его, а затем и правильный проект, с которым вы хотите работать.

В настоящее время функция Go To Definition (Перейти к определению) не очень полезна для вас, поскольку вы еще не знаете, что означают эти термины.

К концу первой части книги, в которой рассказывается о C#, вы будете знать достаточно, чтобы эта функция оказалась полезной.

4. Прокрутите окно редактора кода вниз, чтобы найти метод `Parse` с параметром `string`, начинающимся со строки 87 (рис. 1.10).

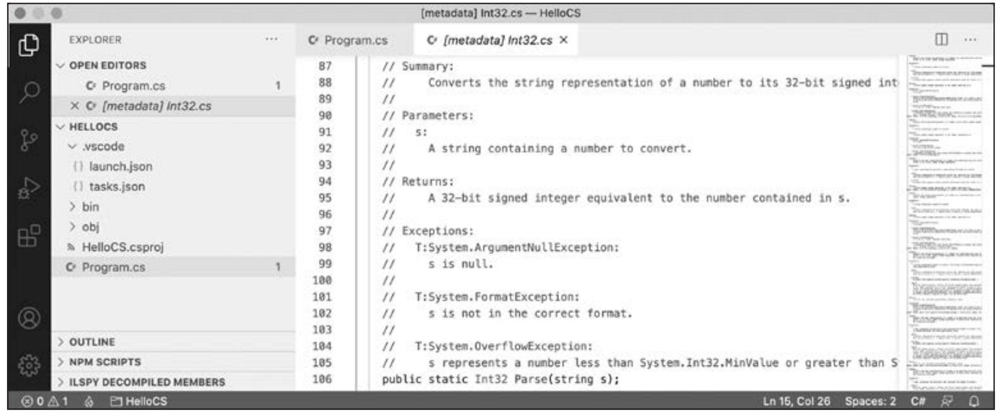


Рис. 1.10. Комментарии к методу Parse

В комментарии вы увидите, что Microsoft задокументировала возникшие при вызове этого метода исключения, включая `ArgumentNullException`, `FormatException` и `OverflowException`. Теперь мы знаем, что нам необходимо обернуть вызов данного метода в оператор `try` и какие исключения необходимо перехватить.

Я надеюсь, вам уже не терпится узнать, что все это значит!

Наберитесь терпения! Вы почти дошли до конца главы и уже в следующей главе погрузитесь в детали языка C#. Но сначала посмотрим, куда еще вы можете обратиться за получением справочной информации.

Ищем ответы на Stack Overflow

StackOverflow — самый популярный сторонний сайт, на котором можно найти ответы на сложные вопросы по программированию. Он настолько популярен, что поисковые системы, такие как *DuckDuckGo*, поддерживают специальный режим поиска на этом сайте.

1. Откройте браузер.
2. Перейдите на сайт DuckDuckGo.com и введите запрос, результаты которого отображены на рис. 1.11.

```
!so securestring
```

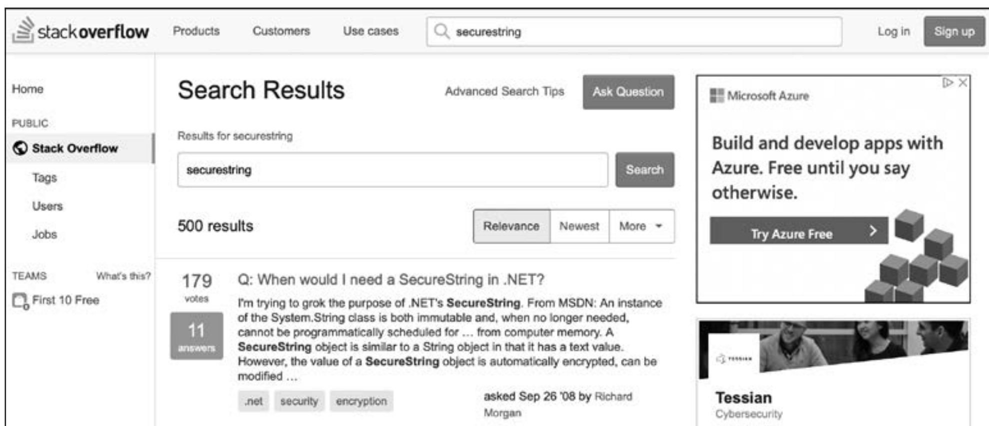


Рис. 1.11. Результаты поиска с переполнением стека для securestring

Поисковая система Google

Вы можете выполнять поиск на сайте *Google*, задавая дополнительные настройки, чтобы увеличить вероятность нахождения нужной информации.

1. Перейдите в Google.
2. Например, если вы ищете в Google информацию о `garbage collection` с помощью обычного запроса, то обнаружите много рекламы, прежде чем увидите ссылку на термин в «Википедии», а также список коммунальных служб.

3. Повысить эффективность поиска можно, ограничив его полезным сайтом, например StackOverflow, и удалив языки, которые могут быть неактуальны в момент поиска, такие как C++, Rust и Python, или добавив C# и .NET, как показано в следующем поисковом запросе:

```
garbage collection site:stackoverflow.com +C# -Java
```

Подписка на официальный блог .NET

Отличный блог, на который рекомендую подписаться, чтобы быть в курсе новостей .NET. Его ведут группы разработчиков .NET. Доступен по адресу https://devblogs.microsoft.com/dotnet/.

Видео от Скотта Хансельмана

У Скотта Хансельмана из Microsoft есть отличный канал на YouTube, благодаря которому вы сможете узнать обо всяких интересных компьютерных «фишках». Рекомендую всем, кто работает в сфере информационных технологий.

Видео Скотта Хансельмана вы можете посмотреть по следующей ссылке: http://computerstufftheydidnteachyou.com/.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 1.1. Проверочные вопросы

Постарайтесь ответить на следующие вопросы, обращая внимание на то, что, невзирая на возможность найти большинство ответов в этой главе, для ответов на другие вопросы потребуется провести некоторые онлайн-исследования или написать код.

1. Почему на платформе .NET Core для разработки приложений программисты могут использовать разные языки, например C# и F#?
2. Какие команды нужно ввести для создания консольного приложения?
3. Какие команды нужно ввести в окне консоли для сборки и запуска исходного кода на языке C#?

4. Какое сочетание клавиш используется для открытия в программе Visual Studio Code панели TERMINAL (Терминал)?
5. Среда разработки Visual Studio 2019 круче, чем Visual Studio Code?
6. Платформа .NET Core лучше .NET Framework?
7. Что такое .NET Standard и почему он так важен?
8. Как называется метод точки входа консольного приложения .NET и как его объявить?
9. Как найти справочную информацию по ключевому слову в C#?
10. Как найти решения общих проблем программирования?

Упражнение 1.2. Практическое задание

Вам не нужно устанавливать среду разработки Visual Studio Code, Visual Studio 2019 или Visual Studio 2019 для Mac, чтобы практиковаться в программировании на языке C#. Посетите сайт .NET Fiddle — dotnetfiddle.net — и начните кодирование в онлайн-режиме.

Упражнение 1.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- документация по Visual Studio Code: https://code.visualstudio.com/docs;
- .NET: https://dotnet.microsoft.com;
- инструменты интерфейса командной строки (CLI) .NET Core: https://aka.ms/dotnet-cli-docs;
- CoreCLR, общеязыковая исполняющая среда .NET Core: https://github.com/dotnet/runtime;
- путеводитель по .NET Core: https://github.com/dotnet/core/blob/master/roadmap.md;
- часто задаваемые вопросы о .NET Standard: https://github.com/dotnet/standard/blob/master/docs/faq.md;
- Stack Overflow: stackoverflow.com;
- расширенный поиск Google: www.google.com/advanced_search;
- виртуальная академия Microsoft: https://docs.microsoft.com/ru-ru/learn/;
- Microsoft Channel 9: видеоуроки для разработчиков: https://channel9.msdn.com/Search?term=.net&lang-en=true.

Резюме

В этой главе мы:

- настроили среду разработки;
- обсудили различия между .NET Framework, .NET 5, .NET Core, Xamarin и .NET Standard;
- научились пользоваться средами разработки Visual Studio Code и .NET Core SDK для создания простого консольного приложения;
- загрузили код решения из данной книги из репозитория GitHub;
- научились находить справочную информацию.

Далее вы научитесь изъясняться на языке C#.

2 Говорим на языке C#

Данная глава посвящена основам языка программирования C# — грамматике и терминологии, которыми вы будете пользоваться ежедневно при разработке своих приложений. Уже к концу главы вы будете чувствовать себя уверенно, зная, как временно хранить информацию в памяти вашего компьютера и работать с ней.

В этой главе:

- введение в C#;
- основы языка C#;
- работа с переменными;
- работа со значениями null;
- дальнейшее изучение консольных приложений.

Введение в C#

Данный раздел книги посвящен языку C# — грамматике и терминологии. Это то, что вы будете использовать каждый день для написания исходного кода своих приложений.

Языки программирования имеют много общего с человеческими, за исключением того, что в языках программирования вы можете создавать собственные слова, как доктор Сьюз!

В книге, написанной доктором Сьюзом в 1950 году, «Если бы у меня был свой зоопарк», он утверждает следующее:

Что дальше? Хочу без затей
Поймать просто невероятных зверей,
Каких вы представить себе не могли,
В стране под названием Гдетовдали!
Смотрите: Махлышка, Павлюн и Свердец,
Носульщик, Уныльщик, Цветец-Размышлец![1]

---

[1] Перевод с англ. Викентия Борисова. Источник: https://stihi.ru/. — *Здесь и далее примеч. пер.*

Обзор версий языка и их функций

Данный раздел книги посвящен языку программирования C# и написан скорее для начинающих, поэтому в нем рассматриваются основные темы, которые должны знать все разработчики: от объявления переменных до хранения данных и определения собственных пользовательских типов данных.

Такие сложные и непонятные темы, как переназначение локальной переменной `ref` и семантика ссылок с типами значений, не рассматриваются.

В книге представлены особенности языка C# от версии 1.0 до последней версии 9.0. На тот случай, если вы уже знакомы с устаревшими версиями C# и желаете узнать о новых функциях в самых последних версиях C#, ниже я привел список языковых версий и их важных новых функций, указав номер главы и название темы.

Более подробную информацию о текущей версии C# вы можете найти по следующей ссылке: https://github.com/dotnet/roslyn/blob/master/docs/Language%20Feature%20Status.md.

C# 1.0

Версия C# 1.0 была выпущена в 2002 году и включала в себя все важные функции статически типизированного объектно-ориентированного современного языка, как приведено в главах 2–6.

C# 2.0

Версия C# 2.0 была выпущена в 2005 году и была нацелена на обеспечение строгой типизации данных с использованием дженериков, для повышения производительности кода и уменьшения количества ошибок типов, включая темы, перечисленные в табл. 2.1.

Таблица 2.1. Темы, затрагивающие C# 2.0, в этой книге

Особенность	Глава	Тема
Типы, допускающие значение null	2	Создание значимого типа, допускающего значение null
Дженерики (обобщения)	6	Создание типов, более пригодных для повторного использования при помощи дженериков

C# 3.0

Версия C# 3.0 вышла в 2007 году и была направлена на включение декларативного программирования с помощью *Language INtegrated Queries (LINQ)* и связанных с ним функций, таких как анонимные типы и лямбда-выражения, включая темы, перечисленные в табл. 2.2.

Таблица 2.2. Темы, затрагивающие C# 3.0, в этой книге

Особенность	Глава	Тема
Неявно типизированные локальные переменные	2	Определение типа локальной переменной
LINQ	12	Все темы, рассматривающиеся в главе 12

C# 4.0

Версия C# 4.0 была выпущена в 2010 году и фокусировалась на улучшении взаимодействия с динамическими языками, такими как F# и Python, включая темы, перечисленные в табл. 2.3.

Таблица 2.3. Темы, затрагивающие C# 4.0, в этой книге

Особенность	Глава	Тема
Динамические типы	2	Тип dynamic
Именованные/необязательные аргументы	5	Необязательные параметры и именованные аргументы

C# 5.0

Версия C# 5.0 вышла в 2012 году и была направлена на упрощение поддержки асинхронных операций за счет автоматической реализации сложных конечных машин, включая темы, перечисленные в табл. 2.4.

Таблица 2.4. Темы, затрагивающие C# 5.0, в этой книге

Особенность	Глава	Тема
Упрощенные асинхронные методы	13	Знакомство с методами async и await

C# 6.0

Версия C# 6.0 была выпущена в 2015 году и ориентирована на незначительные детали языка, включая темы, перечисленные в табл. 2.5.

Таблица 2.5. Темы, затрагивающие C# 6.0, в этой книге

Особенность	Глава	Тема
static (статический импорт)	2	Упрощение использования консоли
Интерполированные строки	2	Отображение вывода для пользователя
Члены класса с телами в виде выражений	5	Определение свойств только для чтения

Более подробную информацию вы можете найти по следующей ссылке: https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/.

C# 7.0

Версия C# 7.0 была выпущена в марте 2017 года и сфокусирована на добавление функциональных языковых возможностей, таких как кортежи и сопоставление с образцом, а также незначительные уточнения языка, включая темы, перечисленные в табл. 2.6.

Таблица 2.6. Темы, затрагивающие C# 7.0, в этой книге

Особенность	**Глава**	**Тема**
Двоичные литералы и разделители цифр	2	Хранение целых чисел
Сопоставление с образцом	3	Сопоставление с образцом, используя оператор if
Переменные out	5	Управление передачей параметров
Кортежи	5	Объединение нескольких значений в кортежи
Локальные функции	6	Определение локальных функций

C# 7.1

Версия C# 7.1 вышла в августе 2017 года и была нацелена на незначительные изменения в языке, включая темы, перечисленные в табл. 2.7.

Таблица 2.7. Темы, затрагивающие C# 7.1, в этой книге

Особенность	**Глава**	**Тема**
Литеральные выражения по умолчанию	5	Установка значений полей с использованием литералов по умолчанию
Автоматически определяемые имена элементов кортежа	5	Вывод имен кортежей
Функция async Main	13	Сокращение времени реагирования для консольных приложений

C# 7.2

Версия C# 7.2 была выпущена в ноябре 2017 года и нацелена на незначительные уточнения в языке, включая темы, перечисленные в табл. 2.8.

Таблица 2.8. Темы, затрагивающие C# 7.2, в этой книге

Особенность	Глава	Тема
Начальное подчеркивание в числовых литералах	2	Хранение целых чисел
Незавершающие именованные аргументы	5	Необязательные параметры и именованные аргументы
Модификатор доступа private protected	5	Описание модификаторов доступа
Операторы == и != с кортежами	5	Сравнение кортежей

C# 7.3

Версия C# 7.3 была выпущена в мае 2018 года и фокусировалась на ориентированном на производительность безопасном коде, который совершенствует переменные `ref`, указатели и команду `stackalloc`. Они редко нужны большинству разработчиков, поэтому не рассматриваются в книге.

Если вы заинтересовались, то можете найти более подробную информацию на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/whats-new/csharp-7-3.

C# 8.0

Версия C# 8.0 была выпущена в сентябре 2019 года и посвящена серьезному изменению языка, связанного с обработкой типов null, включая темы, перечисленные в табл. 2.9.

Таблица 2.9. Темы, затрагивающие C# 8.0, в этой книге

Особенность	Глава	Тема
Ссылочные типы, допускающие значение null	2	Создание ссылочного типа, допускающего значение null
Выражение switch	3	Упрощение операторов switch с помощью выражений switch
Методы интерфейса по умолчанию	6	Описание методов интерфейса по умолчанию

C# 9.0

Версия C# 9.0 была выпущена в ноябре 2020 года и ориентирована на типы записей, уточнения сопоставления с шаблоном и консольные приложения с минимальным кодом, включая темы, перечисленные в табл. 2.10.

Таблица 2.10. Темы, затрагивающие C# 9.0, в этой книге

Особенность	Глава	Тема
Консольные приложения с минимальным кодом	1	Программы верхнего уровня
Улучшенное сопоставление с шаблоном	5	Сопоставление шаблонов с объектами
Записи	5	Работа с записями

Более подробно о новых возможностях C# 9.0 вы можете прочитать на следующем сайте: https://docs.microsoft.com/ru-ru/dotnet/csharp/whats-new/csharp-9.

Версии компилятора C#

С появлением поколения C# 7.x корпорация Microsoft решила ускорить выпуски релизов, поэтому впервые после версии C# 1.1 появились младшие номера версий.

Компиляторы языка .NET для языков C#, Visual Basic и F#, также известные как Roslyn, входят в состав пакета .NET SDK. Чтобы применить определенную версию C#, вы должны иметь установленную версию .NET SDK, одну из перечисленных в табл. 2.11.

Таблица 2.11. Версии компилятора C#

.NET Core SDK	Roslyn	C#
1.0.4	2.0–2.2	7.0
1.1.4	2.3–2.4	7.1
2.1.2	2.6–2.7	7.2
2.1.200	2.8–2.10	7.3
3.0	3.0–3.4	8.0
5.0	5.0	9.0

Более подробно с версиями можно ознакомиться на сайте https://github.com/dotnet/roslyn/blob/master/docs/wiki/NuGet-packages.md.

Посмотрим на имеющиеся версии .NET SDK и компилятора языка C#.

1. Запустите программу Visual Studio Code.
2. Выберите View ▸ Terminal (Вид ▸ Терминал).

3. Для определения имеющейся версии .NET Core SDK введите следующую команду:

```
dotnet –version
```

4. Обратите внимание, что версия на момент написания — .NET Core 5.0.100, что указывает на то, что это начальная версия SDK без каких-либо исправлений ошибок или введения новых функций:

```
5.0.100
```

5. Для определения имеющейся версии компилятора C# введите следующую команду:

```
csc -langversion:?
```

6. Обратите внимание на все версии, доступные на момент написания:

```
Supported language versions:
default
1
2
3
4
5
6
7.0
7.1
7.2
7.3
8.0
9.0 (default)
latestmajor
preview
latest
```

В Windows предыдущая команда возвращает ошибку. Имя csc не распознается как имя команды, функции, файла сценария или исполняемой программы. Проверьте написание имени, а также наличие и правильность пути. Чтобы устранить эту проблему, следуйте инструкциям, приведенным на следующем сайте: https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/compiler-options/command-line-building-with-csc-exe.

Включение версии компилятора на определенном языке

Инструменты разработчика, такие как Visual Studio Code и интерфейс командной строки `dotnet`, предполагают, что вы хотите применять последнюю основную версию компилятора языка C# по умолчанию. Поэтому до выпуска C# 8.0 версия C# 7.0 была последней основной и использовалась по умолчанию. Чтобы за-

действовать обновления доработанных версий C#, таких как 7.1, 7.2 или 7.3, вам необходимо добавить элемент конфигурации в файл проекта следующим образом:

```
<LangVersion>7.3</LangVersion>
```

После выпуска C# 9.0 с .NET 5.0, если корпорация Microsoft выпустит компилятор версии C# 9.1 и вы захотите использовать его новые возможности, вам придется добавить элемент конфигурации в файл проекта следующим образом:

```
<LangVersion>9.1</LangVersion>
```

Потенциальные значения для `<LangVersion>` показаны в табл. 2.12.

Таблица 2.12. Потенциальные значения для <LangVersion>

LangVersion	Описание
7, 7.1, 7.2, 7.3, 8, 9	При вводе определенного номера версии данный компилятор будет использоваться, если был установлен
latestmajor	Использует версию с наибольшим старшим номером, например 7.0 в августе 2019 года, 8.0 в октябре 2019 года, 9.0 в ноябре 2020 года
latest	Использует версию с наибольшими старшим и младшим номерами, например 7.2 в 2017 году, 7.3 в 2018 году, 8 в 2019 году, 9 в 2020 году, возможно, 9.1 в начале 2021 года
preview	Использует самую новую из доступных предварительных версий, например 9.0 в мае 2020 года с установленным .NET 5.0 Preview 4

После создания проекта с помощью инструмента командной строки `dotnet` вы можете отредактировать файл `csproj` и добавить элемент `<LangVersion>` следующим образом:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
</Project>
```

Ваши проекты должны быть нацелены на NET 5.0, чтобы использовать все возможности C# 9.

Если вы еще этого не сделали, установите расширение *MSBuild project tools*. Вы получите `IntelliSense` при редактировании файлов `.csproj`, в том числе упростится добавление элемента `<LangVersion>` с соответствующими значениями.

Более подробно о версии C# 9 вы можете прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/configure-language-version.

Основы языка C#

В процессе изучения языка программирования C# вам нужно будет создать несколько простых приложений. В главах первой части этой книги будет использоваться простейший тип приложения: консольное.

Начнем с изучения основ грамматики и терминологии языка C#. В этой главе вы создадите несколько консольных приложений, каждое из которых будет демонстрировать возможности языка C#. В первую очередь создадим консольное приложение, отображающее версию компилятора.

1. По завершении изучения главы 1 у вас уже должна быть папка Code в вашей пользовательской папке. В противном случае вам нужно ее создать.
2. Создайте вложенную папку Chapter02 с подпапкой Basics.
3. Запустите программу Visual Studio Code и откройте папку Chapter02/Basics.
4. В Visual Studio Code выберите View ▸ Terminal (Вид ▸ Терминал) и введите следующую команду:

```
dotnet new console
```

5. На панели EXPLORER (Проводник) выберите файл `Program.cs`, а затем нажмите кнопку Yes (Да), чтобы добавить недостающие обязательные ресурсы.
6. Откройте файл `Program.cs` и в верхней части файла под оператором `using` добавьте оператор, отображающий текущую версию C# как ошибку, как показано в следующем коде:

```
#error version
```

7. Выберите View ▸ Problems (Вид ▸ Проблемы). Обратите внимание, что версия компилятора и языковая версия отображаются как номер сообщения об ошибке компилятора CS8304 (рис. 2.1).

Рис. 2.1. Версия компилятора и языковая версия отображаются как номер сообщения об ошибке компилятора

8. Закомментируйте оператор, вызывающий ошибку, как показано в следующем коде:

```
// #error version
```

Грамматика языка C#

К грамматике языка C# относятся *операторы* и *блоки*. Вы также можете оставлять пояснения к своему коду с помощью комментариев.

Написание комментариев никогда не должно выступать в качестве единственного способа документирования вашего кода. Для этой цели дополнительно можно прибегнуть к выбору понятных имен для переменных и функций, написанию модульных тестов и созданию настоящих документов.

Операторы

В русском языке мы обозначаем конец повествовательного предложения точкой. Предложение может состоять из разного количества слов и фраз. Порядок слов в предложении тоже относится к правилам грамматики. К примеру, по-русски мы обычно говорим «черный кот», но в опреденных случаях допустимо сказать и «кот черный».

В английском языке прилагательное *«черный»* всегда предшествует существительному *«кот»*: the black cat. Во французском языке порядок иной, прилагательное указывается после существительного: le chat noir. Порядок имеет значение.

Язык C# обозначает конец *оператора* точкой с запятой. При этом оператор может состоять из нескольких *переменных* и *выражений*. Например, для оператора `totalPrice` — это переменная, а `subtotal + salesTax` — выражение:

```
var totalPrice = subtotal + salesTax;
```

Выражение состоит из операнда `subtotal`, операции `+` и второго операнда, `salesTax`. Порядок операндов и операторов имеет значение.

Комментарии

Для описания предназначения кода вы можете добавлять комментарии, предваряя их двумя символами косой черты: `//`. Компилятор игнорирует любой текст после этих символов и до конца строки, например:

```
// налог с продаж должен быть добавлен к промежуточной сумме
var totalPrice = subtotal + salesTax;
```

Среда разработки Visual Studio Code позволяет добавлять и удалять комментарии (двойная косая черта) в начале выбранной строки с помощью сочетаний клавиш Ctrl+K+C и Ctrl+K+U. В операционной системе macOS вместо клавиши Ctrl используйте клавишу Cmd.

Для оформления многострочного комментария используйте символы `/*` в начале комментария и `*/` в его конце:

```
/*
Это многострочный
комментарий.
*/
```

Блоки

В русском языке мы обозначаем абзацы, начиная последующий текст с новой строки. В языке C# *блоки* кода заключаются в фигурные скобки: `{}`. Каждый блок начинается с объявления, описывающего то, что мы определяем. К примеру, блок может определять *пространство имен*, *класс*, *метод* или *оператор*. Со всем этим мы разберемся позднее.

Обратите внимание: в текущем проекте грамматика языка C# уже написана с помощью инструмента командной строки `dotnet`. В следующем примере я добавлю несколько комментариев для описания кода.

```
using System; // точка с запятой указывает на конец оператора

namespace Basics
{ // открывающая фигурная скобка указывает на начало блока
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!"); // оператор
    }
  }
} // закрывающая фигурная скобка указывает на конец блока
```

Терминология языка C#

Терминологический словарь языка C# состоит из *ключевых слов*, *символов* и *типов*.

Среди предопределенных, зарезервированных ключевых слов можно выделить `using`, `namespace`, `class`, `static`, `int`, `string`, `double`, `bool`, `if`, `switch`, `break`, `while`, `do`, `for` и `foreach`.

А вот некоторые из символов: `"`, `'`, `+`, `-`, `*`, `/`, `%`, `@` и `$`.

Изменение цветовой схемы синтаксиса

По умолчанию Visual Studio Code выделяет ключевые слова C# синим цветом, чтобы их было легче отличить от другого кода. Программа позволяет настроить цветовую схему.

1. В Visual Studio Code выберите Code ▸ Preferences ▸ Color Theme (Код ▸ Параметры ▸ Цветовая схема) (находится в меню File (Файл) в операционной системем Windows) или нажмите сочетание клавиш Ctrl+K, Ctrl+T или Cmd+K, Cmd+T.
2. Выберите цветовую схему. Для справки: я буду использовать цветовую схему Light+ (по умолчанию Light), чтобы снимки экрана выглядели четко в напечатанной книге.

Существуют и другие контекстные ключевые слова, имеющие особое значение только в определенном контексте. В языке C# около 100 ключевых слов.

Сравнение языков программирования с естественными языками

В английском языке более 250 тысяч различных слов. Каким же образом языку C# справиться, располагая только сотней ключевых слов? Более того, почему C# так трудно выучить, если он содержит всего 0,0416 % слов по сравнению с количеством в английском языке?

Одно из ключевых различий между человеческим языком и языком программирования заключается в том, что разработчики должны иметь возможность определять новые «слова» с новыми значениями. Помимо 104 ключевых слов на языке C#, эта книга научит вас некоторым из сотен тысяч «слов», определенных другими разработчиками, но вы также узнаете, как определять собственные.

Программисты во всем мире должны изучать английский язык, поскольку большинство языков программирования используют английские слова, такие как namespace и class. Существуют языки программирования, использующие другие человеческие языки, такие как арабский, но они считаются редкими. Если вы заинтересованы в обучении, то можете посмотреть видеоуроки на YouTube, демонстрирующие процесс программирования на арабском языке: https://youtu.be/dkO8cdwf6v8.

Помощь с написанием правильного кода

Простые текстовые редакторы, например приложение Notepad (Блокнот), не помогут правильно писать на английском языке. Анологичным образом это приложение также не поможет вам написать правильный код на языке C#.

Программа Microsoft Word помогает писать без ошибок, подчеркивая ошибки волнистой линией: орфографические — красной (например, icecream следует писать ice-cream или ice cream), а грамматические — синей (например, если предложение должно начинаться с прописной буквы в первом слове).

Расширение C# для Visual Studio Code постоянно отслеживает, как вы набираете код, выделяя орфографические и грамматические ошибки с помощью цветных волнистых линий, аналогично Microsoft Word. Например, имя метода `WriteLine` должно писаться с заглавной буквой `L`, а операторы — заканчиваться точкой с запятой.

Рассмотрим вышесказанное на примере.

1. В файле `Program.cs` замените в методе `WriteLine` прописную букву `L` на строчную.
2. Удалите точку с запятой в конце оператора.
3. Выберите View ▸ Problems (Вид ▸ Проблемы) либо нажмите сочетание клавиш Ctrl+Shift+M или Cmd+Shift+M и обратите внимание, что ошибки кода подчеркиваются красной волнистой линией, а подробная информация отображается на панели PROBLEMS (Проблемы) (рис. 2.2).

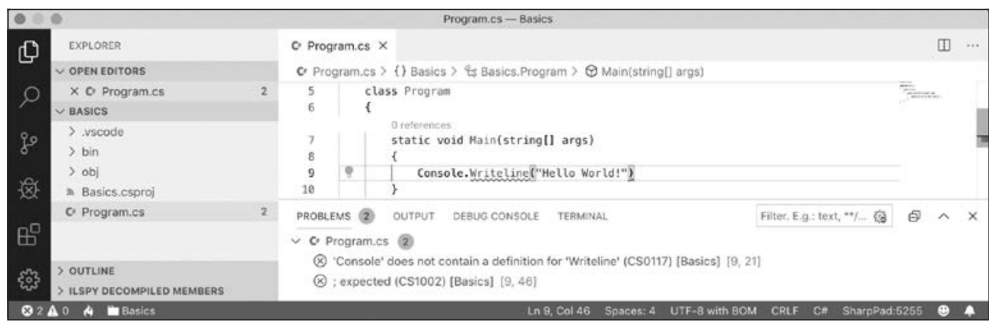


Рис. 2.2. Окно PROBLEMS (Проблемы), отображающее две ошибки компиляции

4. Исправьте две ошибки в коде.

Глаголы = методы

В русском языке глаголы используются для описания действия. В языке C# действия называются *методами*, и их в нем доступны буквально сотни тысяч. В русском языке глаголы меняют форму в зависимости от времени: действие происходило, происходит или будет происходить. К примеру, Амир *попрыгал* в прошлом, Бет *прыгает* в настоящем, они *прыгали* в прошлом, и Чарли *будет прыгать* в будущем.

В языке C# вызов или выполнение методов, таких как `WriteLine`, различаются в зависимости от специфики действия. Это так называемая *перегрузка*, которую мы более подробно изучим в главе 5. Рассмотрим следующий пример:

```
// выводит возврат каретки
Console.WriteLine();

// выводит приветствие и возврат каретки
Console.WriteLine("Hello Ahmed");

// выводит отформатированное число и дату и возврат каретки
Console.WriteLine("Temperature on {0:D} is {1}°C.", DateTime.Today, 23.4);
```

Другая аналогия заключается в том, что некоторые слова пишутся одинаково, но имеют разные значения в зависимости от контекста.

Существительные = типы данных, поля и переменные

В русском языке существительные — это названия предметов или живых существ. К примеру, Шарик — имя (кличка) собаки. Словом «собака» называется тип живого существа с именем Шарик. Скомандовав Шарику принести мяч, мы используем его имя и название предмета, который нужно принести.

В языке C# используются эквиваленты существительных: *типы данных* (чаще называемые просто *типами*), *поля* и *переменные*. Например, `Animal` и `Car` — типы, то есть существительные для категоризации предметов. `Head` и `Engine` — поля, то есть существительные, которые принадлежат `Animal` и `Car`. В то время как `Fido` и `Bob` — переменные, то есть существительные, относящиеся к конкретному предмету.

Для языка C# доступны десятки тысяч типов. Обратите внимание: я сказал не «В языке C# доступны десятки тысяч типов». Разница едва уловимая, но весьма важная. Язык C# содержит лишь несколько ключевых слов для типов данных, таких как `string` и `int`. Строго говоря, C# не определяет какие-либо типы. Ключевые слова наподобие `string` определяют типы как *псевдонимы*, представляющие типы на платформе, на которой запускается C#.

C# не может работать независимо. Это язык приложений, запускаемых на разных платформах .NET. Теоретически можно написать компилятор C# под другую платформу, с другими базовыми типами. На практике платформа для C# — одна из платформ .NET. Это платформа .NET, предоставляющая десятки тысяч типов для C#. Они включают `System.Int32`, к которым относится ключевое слово-псевдоним `int` языка C#, и более сложные типы, такие как `System.Xml.Linq.XDocument`.

Обратите внимание: термин «*тип*» часто путают с «*классом*». Существует такая игра, «*20 вопросов*», в которой первым делом спрашивается категория загаданного предмета: «*животное*», «*растение*» или «*минерал*»? В языке C# каждый *тип* может

быть отнесен к одной из категорий: `class` (класс), `struct` (структура), `enum` (перечисление), `interface` (интерфейс) или `delegate` (делегат). В языке C# ключевое слово `string` — это `class` (класс), но `int` — это `struct` (структура). Поэтому лучше использовать термин *«тип»* для обозначения их обоих.

Подсчет количества типов и методов

Мы знаем, что в языке C# существует более ста ключевых слов, но сколько типов? В нашем простом консольном приложении напишем код, позволяющий подсчитать количество типов и методов, доступных в языке C#.

Не волнуйтесь, если не понимаете, как работает этот код. В нем используется техника под названием «отражение».

1. Начнем с добавления следующих операторов в начале файла `Program.cs`:

```
using System.Linq;
using System.Reflection;
```

2. В методе `Main` удалите оператор, выводящий текст `Hello World!`, и замените его кодом, который показан ниже:

```
// перебор сборок, на которые ссылается приложение
foreach (var r in Assembly.GetEntryAssembly()
  .GetReferencedAssemblies())
{
  // загрузка сборки для чтения данных
  var a = Assembly.Load(new AssemblyName(r.FullName));

  // объявление переменной для подсчета методов
  int methodCount = 0;

  // перебор всех типов в сборке
  foreach (var t in a.DefinedTypes)
  {
    // добавление количества методов
    methodCount += t.GetMethods().Count();
  }
  // вывод количества типов и их методов
  Console.WriteLine(
  "{0:N0} types with {1:N0} methods in {2} assembly.",
  arg0: a.DefinedTypes.Count(),
  arg1: methodCount,
  arg2: r.Name);
}
```

3. Выберите View ▸ Terminal (Вид ▸ Терминал).
4. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet run
```

5. Когда эта команда будет выполнена, вы увидите следующий вывод, в котором отображается фактическое количество типов и методов, доступных вам в простейшем приложении при работе в операционной системе macOS. Количество отображаемых типов и методов может отличаться в зависимости от используемой операционной системы:

```
// Вывод в Windows
0 types with 0 methods in System.Runtime assembly.
103 types with 1,094 methods in System.Linq assembly.
46 types with 662 methods in System.Console assembly.

// Вывод в macOS
0 types with 0 methods in System.Runtime assembly.
103 types with 1,094 methods in System.Linq assembly.
57 types with 701 methods in System.Console assembly.
```

6. Добавьте операторы в начало метода `Main`, чтобы объявить некоторые переменные:

```
static void Main(string[] args)
{
  // объявление некоторых неиспользуемых переменных
  // с помощью типов в дополнительных сборках
  System.Data.DataSet ds;
  System.Net.Http.HttpClient client;
```

Из-за объявления переменных, использующих типы в других сборках, эти сборки загружаются с приложением, что позволяет коду видеть все типы и методы в них. Компилятор предупредит вас о наличии неиспользуемых переменных, но это не остановит выполнение вашего кода.

7. Снова запустите консольное приложение и проанализируйте результаты. Они должны выглядеть примерно так:

```
// Вывод в Windows
0 types with 0 methods in System.Runtime assembly.
376 types with 6,763 methods in System.Data.Common assembly.
533 types with 5,193 methods in System.Net.Http assembly.
103 types with 1,094 methods in System.Linq assembly.
46 types with 662 methods in System.Console assembly.

// Вывод в macOS
0 types with 0 methods in System.Runtime assembly.
376 types with 6,763 methods in System.Data.Common assembly.
522 types with 5,141 methods in System.Net.Http assembly.
103 types with 1,094 methods in System.Linq assembly.
57 types with 701 methods in System.Console assembly.
```

Надеюсь, теперь вы понимаете, почему изучение языка C# — та еще задача. Типов и методов множество, причем методы — только одна категория членов, которую может иметь тип, а программисты постоянно определяют новые члены!

Работа с переменными

Любое приложение занимается обработкой данных. Они поступают, обрабатываются и выводятся. Обычно данные поступают в программы из файлов, баз данных или через пользовательский ввод. Данные могут быть на время помещены в переменные, которые хранятся в памяти работающей программы. Когда она завершается, данные стираются из памяти. Данные обычно выводятся в файлы и базы, на экран или принтер. При использовании переменных вы должны учитывать два фактора. Во-первых, как много объема памяти им требуется, и во-вторых, насколько быстро их можно обработать.

Вы можете управлять этими характеристиками, выбрав определенный тип. Простые распространенные типы, например `int` и `double`, можно представить как хранилище разного размера. Например, добавление 16-битных чисел может обрабатываться не так быстро, как добавление 64-битных чисел в 64-битной операционной системе. Некоторые хранилища можно разместить поближе для быстрого доступа, а другие — убрать подальше, в большое хранилище.

Присвоение переменным имен

Теперь обсудим правила именования переменных, которым рекомендуется следовать. Взгляните на табл. 2.13.

Таблица 2.13. Правила именования переменных

Правило	Примеры	Использование
Верблюжий регистр	cost, orderDetail, dateOfBirth	Локальные переменные и закрытые члены
Прописной стиль	String, Int32, Cost, DateOfBirth, Run	Имена типов, открытые члены, такие как методы

Следование набору соглашений об именах переменных позволит другим разработчикам (и вам самим в будущем!) легко понять ваш код. Дополнительную информацию о правилах присвоения имен переменным можно найти на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/design-guidelines/naming-guidelines.

Блок кода, показанный ниже, отражает пример объявления и инициализации локальной переменной путем присвоения ей значения с символом `=`. Обратите внимание: вы можете вывести имя переменной, используя ключевое слово `nameof`, появившееся в версии C# 6.0:

```
// присвоение переменной heightInMetres значения, равного 1,88
double heightInMetres = 1.88;
Console.WriteLine($"The variable {nameof(heightInMetres)} has the value
{heightInMetres}.");
```

Сообщение в двойных кавычках в предыдущем фрагменте кода было перенесено на вторую строку, поскольку страница этой книги слишком узкая. При вводе подобных операторов в окне редактора кода набирайте их в одну строку.

Литеральные значения

При присвоении значения переменной часто используются *литеральные* значения. Что это такое? Литералами обозначаются фиксированные значения. Типы данных используют разные обозначения для их литеральных значений, и в следующих нескольких разделах вы изучите примеры применения литеральных нотаций для присвоения значений переменным.

Хранение текста

При работе с текстом отдельная буква, например `A`, сохраняется как тип `char` и присваивается с использованием *одинарных* кавычек, оборачивающих литеральное значение:

```
char letter = 'A'; // присваивание литеральных символов
char digit = '1';
char symbol = '$';

char userChoice = GetKeystroke(); // присваивание из функции
```

Если же используется последовательность символов, например слово `Bob`, то такое значение сохраняется как тип `string` и присваивается с использованием *двойных* кавычек, оборачивающих литеральное значение:

```
string firstName = "Bob"; // присваивание литеральных строк
string lastName = "Smith";
string phoneNumber = "(215) 555-4256";

// присваивание строки, возвращаемой функцией
string address = GetAddressFromDatabase(id: 563);
```

Дословные литеральные строки

При сохранении текста в строковой переменной вы можете включить escape-последовательности, которые представляют собой специальные символы, такие как табуляции и новые строки. Это можно сделать с помощью обратного слеша:

```
string fullNameWithTabSeparator = "Bob\tSmith";
```

Подробную информацию об escape-последовательностях можно найти на сайте https://devblogs.microsoft.com/csharpfaq/what-character-escape-sequences-are-available/.

Но что будет, если вы сохраняете путь к файлу и одно из имен папок начинается с буквы T?

```
string filePath = "C:\televisions\sony\bravia.txt";
```

Компилятор преобразует `\t` в символ табуляции и вы получите ошибку!

Вам необходимо использовать префикс `@`, чтобы использовать *дословную (verbatim)* литеральную строку:

```
string filePath = @"C:\televisions\sony\bravia.txt";
```

Подробную информацию о дословных литеральных строках можно прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/tokens/verbatim.

Подытожим:

- *литеральная строка* — символы, заключенные в двойные кавычки. Они могут использовать escape-символы, такие как `\t` для табуляции;
- *дословная литеральная строка* — литеральная строка с префиксом `@` для отключения управляющих символов;
- *интерполированная строка* — литеральная строка с префиксом `$` для включения встроенных форматированных переменных. Вы узнаете больше об этом позже в текущей главе.

Хранение чисел

Числа — данные, которые можно использовать для выполнения арифметических операций, к примеру умножения. Телефонный номер — это не число. Чтобы определиться, какое значение присваивать переменной: числовое или нет, спросите себя, нужно ли вам умножать два телефонных номера или использовать в значении специальные символы, например так: (414)-555-1234. В этих случаях число представляет собой последовательность символов, поэтому должно храниться как строка.

Числа могут быть *натуральными* (например, 42) и использоваться для подсчета, а также отрицательными, например –42. Это *целые числа*. Кроме того, числа могут быть *вещественными* (иначе называемые *действительными*), например 3,9 (с дроб-

ной частью). В программировании они представлены числами *одинарной* и *двойной точности с плавающей запятой*.

Рассмотрим на примере чисел.

1. Создайте папку Numbers внутри папки Chapter02.
2. В программе Visual Studio Code откройте папку Numbers.
3. На панели TERMINAL (Терминал) создайте консольное приложение с помощью команды `dotnet new console`.
4. Внутри метода `Main` введите операторы для объявления некоторых числовых переменных, используя различные типы данных:

```
// целое число без знака означает положительное
// целое число, включая 0
uint naturalNumber = 23;

// целое число означает отрицательное или
// положительное целое число, включая 0
int integerNumber = -23;

// float означает число одинарной точности с плавающей запятой
// суффикс F указывает, что это литерал типа float
float realNumber = 2.3F;

// double означает число двойной точности с плавающей запятой
double anotherRealNumber = 2.3; // литерал типа double
```

Хранение целых чисел

Возможно, вы знаете, что компьютеры хранят все в виде битов. Значение бита равно 0 или 1. Это называется *двоичной системой счисления*. Люди используют *десятичную систему счисления*.

Система десятичных чисел, также известная как Base10, имеет 10 в качестве *основы*, то есть десять цифр от 0 до 9. Несмотря на то что эта система чисел чаще всего используется человеком, в науке, инженерии и вычислительной технике популярны и другие системы счисления. Система двоичных чисел, также известная как Base2, имеет 2 в качестве основы, что означает наличие двух цифр: 0 и 1.

В табл. 2.14 показано, как компьютеры хранят число `10` в двоичной системе счисления. Обратите внимание на бит со значением `1` в столбцах 8 и 2; 8 + 2 = 10.

Таблица 2.14. Хранение целых чисел

128	64	32	16	8	4	2	1
0	0	0	0	1	0	1	0

Таким образом, десятичное число `10` в двоичной системе счисления можно изобразить как `00001010`.

Из двух новых возможностей версии C# 7.0 и более поздних версий одна касается использования символа подчеркивания (_) в качестве разделителя групп разрядов чисел, а вторая внедряет поддержку двоичных литералов. Вы можете использовать символ подчеркивания в любых числовых литералах, включая десятичную, двоичную или шестнадцатеричную систему счисления. Например, вы можете записать значение для миллиона в десятичной системе (Base10) в виде `1_000_000`.

Чтобы использовать запись в двоичной системе (Base2), задействуя только 1 и 0, начните числовой литерал с `0b`. Чтобы применить запись в шестнадцатеричной системе (Base16), используя от 0 до 9 и от A до F, начните числовой литерал с `0x`. Для примера напишем следующий код.

1. В конце метода `Main` введите следующий код, чтобы объявить некоторые числовые переменные, используя знак подчеркивания в качестве разделителя:

```
// три переменные, которые хранят число 2 миллиона
int decimalNotation = 2_000_000;
int binaryNotation = 0b_0001_1110_1000_0100_1000_0000;
int hexadecimalNotation = 0x_001E_8480;

// проверьте, что три переменные имеют одинаковое значение,
// оба оператора выводят true
Console.WriteLine($"{decimalNotation == binaryNotation}");
Console.WriteLine(
  $"{decimalNotation == hexadecimalNotation}");
```

2. Запустите консольное приложение и обратите внимание, что в результате все три числа совпадают:

```
True
True
```

Компьютеры всегда могут точно представлять целые числа, используя тип `int` или один из его родственных типов, например `long` и `short`.

Хранение вещественных чисел

Компьютеры не всегда могут точно представлять числа с плавающей запятой. С помощью типов `float` и `double` можно задавать вещественные числа одинарной и двойной точности с плавающей запятой.

Большинство языков программирования реализуют стандарт IEEE для арифметики с плавающей запятой. IEEE 754 — это технический стандарт для арифметики

с плавающей запятой, установленный в 1985 году Институтом инженеров по электротехнике и электронике (Institute of Electrical and Electronics Engineers, IEEE).

Если хотите подробно ознакомиться с числами с плавающей запятой, то можете прочитать отличный учебник на сайте https://ciechanow.ski/exposing-floating-point/.

В табл. 2.15 показано, как компьютер хранит число `12.75`. Обратите внимание на бит со значением `1` в столбцах 8, 4, 1/2 и 1/4.

8 + 4 + 1/2 + 1/4 = 12 3/4 = 12,75.

Таблица 2.15. Хранение вещественных чисел

128	64	32	16	8	4	2	1	.	1/2	1/4	1/8	1/16
0	0	0	0	1	1	0	0	.	1	1	0	0

Таким образом, десятичное число `12.75` в двоичной системе счисления можно изобразить как `00001100.1100`. Как видите, число `12.75` может быть точно представлено в двоичной системе. Однако такое возможно не для всех чисел. И скоро вы в этом убедитесь.

Пример работы с числами

В языке C# есть *оператор* `sizeof()`, возвращающий количество байтов, используемых в памяти данным типом. Некоторые типы имеют члены с именами `MinValue` и `MaxValue`, возвращающие минимальные и максимальные значения, которые могут храниться в переменной этого типа. Теперь мы собираемся с помощью этих функций создать консольное приложение для изучения типов чисел.

1. В методе `Main` введите операторы, чтобы отобразить размер трех числовых типов данных:

```
Console.WriteLine($"int uses {sizeof(int)} bytes and can store numbers in
the range {int.MinValue:N0} to {int.MaxValue:N0}.");
Console.WriteLine($"double uses {sizeof(double)} bytes and can store
numbers in the range {double.MinValue:N0} to {double.MaxValue:N0}.");
Console.WriteLine($"decimal uses {sizeof(decimal)} bytes and can store
numbers in the range {decimal.MinValue:N0} to {decimal.MaxValue:N0}.");
```

Ширина печатных страниц в этой книге заставляет переносить строковые значения (в двойных кавычках) на несколько строк. Вам необходимо ввести их в одну строку, иначе вы получите ошибки компиляции.

2. Запустите консольное приложение с помощью команды `dotnet run` и проанализируйте результат (рис. 2.3).

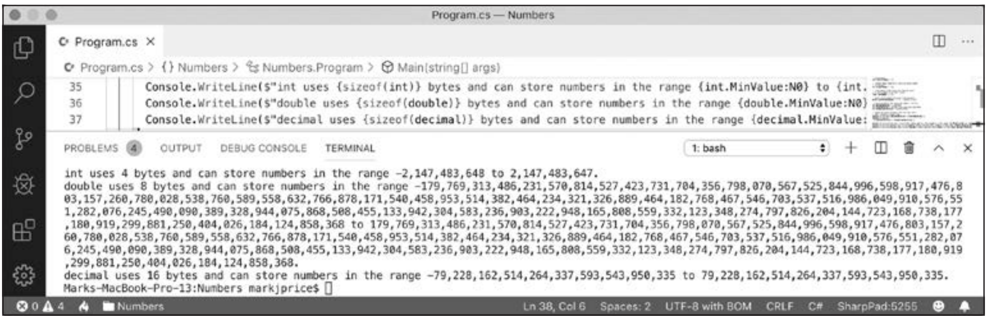


Рис. 2.3. Информация о числовых типах данных

Переменная `int` задействует четыре байта памяти и может хранить положительные или отрицательные числа в пределах до двух миллиардов. Переменная `double` задействует восемь байт памяти и может хранить еще бо́льшие значения! Переменная `decimal` задействует 16 байт памяти и может хранить большие числа, но не настолько большие, как тип `double`.

Почему переменная `double` может хранить бо́льшие значения, чем переменная `decimal`, но при этом задействует вполовину меньше памяти? Разберемся!

Сравнение типов double и decimal

Теперь вы напишете код, чтобы сравнить типы `double` и `decimal`. Не беспокойтесь, если пока не понимаете синтаксис, хотя он и не настолько сложен.

1. После предыдущих операторов введите операторы для объявления двух переменных `double`, суммируйте их и сравните с ожидаемым результатом, а затем запишите результат в консоль:

```
Console.WriteLine("Using doubles:");

double a = 0.1;
double b = 0.2;

if (a + b == 0.3)
{
  Console.WriteLine($"{a} + {b} equals 0.3");
}
else
{
  Console.WriteLine($"{a} + {b} does NOT equal 0.3");
}
```

2. Запустите консольное приложение и проанализируйте результат:

```
Using doubles:
0.1 + 0.2 does NOT equal 0.3
```

Тип `double` не обеспечивает точность, поскольку некоторые числа не могут быть представлены как значения с плавающей запятой.

Почему значение 0,1 не представлено в числах с плавающей запятой, вы узнаете на сайте https://www.exploringbinary.com/why-0-point-1-does-not-exist-in-floating-point/.

Используйте тип `double` только если точность неважна, особенно при сравнении двух чисел; к примеру, при измерении роста человека.

Проблема в предыдущем коде заключается в том, как компьютер хранит число `0,1` или кратное `0,1`. Чтобы представить `0,1` в двоичном формате, компьютер хранит `1` в столбце 1/16, `1` в столбце 1/32, `1` в столбце 1/256, `1` в столбце 1/512 и т. д.

Число `0,1` в десятичной системе представлено как `0.00011001100110011`... с бесконечным повторением (табл. 2.16).

Таблица 2.16. Сравнение типов double и decimal

4	2	1	.	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512	1/1024	1/2048
0	0	0	.	0	0	0	1	0	0	1	0	0	1	0

Никогда не сравнивайте числа двойной точности с плавающей запятой с помощью оператора ==. Во время войны в Персидском заливе американский противоракетный комплекс Patriot был запрограммирован с использованием чисел двойной точности с плавающей запятой в вычислениях. Неточность в расчетах привела к тому, что комплекс не смог перехватить иракскую ракету Р-17 и та попала в американские казармы в городе Дхарам, в результате чего погибли 28 американских солдат; более подробно об этом инциденте можно прочитать на сайте https://www.ima.umn.edu/~arnold/disasters/patriot.html.

3. Скопируйте и вставьте код, который вы написали до использования переменных `double`.
4. Затем измените операторы, чтобы они использовали числа типа `decimal`, и переименуйте переменные в `c` и `d`:

```
Console.WriteLine("Using decimals:");
decimal c = 0.1M; // M обозначает литерал типа decimal
decimal d = 0.2M;
if (c + d == 0.3M)
```

```
{
  Console.WriteLine($"{c} + {d} equals 0.3");
}
else
{
  Console.WriteLine($"{c} + {d} does NOT equal 0.3");
}
```

5. Запустите консольное приложение и проанализируйте результат вывода:

```
Using decimals:
0.1 + 0.2 equals 0.3
```

Тип `decimal` точен, поскольку хранит значение как большое целое число и смещает десятичную запятую. К примеру, 0,1 хранится как 1, с записью, что десятичная запятая смещается на один разряд влево. Число 12,75 хранится как 1275, с записью, что десятичная запятая смещается на два разряда влево.

Тип int используйте для натуральных чисел, а double — для вещественных. Тип decimal применяйте для денежных расчетов, измерений в чертежах и машиностроительных схемах и повсюду, где важна точность вещественных чисел.

Типу `double` присущи некоторые полезные специальные значения. Так, `double.NaN` представляет значение, не являющееся числом, `double.Epsilon` — наименьшее положительное число, которое может быть сохранено как значение `double`, и `double.Infinity` — бесконечно большое значение.

Хранение логических значений

Логическое значение может содержать только одно из двух литеральных значений: или `true` (истина), или `false` (ложь):

```
bool happy = true;
bool sad = false;
```

Логические значения чаще всего используются при ветвлении и зацикливании, как вы увидите в главе 3.

Использование рабочих областей Visual Studio Code

Прежде чем мы создадим еще больше проектов, поговорим о рабочих областях.

Хотя мы могли бы продолжать создавать и открывать отдельные папки для каждого проекта, может быть полезно иметь несколько папок, открытых одновременно.

В среде разработки Visual Studio имеется функция под названием «рабочая область», которая позволяет это выполнить.

Создадим рабочую область для двух проектов.

1. В Visual Studio Code выберите File ▸ Save Workspace As (Файл ▸ Сохранить рабочую область как).
2. Введите текст `Chapter02` в качестве имени рабочей области, перейдите в папку Chapter02 и нажмите кнопку Save (Сохранить) (рис. 2.4).

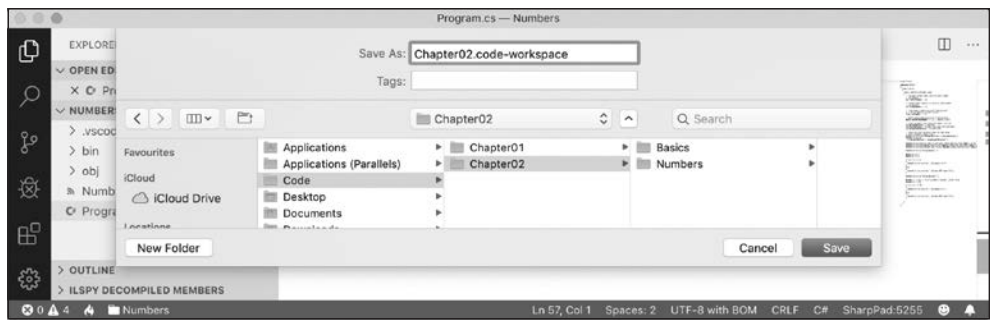


Рис. 2.4. Сохранение рабочей области

3. Выберите команду меню File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область).
4. Выберите папку Basics, нажмите кнопку Add (Добавить) и обратите внимание, что теперь папки Basics и Numbers — часть рабочей области `Chapter02`.

При использовании рабочих областей будьте осторожны при вводе команд на панели TERMINAL (Терминал). Убедитесь, что вы находитесь в правильной папке, прежде чем вводить потенциально деструктивные команды! Мы рассмотрим это в следующем задании.

Хранение объектов любого типа

Специальный тип `object` позволяет хранить данные любого типа, но такая гибкость требует жертв: код получается более сложным и менее производительным. Поэтому по возможности вы должны избегать использования типа `object`.

1. Создайте папку Variables и добавьте ее в рабочую область `Chapter02`.
2. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал).

3. Выберите проект Variables (рис. 2.5).

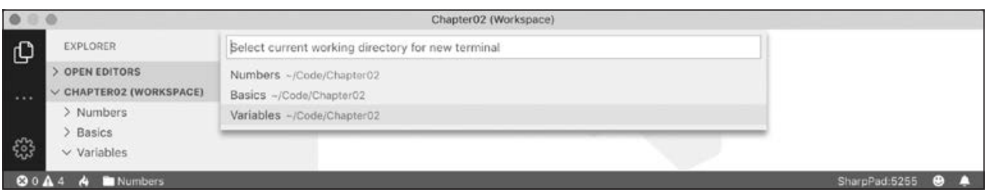


Рис. 2.5. Выбор проекта Variables

4. Введите команду для создания консольного приложения: dotnet new console.
5. Выберите View ▸ Command Palette (Вид ▸ Палитра команд).
6. Перейдите в каталог Chapter02 и выберите пункт OmniSharp: Select Project.
7. Выберите проект Variables и при появлении запроса нажмите кнопку Yes (Да), чтобы добавить необходимые ресурсы для отладки.
8. На панели EXPLORER (Проводник) в проекте Variables откройте файл Program.cs.
9. В методе Main добавьте операторы для объявления и использования некоторых переменных, задействуя тип object:

```
object height = 1.88;              // запись double в object
object name = "Amir";              // запись string в object

Console.WriteLine($"{name} is {height} metres tall.");

int length1 = name.Length;           // выдает ошибку компиляции!
int length2 = ((string)name).Length; // сообщение компилятору,
                                     // что это строка

Console.WriteLine($"{name} has {length2} characters.");
```

10. На панели TERMINAL (Терминал) запустите код с помощью команды dotnet run, и обратите внимание, что четвертый оператор не может скомпилироваться, поскольку тип данных переменной name неизвестен компилятору.
11. Закомментируйте оператор, который не может быть скомпилирован, с помощью двойной косой черты в начале оператора.
12. На панели TERMINAL (Терминал) запустите код с помощью команды dotnet run и обратите внимание, что компилятор может получить доступ к длине строки, если программист явно сообщает компилятору, что переменная object содержит строку:

```
Amir is 1.88 metres tall.
Amir has 4 characters.
```

Тип `object` доступен с самой первой версии языка C#, но в версии C# 2.0 и более поздних в качестве альтернативы используются более эффективные *дженерики*, которые мы рассмотрим позже, в главе 6, и которые обеспечивают желаемую гибкость, не снижая производительности.

Хранение данных динамического типа

Существует еще один специальный тип, `dynamic`, который тоже позволяет хранить данные любого типа и, подобно типу `object`, делает это за счет производительности. Ключевое слово `dynamic` было введено в версии C# 4.0. В отличие от типа `object` для значения, хранящегося в такой переменной, можно осуществлять вызов его членов без явного приведения.

1. В методе `Main` добавьте операторы для объявления переменной `dynamic` и присвойте строковое значение:

```
// хранение строки как dynamic
dynamic anotherName = "Ahmed";
```

2. Добавьте оператор, позволяющий получить длину значения `string`:

```
// компилируется, но может вызвать исключение во время
// выполнения, если вы позже сохраните тип данных,
// у которого нет свойства Length
int length = anotherName.Length;
```

Ограничения типа `dynamic` заключаются в том, что программа Visual Studio Code не отображает меню IntelliSense для набора кода, поскольку компилятор не выполняет проверку во время сборки. Вместо этого общеязыковая исполняющая среда осуществляет проверку члена во время выполнения.

Для сообщения о возникших нарушениях используются исключения. Более подробную информацию об исключениях и о том, как с ними обращаться, вы найдете в главе 3.

Локальные переменные

Локальные переменные объявляются внутри методов и существуют только во время вызова последних. После возвращения метода память, выделенная для хранения любых локальных переменных, освобождается.

Строго говоря, типы значений освобождаются, а ссылочные типы должны ожидать сборку мусора. В чем разница между типами значений и ссылочными типами, вы узнаете в главе 6.

Определение типа локальной переменной

Рассмотрим локальные переменные, объявленные с определенными типами и с использованием определения типов.

1. В методе `Main` введите код для объявления и присвоения значений некоторым локальным переменным, используя определенные типы, как показано ниже:

   ```
   int population = 66_000_000; // 66 миллионов человек в Великобритании
   double weight = 1.88;        // в килограммах
   decimal price = 4.99M;       // в фунтах стерлингов
   string fruit = "Apples";     // строки в двойных кавычках
   char letter = 'Z';           // символы в одиночных кавычках
   bool happy = true;           // логическое значение — true или false
   ```

   Программа Visual Studio Code подчеркнет зеленой волнистой линией имена переменных, значения которым присвоены, но нигде не используются.

   Вы можете использовать ключевое слово `var` для объявления локальных переменных. Компилятор определит тип данных по литеральному значению, введенному вами после оператора присваивания, `=`.

   Числовой литерал без десятичной запятой определяется как переменная `int`, если не добавлен суффикс `L`. В последнем случае определяется переменная `long`.

   Числовой литерал с десятичной запятой определяется как `double`. А если добавить суффикс `M`, то как переменная `decimal`; если `F` — то как `float`. Двойные кавычки обозначают переменную `string`, а одинарные — переменную `char`. Значения `true` и `false` определяют переменную `bool`.

2. Измените свой код так, чтобы использовать ключевое слово `var`:

   ```
   var population = 66_000_000; // 66 миллионов человек в Великобритании
   var weight = 1.88;           // в килограммах
   var price = 4.99M;           // в фунтах стерлингах
   var fruit = "Apples";        // строки в двойных кавычках
   var letter = 'Z';            // символы в одиночных кавычках
   var happy = true;            // логическое значение — true или false
   ```

Несмотря на несомненное удобство ключевого слова var, умные программисты стараются избегать его, чтобы обеспечить читабельность кода и определения типов на глаз. Что касается меня, то я использую это ключевое слово, только если тип и без того ясен. Например, в коде, показанном ниже, первый оператор так же понятен и ясен, как и второй, в котором указывается тип переменной xml, но первый короче. Тем не менее третий оператор неясен на первый взгляд, поэтому лучше использовать четвертый вариант. Если сомневаетесь, то пишите имя типа!

3. В начале файла класса импортируйте несколько пространств имен, как показано в следующем коде:

```
using System.IO;
using System.Xml;
```

4. Для создания некоторых новых объектов под служебными словами добавьте операторы, как показано в следующем коде:

```
// удачное применение var,
// поскольку он избегает повторного типа
var xml1 = new XmlDocument();
XmlDocument xml2 = new XmlDocument();

// неудачное применение var,
// поскольку мы не можем определить тип, поэтому должны использовать
// конкретное объявление типа, как показано во втором выражении
var file1 = File.CreateText(@"C:\something.txt");
StreamWriter file2 = File.CreateText(@"C:\something.txt");
```

Использование целевого типа выражения new для создания экземпляров объектов

В C# 9 Microsoft для создания экземпляров объектов представила другой синтаксис, известный как целевой тип выражения `new`. При создании экземпляра объекта вы можете сначала указать тип, а затем использовать `new`, не повторяя тип, как показано в следующем коде:

```
XmlDocument xml3 = new(); // целевой тип выражения new (версия C# 9)
```

Получение значений по умолчанию для типов

Большинство примитивных типов, кроме `string`, представляют собой *типы значений*. То есть им должны присваиваться значения. Вы можете определить значение по умолчанию типа, используя оператор `default()`.

Строки — *ссылочные типы*. Это значит, они содержат адрес значения в памяти, а не значение самой переменной. Переменная ссылочного типа может иметь значение `null`, которое считается литералом, указывающим, что переменная не ссылается ни на что (пока). `null` — значение по умолчанию для всех ссылочных типов.

Более подробную информацию о типах значений и ссылочных типах вы найдете в главе 6.

Рассмотрим значения по умолчанию.

1. В методе Main добавьте операторы для отображения значений по умолчанию int, bool, DateTime и string:

```
Console.WriteLine($"default(int) = {default(int)}");
Console.WriteLine($"default(bool) = {default(bool)}");
Console.WriteLine($"default(DateTime) = {default(DateTime)}");
Console.WriteLine($"default(string) = {default(string)}");
```

2. Запустите консольное приложение и проанализируйте результат. Обратите внимание, что ваш вывод для даты и времени может быть отформатирован по-другому, если вы не запускаете его в Великобритании, как показано в следующем выводе:

```
default(int) = 0
default(bool) = False
default(DateTime) = 01/01/0001 00:00:00
default(string) =
```

Хранение нескольких значений

Если вам нужно сохранить несколько значений одного и того же типа, то можете объявить массив. В качестве примера сохраним четыре имени в строковом массиве.

Код, показанный ниже, объявляет массив для хранения четырех строковых значений. Затем в нем сохраняются строковые значения с индексами позиций от 0 до 3 (индексация массивов ведется с нуля, поэтому последний элемент всегда на единицу меньше, чем длина массива). И наконец, выполняется перебор каждого элемента в массиве с помощью оператора for, о чем мы более подробно поговорим в главе 3.

Рассмотрим пример использования массива:

1. В папке Chapter02 создайте папку Arrays.
2. Добавьте папку Arrays в рабочую область Chapter02.
3. Создайте новое окно терминала для проекта Arrays.
4. Создайте проект консольного приложения в папке Arrays.
5. Выберите Arrays в качестве текущего проекта для OmniSharp.
6. В проекте Arrays в файле Program.cs в методе Main добавьте операторы для объявления и используйте массив строковых значений:

```
string[] names; // может ссылаться на любой массив строк

// объявление размера массива
string[] names = new string[4];
```

```
// хранение элементов с индексами позиций
names[0] = "Kate";
names[1] = "Jack";
names[2] = "Rebecca";
names[3] = "Tom";

// перебираем имена
for (int i = 0; i < names.Length; i++)
{
  // прочитать элемент с данным индексом позиции
  Console.WriteLine(names[i]);
}
```

7. Запустите консольное приложение и проанализируйте результат:

```
Kate
Jack
Rebecca
Tom
```

Массивы всегда имеют фиксированный размер, поэтому вам нужно предварительно решить, сколько элементов вы хотите сохранить в массиве, прежде чем создавать его.

Массивы удобны для временного хранения нескольких элементов, а *коллекции* предпочтительны при динамическом добавлении и удалении элементов. О коллекциях мы поговорим в главе 8.

Работа со значениями null

Теперь вы знаете, как хранить примитивные значения, например числа в переменных. Но что, если переменная еще не имеет значения? Как мы можем указать это? Язык C# имеет концепцию значения `null`, которое может использоваться для указания того, что переменная не была установлена.

Создание значимого типа, допускающего значение null

По умолчанию типы значений, такие как `int` и `DateTime`, должны всегда иметь значение, что и дало им такое название. Иногда, например при чтении значений, хранящихся в базе данных, которая допускает пустые, отсутствующие или значения `null`, удобно, чтобы тип значения позволял значение `null`. Мы называем это типом, допускающим значение `null`.

Вы можете включить это, добавив знак вопроса в качестве суффикса к типу при объявлении переменной. Рассмотрим на примере.

1. В папке Chapter02 создайте папку NullHandling.
2. Добавьте папку NullHandling в рабочую область Chapter02.
3. Создайте новое окно терминала для проекта NullHandling.
4. Создайте проект консольного приложения в папке NullHandling.
5. Выберите NullHandling в качестве текущего проекта для OmniSharp.
6. В проекте NullHandling в Program.cs в методе Main добавьте операторы для объявления и присваивания значений, включая null, переменным int:

```
int thisCannotBeNull = 4;
thisCannotBeNull = null; // ошибка компиляции!

int? thisCouldBeNull = null;
Console.WriteLine(thisCouldBeNull);
Console.WriteLine(thisCouldBeNull.GetValueOrDefault());

thisCouldBeNull = 7;
Console.WriteLine(thisCouldBeNull);
Console.WriteLine(thisCouldBeNull.GetValueOrDefault());
```

7. Закомментируйте утверждение, которое выдает ошибку компиляции.
8. Запустите приложение и проанализируйте результат, как показано в следующем выводе:

```

0
7
7
```

Первая строка пуста, поскольку выводит значение null!

Ссылочные типы, допускающие значение null

Использование значений null настолько распространено во многих языках, что многие опытные программисты никогда не сомневаются в необходимости его существования. Однако есть много сценариев, в которых мы могли бы написать лучший, более простой код, если переменной не разрешено иметь значение null.

Узнать больше можно, перейдя по следующей ссылке, где изобретатель значений null, сэр Чарльз Энтони Ричард Хоар, признает свою ошибку в записи интервью: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.

Введение ссылочных типов, допускающих и не допускающих значение null, — самое значительное изменение в версии C# 8.0. Вы, вероятно, думаете: «Но подождите! Ссылочные типы уже допускают null!»

И вы были бы правы, однако в версии C# 8.0 и более поздних ссылочные типы можно настроить так, чтобы они больше не допускали значение `null`, установив параметр на уровне файла или проекта, включающий эту новую полезную функцию. Поскольку это весьма значительное изменение для языка C#, корпорация Microsoft решила сделать необходимым явное включение данной функции.

Потребуется несколько лет, чтобы эта новая языковая функция C# дала о себе знать, поскольку все еще есть тысячи библиотек и приложений, работаюих по-старому. Даже корпорация Microsoft не успела полностью реализовать новую функцию во всех основных пакетах .NET 5.

Информацию о достижении 80 % аннотаций в .NET 5 вы можете прочитать по следующей ссылке: https://twitter.com/terrajobst/status/1296566363880742917.

Во время перехода вы можете выбрать один из нескольких подходов для своих проектов:

- *по умолчанию* — никаких изменений не требуется. Ссылочные типы, не допускающие значение `null`, не поддерживаются;
- *включить для проекта, отключить в файлах* — включить функцию на уровне проекта и отключить для всех файлов, которые должны оставаться совместимыми со старыми версиями. К данному подходу прибегает корпорация Microsoft внутри компании, когда обновляет собственные пакеты, чтобы использовать эту новую функцию;
- *включать в файлах* — включить функцию только для отдельных файлов.

Включение ссылочных типов, допускающих и не допускающих значение null

Чтобы включить эту функцию в проекте, добавьте в файл проекта следующий код:

```
<PropertyGroup>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

Чтобы отключить эту функцию в файле, добавьте следующую команду в начало кода:

```
#nullable disable
```

Чтобы включить функцию в файле, добавьте следующую команду в начало кода:

```
#nullable enable
```

Объявление переменных и параметров, не допускающих значение null

Если вы активируете ссылочные типы, допускающие значение `null`, и хотите, чтобы такому типу было присвоено значение `null`, то вам придется использовать тот же синтаксис, что и для типа значения, допускающего значение `null`, то есть добавить символ ? после объявления типа.

Итак, как работают ссылочные типы, допускающие значение `null`? Рассмотрим пример, в котором при хранении информации об адресе может потребоваться ввести значение для улицы, города и региона, но номер здания можно оставить пустым, то есть `null`.

1. В проекте `NullHandling.csproj` добавьте элемент для включения ссылочных типов, допускающих значение `null`, как показано выделением в следующем фрагменте кода:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp5.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

2. В начале файла `Program.cs` добавьте оператор для включения ссылочных типов, допускающих значение `null`:

```
#nullable enable
```

3. В файле `Program.cs` в пространстве имен `NullHandling` над классом `Program` добавьте код для объявления класса `Address` с четырьмя полями:

```
class Address
{
  public string? Building;
  public string Street;
  public string City;
  public string Region;
}
```

4. Обратите внимание: через несколько секунд расширение C# предупреждает о проблемах с полями, не допускающими значение `null`, такими как `Street`, что показано на рис. 2.6.

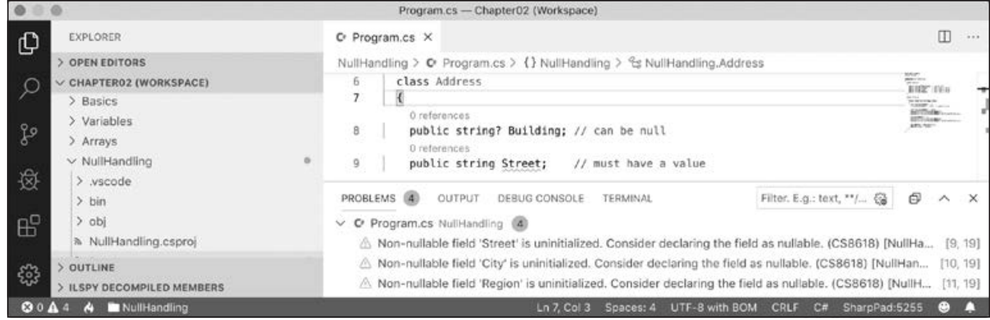


Рис. 2.6. Предупреждающие сообщения о полях, не допускающих значения null, в окне PROBLEMS (Проблемы)

5. Присвойте значение пустой строки каждому из трех полей, не допускающих значение `null`:

```
public string Street = string.Empty;
public string City = string.Empty;
public string Region = string.Empty;
```

6. В методе `Main` добавьте операторы для создания экземпляра `Address` и установки его свойств:

```
var address = new Address();
address.Building = null;
address.Street = null;
address.City = "London";
address.Region = null;
```

7. Обратите внимание на предупреждения (рис. 2.7).

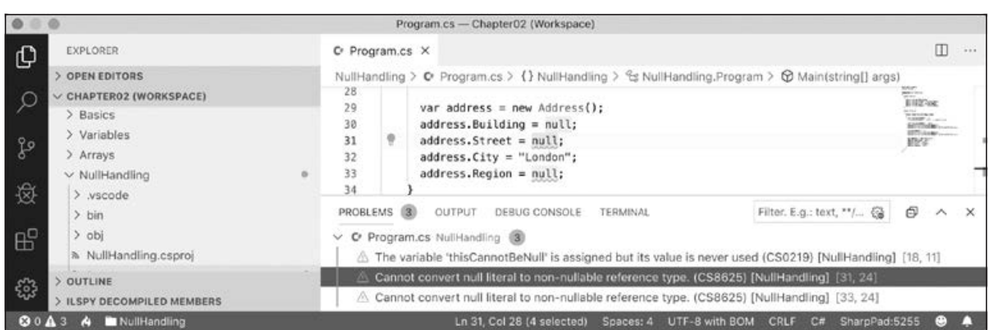


Рис. 2.7. Предупреждающее сообщение о попытке присвоения null полю, не допускающему значения null

Вот почему новая функция языка называется ссылочными типами, допускающими значение `null`. Начиная с версии C# 8.0, ссылочные типы без дополнений могут не допускать это значение, и тот же синтаксис используется для того, чтобы ссылочный тип допускал значение `null`, как и в случае типов значений.

Узнать, как навсегда избавиться от исключений ссылок, допускающих значение null, можно, посмотрев видеоурок на сайте https://channel9.msdn.com/Shows/On-NET/This-is-how-you-get-rid-of-null-reference-exceptions-forever.

Проверка на null

Важно проверять, содержит ли значение `null` переменная ссылочного типа или типа, допускающего значение `null`. В противном случае может возникнуть исключение `NullReferenceException`, которое приведет к ошибке при выполнении кода.

```
// проверить thisCouldBeNull на значение null перед использованием
if (thisCouldBeNull != null)
{
  // получить доступ к thisCouldBeNull
  int length = thisCouldBeNull.Length; // может возникнуть исключение
  ...
}
```

Если вы пытаетесь получить поле или свойство из переменной, которая может быть `null`, используйте оператор доступа к членам с проверкой на `null` (`?.`), как показано в коде, приведенном ниже.

```
string authorName = null;

// выдаст исключение NullReferenceException
int x = authorName.Length;

// вместо вызова исключения, y получит значение null
int? y = authorName?.Length;
```

Более подробную информацию о null-условном операторе можно получить на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/null-conditional-operators.

Иногда требуется либо назначить переменную в качестве результата, либо использовать альтернативное значение, например `3`, если переменная равна `null`. Это достигается с помощью оператора объединения с `null` (`??`), как показано в коде, приведенном ниже.

```
// результат равен 3, если authorName?.Length равен нулю
var result = authorName?.Length ?? 3;
Console.WriteLine(result);
```

Прочитать об операторе объединения с null можно на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/null-coalescing-operator.

Дальнейшее изучение консольных приложений

Мы уже создали и использовали базовые консольные приложения, но сейчас находимся на этапе, когда должны еще больше углубиться в них.

Консольные приложения основаны на тексте и запускаются в командной строке. Обычно они выполняют простые задачи, развивающиеся по наперед заданному сценарию, такие как компиляция файла или шифрование раздела файла конфигурации.

Им можно передавать аргументы для управления поведением. Примером этого может быть создание консольного приложения на языке F# с указанным именем вместо имени текущей папки, как показано в следующей командной строке:

```
dotnet new console -lang "F#" --name "ExploringConsole"
```

Отображение вывода пользователю

Две основные задачи любого консольного приложения заключаются в записи и чтении данных. Мы уже использовали метод `WriteLine` для вывода. Если бы не требовался возврат каретки в конце каждой строки, мы могли бы применить метод `Write`.

Форматирование с использованием пронумерованных позиционных аргументов

Использование пронумерованных позиционных аргументов — один из способов создания форматированных строк.

Эта функция поддерживается такими методами, как `Write` и `WriteLine`, а для методов, которые не поддерживают данную функцию, строковый параметр может быть отформатирован с использованием метода `Format` типа `string`.

1. Добавьте проект консольного приложения `Formatting` в папку и рабочую область `Chapter02`.

2. В методе `Main` добавьте операторы для объявления некоторых числовых переменных и записи их в консоль:

```
int numberOfApples = 12;
decimal pricePerApple = 0.35M;

Console.WriteLine(
    format: "{0} apples costs {1:C}",
    arg0: numberOfApples,
    arg1: pricePerApple * numberOfApples);

string formatted = string.Format(
    format: "{0} apples costs {1:C}",
    arg0: numberOfApples,
    arg1: pricePerApple * numberOfApples);

//WriteToFile(formatted); // записывает строку в файл
```

Метод `WriteToFile` — несуществующий, использованный для демонстрации примера.

Форматирование с использованием интерполированных строк

Версия C# 6.0 и выше содержит удобную функцию *интерполяции строк*. Она позволяет легко выводить одну или несколько переменных в удобном отформатированном виде. Строка с префиксом `$` должна содержать фигурные скобки вокруг имени переменной для вывода текущего значения этой переменной в данной позиции строки.

1. В методе `Main` введите оператор в конце этого метода:

```
Console.WriteLine($"{numberOfApples} apples costs {pricePerApple *
numberOfApples:C}");
```

2. Запустите консольное приложение и проанализируйте результат:

```
12 apples costs £4.20
```

Для коротких отформатированных строк интерполированная строка может быть более простой для чтения. Но для примеров кода в книге, где строки должны переноситься на несколько строк, это может быть сложно. Для многих примеров кода в этой книге я буду использовать пронумерованные позиционные аргументы.

Форматирующие строки

Переменная или выражение могут быть отформатированы с использованием форматирующей строки после запятой или двоеточия.

Код `N0` форматирует число с запятыми в качестве разделителей тысяч и без дробной части. Код `C` форматирует число в значение валюты. Формат последней определяет-

ся текущим потоком. Если вы запустите этот код на компьютере в Великобритании, то значение будет выведено в фунтах стерлингов, а если в Германии — то в евро.

Полный синтаксис форматирующего элемента выглядит следующим образом:

```
{ index [, alignment ] [ : formatString ] }
```

Каждый элемент форматирования можно выравнивать, что полезно при выводе таблиц значений. Некоторые значения необходимо будет выровнять по левому или правому краю в пределах ширины символов. Значения выравнивания — целые числа. При выравнивании по правому краю значения будут положительными целыми числами, а по левому краю — отрицательными целыми числами.

Например, чтобы вывести на печать таблицу фруктов и их количество, мы могли бы выровнять имена в столбце из восьми символов по левому краю и выровнять по правому краю отформатированные числа с нулевыми десятичными знаками в столбце из шести символов.

1. В методе `Main` введите операторы в конце этого метода:

   ```
   string applesText = "Apples";
   int applesCount = 1234;
   string bananasText = "Bananas";
   int bananasCount = 56789;

   Console.WriteLine(
       format: "{0,-8} {1,6:N0}",
       arg0: "Name",
       arg1: "Count");

   Console.WriteLine(
       format: "{0,-8} {1,6:N0}",
       arg0: applesText,
       arg1: applesCount);

   Console.WriteLine(
       format: "{0,-8} {1,6:N0}",
       arg0: bananasText,
       arg1: bananasCount);
   ```

2. Запустите консольное приложение и обратите внимание на эффект выравнивания и числового формата, как показано в следующем выводе:

   ```
   Name      Count
   Apples    1,234
   Bananas  56,789
   ```

Более подробную информацию о типах форматирования в .NET можно прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/base-types/formatting-types.

Получение пользовательского ввода

Мы можем получать ввод от пользователя с помощью метода ReadLine. Он ожидает, пока пользователь не начнет набирать некий текст. После того как тот нажал клавишу Enter, весь пользовательский ввод возвращается как строка.

1. В методе Main введите операторы, чтобы запросить у пользователя имя и возраст, а затем выведите пользовательскую информацию:

```
Console.Write("Type your first name and press ENTER: ");
string firstName = Console.ReadLine();

Console.Write("Type your age and press ENTER: ");
string age = Console.ReadLine();

Console.WriteLine(
  $"Hello {firstName}, you look good for {age}.");
```

2. Запустите консольное приложение.
3. Введите имя и возраст:

```
Type your name and press ENTER: Gary
Type your age and press ENTER: 34
Hello Gary, you look good for 34.
```

Импорт пространства имен

Возможно, вы заметили, что в отличие от нашего самого первого приложения, созданного в главе 1, мы не набирали слово System перед Console. Так мы поступили потому, что System — это пространство имен, как бы «адрес» для типа. Чтобы обратиться к кому-то лично, понадобится использовать код типа Oxford.HighStreet.BobSmith, сообщающий, что надо искать человека по имени Боб Смит на улице Хай-стрит в городе Оксфорд.

Строка System.Console.WriteLine сообщает компилятору, что следует искать метод WriteLine в типе Console в пространстве имен System. Чтобы упростить наш код, команда dotnet new console добавляет в начало файла оператор, сообщающий компилятору, чтобы тот всегда использовал пространство имен System для типов, для которых не были указаны префиксы пространств имен, как показано в коде, приведенном ниже:

```
using System;
```

Так выполняется *импорт пространства имен*. Его результат заключается в том, что все доступные типы в этом пространстве будут доступны для вашей программы без необходимости вводить префикс пространства и будут отображаться в IntelliSense во время написания кода.

Упрощение работы с командной строкой

В языке C# 6.0 и более поздних версиях оператор `using` можно использовать для дальнейшего упрощения кода. Теперь в нашем коде не нужно указывать тип `Console`. Вы можете найти все вхождения и удалить их с помощью функции замены в Visual Studio Code.

1. Добавьте оператор для статического импорта класса `System.Console` в начало файла `Program.cs`:

   ```
   using static System.Console;
   ```

2. Выделите первое слово `Console.` в коде, убедившись, что вы также выбрали точку после слова `Console`.
3. Выберите Edit ▸ Replace (Правка ▸ Заменить) и обратите внимание, что отображается диалоговое окно с наложением, готовое для ввода значения, которым вы хотите заменить вариант `Console.` (рис. 2.8).

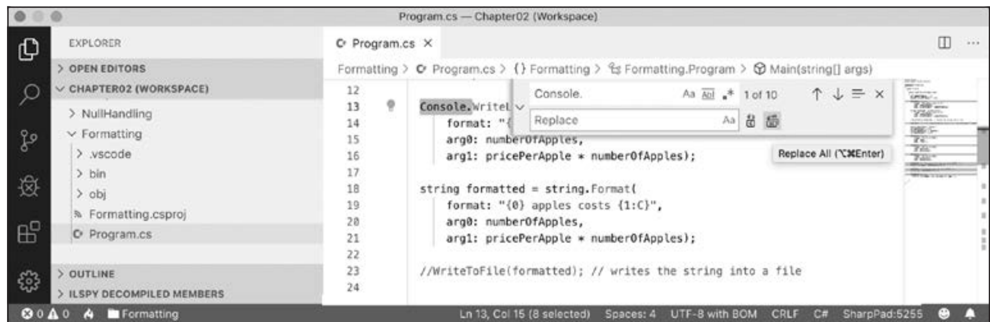


Рис. 2.8. Использование диалогового окна Replace (Заменить) для упрощения кода

4. Нажмите кнопку Replace All (Заменить все) (вторая из двух кнопок справа от поля замены) или сочетание клавиш Alt+A или Alt+Cmd+Enter для замены всех вхождений, затем закройте панель поиска щелчком кнопкой мыши на значке × в правом верхнем углу.

Получение клавиатурного ввода от пользователя

Мы можем получить клавиатурный ввод от пользователя с помощью метода `ReadKey`. Он ожидает, пока пользователь нажмет клавишу или комбинацию клавиш, которая затем возвращается как значение `ConsoleKeyInfo`.

1. В методе `Main` введите код, предлагающий пользователю нажать одну из клавиш (или сочетаний), а затем вывести информацию о ней:

   ```
   Write("Press any key combination: ");
   ConsoleKeyInfo key = ReadKey();
   ```

```
WriteLine();
WriteLine("Key: {0}, Char: {1}, Modifiers: {2}",
arg0: key.Key,
arg1: key.KeyChar,
arg2: key.Modifiers);
```

2. Запустите консольное приложение, нажмите клавишу K и проанализируйте результат:

```
Press any key combination: k
Key: K, Char: k, Modifiers: 0
```

3. Запустите консольное приложение, нажав и удерживая клавишу Shift, нажмите клавишу K и проанализируйте результат:

```
Press any key combination: K
Key: K, Char: K, Modifiers: Shift
```

4. Запустите консольное приложение, нажмите клавишу F12 и проанализируйте результат:

```
Press any key combination:
Key: F12, Char: , Modifiers: 0
```

При запуске консольного приложения на панели TERMINAL (Терминал) программы Visual Studio Code некоторые комбинации клавиш будут захвачены редактором кода или операционной системой, прежде чем они могут быть обработаны вашим приложением.

Чтение аргументов

Вероятно, вам было интересно узнать, что аргумент `string[] args` находится в методе `Main`. Это массив, используемый для передачи аргументов в консольное приложение. Посмотрим, как это работает.

Аргументы командной строки разделяются пробелами. Другие символы, например дефисы и двоеточия, рассматриваются как часть значения аргумента. Чтобы включить пробелы в значение аргумента, заключите значение аргумента в одинарные или двойные кавычки.

Представьте, что требуется возможность вводить имена цветов переднего плана и фона, а также размеры окна терминала в командной строке. Мы могли бы получать цвета и числа, считывая их из массива `args`, который всегда передается в метод `Main` консольного приложения.

1. Создайте папку Arguments для проекта консольного приложения и добавьте ее в рабочую область `Chapter02`.

2. Добавьте оператор для статического импорта типа `System.Console` и оператор для вывода количества аргументов, переданных приложению, как показано ниже:

```
using System;
using static System.Console;

namespace Arguments
{
  class Program
  {
    static void Main(string[] args)
    {
      WriteLine($"There are {args.Length} arguments.");
    }
  }
}
```

Не забывайте статически импортировать тип System.Console во всех будущих проектах для упрощения кода. Эти инструкции я не стану повторять.

3. Запустите консольное приложение и проанализируйте результат:

```
There are 0 arguments.
```

4. На панели TERMINAL (Терминал) введите несколько аргументов после команды `dotnet run`, как показано в следующей командной строке:

```
dotnet run firstarg second-arg third:arg "fourth arg"
```

5. Обратите внимание: результат указывает на четыре аргумента, как показано в следующем выводе:

```
There are 4 arguments.
```

6. Чтобы перечислить или выполнить итерации (то есть перебрать) значений этих четырех аргументов, добавьте следующие операторы после вывода длины массива:

```
foreach (string arg in args)
{
  WriteLine(arg);
}
```

7. На панели TERMINAL (Терминал) повторите те же аргументы после команды `dotnet run`, как показано в следующей командной строке:

```
dotnet run firstarg second-arg third:arg "fourth arg"
```

8. Обратите внимание на подробности о четырех аргументах:

```
There are 4 arguments.
firstarg
second-arg
third:arg
fourth arg
```

Настройка параметров с помощью аргументов

Теперь мы будем применять эти аргументы, чтобы пользователь мог выбрать цвет фона, переднего плана, ширины и высоты окна вывода, а также размер курсора.

Пространство имен `System` уже импортировано, поэтому компилятор знает о типах `ConsoleColor` и `Enum`. Если вы не видите ни одного из этих типов в списке IntelliSense, то это потому, что пропустили оператор `using System;` в начале файла.

1. Добавьте операторы, запрашивающие у пользователя три аргумента, а затем проанализируйте эти аргументы и примените для настройки цвета и размера окна консоли:

```
if (args.Length < 3)
{
  WriteLine("You must specify two colors and cursor size, e.g.");
  WriteLine("dotnet run red yellow 50");
  return; // прекращение запуска
}
ForegroundColor = (ConsoleColor)Enum.Parse(
  enumType: typeof(ConsoleColor),
  value: args[0],
  ignoreCase: true);

BackgroundColor = (ConsoleColor)Enum.Parse(
  enumType: typeof(ConsoleColor),
  value: args[1],
  ignoreCase: true);

CursorSize = int.Parse(args[2]);
```

2. Введите на панели TERMINAL (Терминал) следующую команду:

```
dotnet run red yellow 50
```

В операционной системе Linux это будет работать правильно. В операционной системе Windows код будет работать, но курсор не меняет свой размер.

В операционной системе macOS вы увидите необработанное исключение (рис. 2.9).

Хотя компилятор не выдал ошибку или предупреждение, во время выполнения некоторые API-вызовы могут вызывать ошибку на отдельных платформах. Хотя

консольное приложение, работающее в Linux, может изменять размер курсора, в macOS это невозможно и при попытке выполнить это вы получите ошибку.

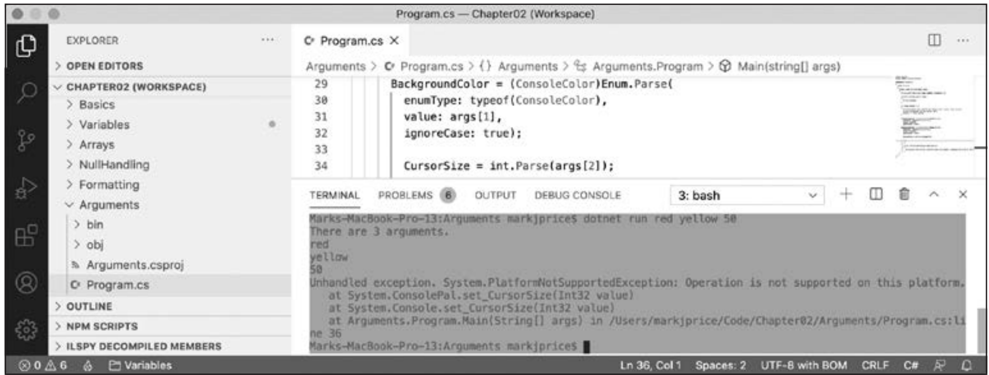


Рис. 2.9. Необработанное исключение в неподдерживаемой операционной системе macOS

Работа с платформами, не поддерживающими некоторые API

Итак, как мы решаем эту проблему? Мы можем решить ее с помощью обработчика исключений. Более подробно об инструкции `try-catch` вы узнаете в главе 3, поэтому сейчас просто введите код.

1. Измените код, чтобы обернуть строки, которые изменяют размер курсора, в оператор `try`:

```
try
{
  CursorSize = int.Parse(args[2]);
}
catch (PlatformNotSupportedException)
{
  WriteLine("The current platform does not support changing
  the size of the cursor.");
}
```

2. Перезапустите консольное приложение. Обратите внимание: исключение перехвачено и пользователю отображено понятное сообщение.

Еще один способ справиться с различиями в операционных системах — использовать класс операционной системы:

```
if (OperatingSystem.IsWindows())
{
  // выполнить код, работающий только в Windows
}
```

Класс `OperatingSystem` содержит эквивалентные методы для других распространенных ОС, таких как Android, iOS, Linux, macOS и даже для браузера, что полезно для веб-компонентов Blazor.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 2.1. Проверочные вопросы

Получить лучший ответ на некоторые из этих вопросов можно, проведя собственное исследование. Я хочу, чтобы вы «мыслили вне книги», поэтому сознательно не предоставил все ответы.

Я хочу научить вас пользоваться дополнительной информацией из других источников.

Какой тип следует выбрать для каждого указанного ниже числа?

1. Телефонный номер.
2. Рост.
3. Возраст.
4. Размер оклада.
5. Артикул книги.
6. Цена книги.
7. Вес книги.
8. Размер населения страны.
9. Количество звезд во Вселенной.
10. Количество сотрудников на каждом из предприятий малого и среднего бизнеса (примерно 50 000 сотрудников на каждом предприятии).

Упражнение 2.2. Практическое задание — числовые размеры и диапазоны

Создайте проект консольного приложения `Exercise02`, которое выводит количество байтов в памяти для каждого из следующих числовых типов, а также минимальное и максимальное допустимые значения: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`.

Ознакомиться с документацией, касающейся составного форматирования, можно на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/base-types/composite-formatting, чтобы разобраться, как выравнивать текст в окне консольного приложения.

Результат работы вашего приложения должен выглядеть примерно так (рис. 2.10).

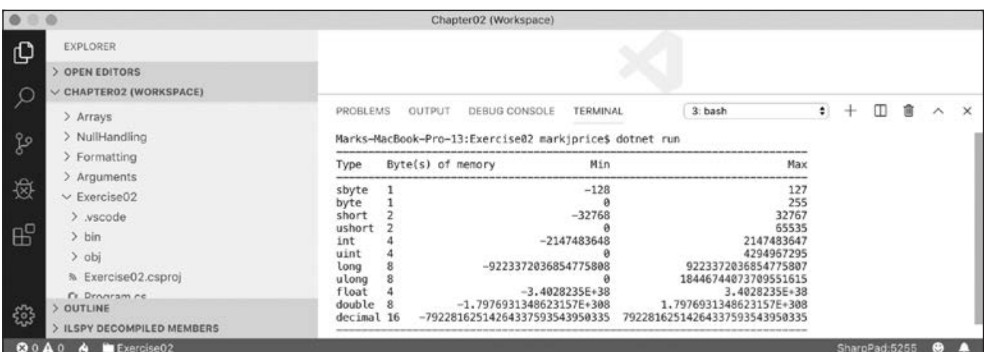


Рис. 2.10. Результат работы консольного приложения

Упражнение 2.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- ключевые слова C#: https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/index;
- `Main()` и аргументы командной строки (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/main-and-command-args/;
- типы (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/;
- операторы и выражения (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/statements-expressions-operators/;
- строки (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/strings/;
- типы значений, допускающие значения `null` (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/nullable-types/;
- Ссылочные типы, допускающие значения `null`: https://docs.microsoft.com/ru-ru/dotnet/csharp/nullable-references;
- класс `Console`: https://docs.microsoft.com/ru-ru/dotnet/api/system.console.

Резюме

Вы научились объявлять переменные с использованием явно указанного или автоматически определенного типа; мы обсудили встроенные числовые, текстовые и логические типы; я рассказал о разнице между различными числовыми типами; а также мы рассмотрели типы, допускающие значение `null`, и узнали, как управлять форматированием вывода в консольных приложениях.

Далее вы узнаете о принципах ветвления, перебора, преобразования типов.

3 Управление потоком исполнения и преобразование типов

Данная глава посвящена написанию кода, который принимает решения, повторяет блоки операторов, преобразует значения переменных или выражений из одного типа в другой, обрабатывает исключения и проверяет переполнение числовых переменных.

В этой главе:

- работа с переменными;
- операторы выбора;
- операторы цикла;
- приведение и преобразование типов;
- обработка исключений;
- проверка переполнения.

Работа с переменными

Операции[1] *(operators)* применяют простые действия, такие как сложение и умножение, к *операндам*, например переменным и литеральным значениям. Обычно они в результате возвращают новое значение.

Большинство операций — бинарные, то есть работают с двумя операндами, как показано в следующем псевдокоде:

```
var результатОперации = ПервыйОперанд операция ВторойОперанд;
```

[1] Английское слово operator, соответствующее термину «операция», иногда ошибочно переводят как «оператор». На самом деле (по историческим причинам) русский термин «оператор» соответствует английскому statement. Разговаривая с коллегами, скорее всего, вы будете использовать термин «оператор» как аналог англоязычного operator.

Некоторые операции — унарные, то есть работают с одним операндом и могут применяться до или после него, как показано в следующем псевдокоде:

```
var результатОперации = ТолькоОдинОперанд операция;
```

или

```
var результатОперации = операция ТолькоОдинОперанд;
```

Примеры унарных операций включают инкременты и извлечение типа или его размера в байтах:

```
int x = 5;
int incrementedByOne = x++;
int incrementedByOneAgain = ++x;
Type theTypeOfAnInteger = typeof(int);
int howManyBytesInAnInteger = sizeof(int);
```

Тернарная операция работает с тремя операндами, как показано в следующем псевдокоде:

```
var результатОперации = ПервыйОперанд перваяОперация
ВторойОперанд втораяОперация ТретийОперанд;
```

Унарные операции

К двум самым распространенным унарным операциям относятся операции инкремента (увеличения), `++`, и декремента (уменьшения), `--`, числа.

1. Если вы завершили предыдущие главы, то в вашей пользовательской папке уже есть папка Code. Если нет, то создайте ее.
2. В папке Code создайте подпапку Chapter03.
3. Запустите Visual Studio Code и закройте любую открытую рабочую область или папку.
4. Сохраните текущую рабочую область в папке Chapter03 под именем `Chapter03.code-workspace`.
5. Создайте папку Operators и добавьте ее в рабочую область `Chapter03`.
6. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал).
7. На панели TERMINAL (Терминал) введите команду для создания консольного приложения в папке Operators.
8. Откройте проект `Program.cs`.
9. Статически импортируйте тип `System.Console`.

10. В методе `Main` объявите две целочисленные переменные с именами `a` и `b`, задайте `a` равным трем, увеличьте `a`, присваивая результат `b`, а затем выведите значения:

```
int a = 3;
int b = a++;
WriteLine($"a is {a}, b is {b}");
```

11. Перед запуском консольного приложения задайте себе вопрос: как вы думаете, каким будет значение переменной `b` при выводе? Далее запустите консольное приложение и проанализируйте результат:

```
a is 4, b is 3
```

Переменная `b` имеет значение `3`, поскольку операция `++` выполняется *после* присваивания; это известно как *постфиксная операция*. Если вам нужно увеличить значение перед присваиванием, то используйте *префиксную операцию*.

12. Скопируйте и вставьте операции, а затем измените их, переименовав переменные и использовав префиксную операцию:

```
int c = 3;
int d = ++c; // увеличение c перед присваиванием
WriteLine($"c is {c}, d is {d}");
```

13. Перезапустите консольное приложение и проанализируйте результат, как показано в следующем выводе:

```
a is 4, b is 3
c is 4, d is 4
```

Из-за путаницы с префиксами и постфиксами операций инкремента и декремента при присваивании разработчики языка программирования Swift планируют отказаться от поддержки этой операции в версии 3. Я рекомендую программистам на языке C# никогда не сочетать операции ++ и –– с операцией присваивания =. Лучше выполнять эти действия как отдельные операторы.

Арифметические бинарные операции

Инкремент и декремент — унарные арифметические операции. Другие арифметические операции обычно бинарные и позволяют выполнять арифметические действия с двумя числами.

1. Добавьте операторы в конце метода `Main`, которые объявляют и присваивают значения двум целочисленным переменным с именами `e` и `f`, а затем выполните

пять обычных бинарных арифметических операций для двух чисел, как показано ниже:

```
int e = 11;
int f = 3;
WriteLine($"e is {e}, f is {f}");
WriteLine($"e + f = {e + f}");
WriteLine($"e - f = {e - f}");
WriteLine($"e * f = {e * f}");
WriteLine($"e / f = {e / f}");
WriteLine($"e % f = {e % f}");
```

2. Перезапустите консольное приложение и проанализируйте результат, как показано в следующем выводе:

```
e is 11, f is 3
e + f = 14
e - f = 8
e * f = 33
e / f = 3
e % f = 2
```

Чтобы понять, как работают операции деления (`/`) и остатка от деления (деления по модулю) (`%`) при применении к целым числам (натуральным числам), вам нужно вспомнить уроки средней школы. Представьте, что у вас есть одиннадцать леденцов и три друга. Как разделить леденцы поровну между друзьями? Вы можете дать по три леденца каждому другу, после чего останется два леденца. Они представляют собой *остаток от деления*, также известный как *модуль*. Если же у вас двенадцать леденцов, то каждый друг получает четыре леденца и ничего не остается. Таким образом, остаток равен 0.

3. Добавьте операторы для объявления переменной и присвойте значение переменной типа `double` с именем `g`, чтобы показать разницу между делением на целые и действительные числа:

```
double g = 11.0;
WriteLine($"g is {g:N1}, f is {f}");
WriteLine($"g / f = {g / f}");
```

4. Перезапустите консольное приложение и проанализируйте результат:

```
g is 11.0, f is 3
g / f = 3.6666666666666665
```

Если брать вещественные числа в качестве первого операнда, например значение переменной `g`, равное `11,0`, то операция деления возвращает значение с плавающей запятой, например `3,6666666666665`, а не целое число.

Операция присваивания

Вы уже использовали самую распространенную операцию присваивания, =.

Чтобы сократить ваш код, вы можете объединить операцию присваивания с другими операциями, такими как арифметические операции:

```
int p = 6;
p += 3; // эквивалентно p = p + 3;
p -= 3; // эквивалентно p = p - 3;
p *= 3; // эквивалентно p = p * 3;
p /= 3; // эквивалентно p = p / 3;
```

Логические операции

Логические операции работают с логическими значениями и возвращают в результате значение `true` (истина) или `false` (ложь).

Рассмотрим бинарные логические операции, работающие с двумя логическими значениями.

1. Создайте папку и консольное приложение `BooleanOperators` и добавьте его в рабочую область `Chapter03`. Не забудьте использовать палитру команд, чтобы выбрать `BooleanOperators` в качестве проекта.

Не забудьте статически импортировать тип System.Console, чтобы упростить операторы в консольном приложении.

2. В файле `Program.cs` в методе `Main` добавьте операторы для объявления двух логических переменных со значениями `true` и `false`, а затем выведите таблицы истинности, отображающие результаты применения логических операций AND, OR и XOR (исключающее ИЛИ):

```
bool a = true;
bool b = false;

WriteLine($"AND  | a     | b    ");
WriteLine($"a    | {a & a,-5} | {a & b,-5} ");
WriteLine($"b    | {b & a,-5} | {b & b,-5} ");
WriteLine();
WriteLine($"OR   | a     | b    ");
WriteLine($"a    | {a | a,-5} | {a | b,-5} ");
WriteLine($"b    | {b | a,-5} | {b | b,-5} ");
WriteLine();
```

```
WriteLine($"XOR   | a      | b    ");
WriteLine($"a     | {a ^ a,-5} | {a ^ b,-5} ");
WriteLine($"b     | {b ^ a,-5} | {b ^ b,-5} ");
```

3. Запустите консольное приложение и проанализируйте результаты, как показано в следующем выводе:

```
AND   | a      | b
a     | True   | False
b     | False  | False

OR    | a      | b
a     | True   | True
b     | True   | False

XOR   | a      | b
A     | False  | True
b     | True   | False
```

Для логической операции `AND &` (логическое И) оба операнда должны быть равными `true`, чтобы в результате вернулось значение `true`. Для логической операции `OR |` (логическое ИЛИ), любой операнд может быть равным `true`, чтобы в результате вернулось значение `true`. Для логической операции `XOR ^` (исключающее ИЛИ) один из операндов может быть равным `true` (но не оба!), чтобы результат был равным `true`.

Более подробно о таблицах истинности можно прочитать на сайте en.wikipedia.org/wiki/Truth_table.

Условные логические операции

Условные логические операции аналогичны логическим операциям, но вместо одного символа вы используете два, например `&&` вместо `&` или `||` вместо `|`.

В главе 4 вы узнаете о функциях более подробно, а сейчас мне нужно представить функции для объяснения условных операций, также известных как короткозамкнутый аналог логических операций.

Функция выполняет определенный список операторов, а затем возвращает значение. Оно может быть логическим значением (например, `true`), которое используется в логической операции. Рассмотрим пример использования логических операторов.

1. После и вне метода `Main` напишите операторы для объявления функции, которая выводит сообщение в консоль и возвращает значение `true`:

```
class Program
{
  static void Main(string[] args)
```

```
    {
      ...
    }

    private static bool DoStuff()
    {
      WriteLine("I am doing some stuff.");
      return true;
    }
  }
```

2. Внутри и в конце метода `Main` выполните операцию `AND` `&` с переменными `a` и `b` и проанализируйте результат вызова функции:

```
WriteLine($"a & DoStuff() = {a & DoStuff()}");
WriteLine($"b & DoStuff() = {b & DoStuff()}");
```

3. Запустите консольное приложение, проанализируйте результат и обратите внимание, что функция была вызвана дважды — для `a` и для `b`, как показано в следующем выводе:

```
I am doing some stuff.
a & DoStuff() = True
I am doing some stuff.
b & DoStuff() = False
```

4. Замените операторы `&` операторами `&&`:

```
WriteLine($"a && DoStuff() = {a && DoStuff()}");
WriteLine($"b && DoStuff() = {b && DoStuff()}");
```

5. Запустите консольное приложение, проанализируйте результат и обратите внимание, что функция работает, когда объединяется с переменной `a`, но не запускается в момент объединения с переменной `b`, поскольку переменная `b` имеет значение `false`, поэтому в любом случае результат будет ложным, ввиду чего функция не выполняется:

```
I am doing some stuff.
a && DoStuff() = True
b && DoStuff() = False // Функция DoStuff не выполнена!
```

Теперь вы можете понять, почему условные логические операции описаны как короткозамкнутые. Они могут сделать ваши приложения более эффективными, но могут и вносить незначительные ошибки в тех случаях, когда вы предполагаете, что функция будет вызываться всегда. Лучше всего избегать их при использовании в сочетании с функциями, вызывающими побочные эффекты.

Информацию, касающуюся побочных эффектов, вы можете найти на сайте https://en.wikipedia.org/wiki/Side_effect_(computer_science).

Побитовые операции и операции побитового сдвига

Побитовые операции влияют на биты в числе. Операции побитового сдвига могут выполнять некоторые распространенные арифметические вычисления намного быстрее, чем классические операции.

Рассмотрим побитовые операции и операции побитового сдвига.

1. Создайте папку и консольное приложение `BitwiseAndShiftOperators` и добавьте ее в рабочую область.
2. Добавьте операторы в метод `Main`, объявляющие две целочисленные переменные со значениями `10` и `6`, а затем выведите результаты применения побитовых операций AND, OR и XOR (исключающее ИЛИ):

```
int a = 10; // 0000 1010
int b = 6; // 0000 0110

WriteLine($"a = {a}");
WriteLine($"b = {b}");
WriteLine($"a & b = {a & b}"); // только столбец 2-го бита
WriteLine($"a | b = {a | b}"); // столбцы 8, 4 и 2-го битов
WriteLine($"a ^ b = {a ^ b}"); // столбцы 8-го и 4-го битов
```

3. Запустите консольное приложение и проанализируйте результаты, как показано в следующем выводе:

```
a = 10
b = 6
a & b = 2
a | b = 14
a ^ b = 12
```

4. Добавьте операторы в метод `Main`, чтобы вывести результаты применения операции сдвига влево для перемещения битов переменной `a` на три столбца, умножения `a` на `8` и сдвига вправо битов переменной `b` на один столбец, как показано ниже:

```
// 0101 0000 — сдвиг влево a на три битовых столбца
WriteLine($"a << 3 = {a << 3}");

// умножение на 8
WriteLine($"a * 8 = {a * 8}");

// 0000 0011 сдвиг вправо b на один битовый столбец
WriteLine($"b >> 1 = {b >> 1}");
```

5. Запустите консольное приложение и запишите результаты, как показано в следующем выводе:

```
a << 3 = 80
a * 8 = 80
b >> 1 = 3
```

Результат `80` получился из-за того, что биты в нем были смещены на три столбца влево, поэтому биты-единицы переместились в столбцы 64-го и 16-го битов, а 64 + 16 = 80. Это эквивалентно умножению на `8`, однако процессоры могут выполнить битовый сдвиг быстрее. Результат `3` получился из-за того, что биты-единицы в `b` были сдвинуты на один столбец, и оказались во 2-м и 1-м столбцах.

Прочие операции

Операции `nameof` и `sizeof` весьма удобны при работе с типами.

- Операция `nameof` возвращает короткое имя (без пространства имен) переменной, типа или члена в виде строкового значения, что полезно при выводе сообщений об исключениях.
- Операция `sizeof` возвращает размер в байтах простых типов, что полезно для определения эффективности хранения данных.

Существует большое разнообразие других операций. Например, точка между переменной и ее членами называется *операцией доступа к элементу*, а круглые скобки в конце имени функции или метода называются *операцией вызова*:

```
int age = 47;

// Сколько операций в следующем операторе?
string firstDigit = age.ToString()[0];

// Здесь четыре операции:
// = — операция присваивания
// . — операция доступа
// () — операция вызова
// [] — операция доступа к индексатору
```

Более подробно о некоторых операциях доступа к членам можно узнать на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/member-access-operators.

Операторы выбора

Код каждого приложения должен обеспечивать возможность выбора одного из нескольких вариантов и ветвиться в соответствии с ним. Два оператора выбора в языке C# носят имена `if` и `switch`. Первый применим для любых сценариев, но второй позволяет упростить код в некоторых распространенных случаях.

Например, когда есть одна переменная, которая может иметь несколько значений, каждое из которых требует особой обработки.

Ветвление с помощью оператора if

Оператор `if` определяет, по какой ветви кода необходимо следовать после вычисления логического выражения. Если выражение имеет значение `true` (истинно), то блок выполняется. Блок `else` — необязателен и выполняется, если выражение в `if` принимает значение `false` (ложь). Оператор `if` может быть вложенным.

Оператор `if` комбинируется с другими операторами `if`, как в случае с `else if`, как показано ниже:

```
if (выражение1)
{
  // работает, если выражение1 возвращает значение true
}
else if (выражение2)
{
  // работает, если выражение1 возвращает значение false,
  //а выражение2 — true
}
else if (выражение3)
{
  // работает, если выражение1 и выражение2 возвращают
  // значение false, а выражение3 — true
}
else
{
  // работает, если все выражения возвращают значение false
}
```

Каждое логическое значение выражение оператора `if` может быть независимым от других и, в отличие от операторов `switch`, не обязательно должно ссылаться на одно значение.

Создадим консольное приложение для изучения операторов выбора вроде `if`.

1. Создайте папку и консольное приложение `SelectionStatements` и добавьте его в рабочую область.
2. Добавьте следующие операторы в метод `Main` для проверки, имеет ли это консольное приложение какие-либо аргументы, переданные ему:

   ```
   if (args.Length == 0)
   {
     WriteLine("There are no arguments.");
   }
   else
   ```

```
{
  WriteLine("There is at least one argument.");
}
```

3. Запустите консольное приложение, введя следующую команду на панели TERMINAL (Терминал):

```
dotnet run
```

Почему в операторах if необходимы фигурные скобки

Поскольку в каждом блоке указывается только один оператор, этот код *можно* переписать без фигурных скобок:

```
if (args.Length == 0)
  WriteLine("There are no arguments.");
else
  WriteLine("There is at least one argument.");
```

Такой формат оператора `if` не рекомендуется, поскольку может содержать серьезные ошибки, например печально известный баг `#gotofail` в операционной системе iOS на смартфонах Apple iPhone. На протяжении 18 месяцев после релиза версии iOS 6 в сентябре 2012 года в ней присутствовала ошибка в коде протокола *Secure Sockets Layer (SSL)*. Из-за этого любой пользователь, который подключался через браузер Safari к защищенным сайтам, например к сервису интернет-банкинга, не был защищен должным образом, поскольку важная проверка была случайно пропущена.

Вы можете прочитать об этой ошибке на сайте gotofail.com.

Только из возможности опустить фигурные скобки не следует так делать. Ваш код не становится «более эффективным» без них, вместо этого он менее удобен в сопровождении и потенциально более уязвим.

Сопоставление шаблонов с помощью операторов if

Новая возможность версии C# 7.0 — сопоставление шаблонов. В операторе `if` можно использовать ключевое слово `is` в сочетании с объявлением локальной переменной, чтобы сделать ваш код более безопасным.

1. Добавьте в конец метода `Main` приведенные ниже операторы. Если значение, хранящееся в переменной `o`, представляет собой тип `int`, то значение присваивается локальной переменной `i`, которая затем может применяться в операторе `if`.

Это безопаснее, чем использование переменной `o`, поскольку мы точно знаем, что `i` — это `int`, а не что-то другое:

```
// добавьте или удалите "" чтобы изменить поведение
object o = "3";
int j = 4;

if(o is int i)
{
  WriteLine($"{i} x {j} = {i * j}");
}
else
{
  WriteLine("o is not an int so it cannot multiply!");
}
```

2. Запустите консольное приложение и проанализируйте результат:

```
o is not an int so it cannot multiply!
```

3. Удалите двойные кавычки вокруг значения `"3"`, чтобы значение, хранящееся в переменной `o`, стало `int` вместо `string`.
4. Перезапустите консольное приложение и проанализируйте результат:

```
3 x 4 = 12
```

Ветвление с помощью оператора switch

Оператор `switch` отличается от `if` тем, что проверяет одно выражение на соответствие трем или больше условиям (`case`). Каждое условие относится к единственному выражению. Каждый раздел `case` должен заканчиваться:

- ключевыми словами `break` (например, в коде ниже `case 1`);
- или `goto case` (например, в коде ниже `case 2`);
- либо не иметь никаких операторов (например, в коде ниже `case 3`);
- или ключевым словом `return` для выхода из текущей функции (не показано в коде).

Напишем пример кода для изучения операторов `switch`.

1. Введите код для оператора `switch`, указанный ниже, после операторов `if`, написанных вами ранее. Обратите внимание: первая строка — метка, к которой можно перейти, а вторая строка генерирует случайное число. Оператор `switch` выбирает ветвь, основываясь на значении этого числа:

```
A_label:
  var number = (new Random()).Next(1, 7);
  WriteLine($"My random number is {number}");

  switch (number)
```

```
{
  case 1:
    WriteLine("One");
    break; // переход в конец оператора switch
  case 2:
    WriteLine("Two");
    goto case 1;
  case 3:
  case 4:
    WriteLine("Three or four");
    goto case 1;
  case 5:
    // заснуть на полсекунды
    System.Threading.Thread.Sleep(500);
    goto A_label;
  default:
    WriteLine("Default");
    break;
} // конец оператора switch
```

Вы можете добавить ключевое слово goto для перехода к другому условию или метке. Применение этого слова не одобряется большинством программистов, но может стать хорошим решением при кодировании логики в некоторых сценариях. Не стоит использовать его слишком часто.

О ключевом слове goto и примерах его использования можно прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/goto.

2. Запустите приложение несколько раз, чтобы посмотреть на происходящее при различных условиях случайных чисел, как показано в следующем примере:

```
bash-3.2$ dotnet run
My random number is 4
Three or four
One
bash-3.2$ dotnet run
My random number is 2
Two
One
bash-3.2$ dotnet run
My random number is 1
One
```

Сопоставление шаблонов с помощью оператора switch

Как и `if`, оператор `switch` поддерживает сопоставление шаблонов в версии C# 7.0 и более поздних. Значения условий `case` больше не должны быть литеральными. Они могут быть шаблонами.

Рассмотрим пример сопоставления шаблона с оператором `switch` с помощью пути к папке. Если вы работаете в операционной системе macOS, то поменяйте местами закомментированный оператор, который устанавливает переменную пути, и замените мое имя пользователя именем вашей пользовательской папки.

1. Добавьте следующий оператор в начало файла, чтобы импортировать типы для работы с вводом/выводом:

   ```
   using System.IO;
   ```

2. Добавьте операторы в конец метода `Main`, чтобы объявить строковый путь к файлу, открыть его как поток и затем показать сообщение в зависимости от типа и возможностей потока:

   ```
   // string path = "/Users/markjprice/Code/Chapter03";
   string path = @"C:\Code\Chapter03";

   Write("Press R for readonly or W for write: ");
   ConsoleKeyInfo key = ReadKey();
   WriteLine();

   Stream s = null;

   if (key.Key == ConsoleKey.R)
   {
     s = File.Open(
       Path.Combine(path, "file.txt"),
       FileMode.OpenOrCreate,
       FileAccess.Read);
   }
   else
   {
     s = File.Open(
       Path.Combine(path, "file.txt"),
       FileMode.OpenOrCreate,
       FileAccess.Write);
   }

   string message = string.Empty;

   switch (s)
   {
     case FileStream writeableFile when s.CanWrite:
       message = "The stream is a file that I can write to.";
       break;
     case FileStream readOnlyFile:
       message = "The stream is a read-only file.";
       break;
     case MemoryStream ms:
       message = "The stream is a memory address.";
       break;
   ```

```
    default: // всегда выполняется последним, несмотря на текущее положение
      message = "The stream is some other type.";
      break;
    case null:
      message = "The stream is null.";
      break;
  }
  WriteLine(message);
```

3. Запустите консольное приложение и обратите внимание, что переменная `s` объявлена как тип `Stream`, вследствие чего это может быть любой подтип потока, например поток памяти или файлов. В данном коде поток создается с помощью метода `File.Open`, который возвращает файловый поток, и благодаря `FileMode` он будет доступен для записи, поэтому результатом будет сообщение, описывающее ситуацию:

```
The stream is a file that I can write to.
```

На платформе .NET доступно несколько подтипов `Stream`, включая `FileStream` и `MemoryStream`. В версии C# 7.0 ваш код может быть более лаконичным благодаря ветвлению на основе подтипа потока и объявлению/назначению локальной переменной для безопасного использования. Более подробно пространство имен `System.IO` и тип `Stream` мы разберем в главе 9.

Кроме того, операторы `case` могут содержать ключевое слово `when` для выполнения более специфичного сопоставления шаблонов. В первом операторе `case` в предыдущем коде совпадение было бы установлено только в том случае, если поток — это `FileStream`, а его свойство `CanWrite` истинно.

Более подробную информацию о сопоставлении шаблонов можно получить на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/pattern-matching.

Упрощение операторов switch с помощью выражений switch

В версии C# 8.0 или более поздних версиях вы можете упростить операторы `switch`, используя *выражения switch*.

Большинство операторов `switch` очень просты, но требуют писать много кода. Выражения `switch` предназначены для упрощения кода, который нужно набирать, при этом выражая то же самое намерение в сценариях, где все варианты возвращают значение для установки одной переменной. В выражениях `switch` для обозначения возвращаемого значения используется лямбда `=>`.

Для сравнения выполним предыдущий код, использующий оператор `switch`, с помощью выражения `switch`.

1. Добавьте операторы в конец метода `Main`, чтобы установить сообщение в зависимости от типа и возможностей потока, использующего выражение `switch`, как показано ниже:

```
message = s switch
{
  FileStream writeableFile when s.CanWrite
    => "The stream is a file that I can write to.",
  FileStream readOnlyFile
    => "The stream is a read-only file.",
  MemoryStream ms
    => "The stream is a memory address.",
  null
    => "The stream is null.",
  _
    => "The stream is some other type."
};

WriteLine(message);
```

   Основное отличиое — удаление ключевых слов `case` и `break`. Символ подчеркивания используется для представления возвращаемого значения по умолчанию.

2. Запустите консольное приложение и обратите внимание, что результат такой же, как и раньше.

Более подробную информацию о шаблонах и выражениях switch можно узнать на сайте https://devblogs.microsoft.com/dotnet/do-more-with-patterns-in-c-8-0/.

Операторы цикла

Операторы цикла повторяют блок операторов, пока условие истинно. Выбор того, какой оператор следует использовать, основывается на сочетании простоты понимания решения логической задачи и личных предпочтений.

Оператор while

Оператор `while` оценивает логическое выражение и продолжает цикл, пока оно остается истинным. Рассмотрим операторы цикла на примере.

1. Создайте папку для консольного приложения `IterationStatements` и добавьте ее в рабочую область.

2. Введите следующий код внутри метода `Main`:

```
int x = 0;

while (x < 10)
{
  WriteLine(x);
  x++;
}
```

3. Запустите консольное приложение и проанализируйте результат (должны быть числа от 0 до 9):

```
0
1
2
3
4
5
6
7
8
9
```

Оператор do

Оператор `do` похож на `while`, за исключением того, что логическое выражение проверяется в конце блока кода, а не в начале, что означает обязательное выполнение кода хотя бы один раз.

1. Введите код, показанный ниже, в конце метода `Main`:

```
string password = string.Empty;

do
{
  Write("Enter your password: ");
  password = ReadLine();
}
while (password != " Pa$$w0rd");

WriteLine("Correct!");
```

2. Запустите консольное приложение и обратите внимание, что вам будет предложено ввести пароль несколько раз, пока вы не введете его правильно, как показано в следующем выводе:

```
Enter your password: password
Enter your password: 12345678
Enter your password: ninja
```

```
Enter your password: correct horse battery staple
Enter your password: Pa$$w0rd
Correct!
```

3. В качестве дополнительной практики добавьте операторы, ограничивающие количество попыток ввода пароля десятью, после чего, если правильный пароль так и не был введен, выводится сообщение об ошибке.

Оператор for

Оператор `for` аналогичен `while`, за исключением более лаконичного синтаксиса. Он комбинирует:

- *выражение инициализатора*, которое выполняется однократно при запуске цикла;
- *условное выражение*, выполняющееся на каждой итерации в начале цикла, чтобы проверить, следует ли продолжать цикл;
- *выражение-итератор*, которое выполняется в каждом цикле в конце выполнения оператора.

Оператор `for` обычно используется с целочисленным счетчиком. Рассмотрим следующий пример.

1. Введите оператор `for` для вывода чисел от 1 до 10:

```
for (int y = 1; y <= 10; y++)
{
  WriteLine(y);
}
```

2. Запустите консольное приложение и проанализируйте результат, который должен быть представлен числами в диапазоне от 1 до 10.

Оператор foreach

Оператор `foreach` несколько отличается от предыдущих трех операторов цикла.

Он используется в целях выполнения блока операторов для каждого элемента в последовательности, к примеру в массиве или коллекции. Каждый элемент доступен только для чтения, и если во время итерации последовательность изменяется, к примеру, путем добавления или удаления элемента, то будет вызвано исключение. Рассмотрим следующий пример.

1. Введите операторы, чтобы создать массив строковых переменных, а затем выведите длину каждой из них:

```
string[] names = { "Adam", "Barry", "Charlie" };

foreach (string name in names)
{
  WriteLine($"{name} has {name.Length} characters.");
}
```

2. Запустите консольное приложение и проанализируйте результат:

```
Adam has 4 characters.
Barry has 5 characters.
Charlie has 7 characters.
```

Принципы работы оператора foreach

Технически оператор `foreach` будет работать с любым типом, соблюдающим правила, изложенные ниже.

1. Тип должен иметь метод `GetEnumerator`, который возвращает объект.
2. Возвращаемый объект должен иметь свойство `Current` и метод `MoveNext`.
3. Метод `MoveNext` должен возвращать значение `true`, если есть другие элементы для перечисления, или значение `false`, если элементов больше нет.

Существуют интерфейсы с именами `IEnumerable` и `IEnumerable <T>`, которые формально определяют эти правила, но технически компилятору не требует от типа реализации этих интерфейсов.

Компилятор превращает код оператора `foreach` из примера в предыдущем подразделе в нечто похожее на следующий псевдокод:

```
IEnumerator e = names.GetEnumerator();

while (e.MoveNext())
{
  string name = (string)e.Current; // Current — только для чтения!
  WriteLine($"{name} has {name.Length} characters.");
}
```

Из-за применения итератора переменная, объявленная в операторе `foreach`, не может использоваться для изменения значения текущего элемента.

Приведение и преобразование типов

Вам часто понадобится выполнять преобразование различных типов. Например, данные часто вводятся в виде текста в консоли, поэтому изначально сохраняются в строковой переменной, а затем их необходимо преобразовать в дату/время,

число или какой-либо другой тип данных, в зависимости от того, как они должны храниться и обрабатываться.

Иногда вам нужно будет выполнить преобразование числовых типов, например целого числа и числа с с плавающей запятой, прежде чем выполнять вычисления.

Преобразование также называется *приведением*, и оно имеет две разновидности: *неявное* и *явное*. Неявное происходит автоматически, и это безопасно, то есть вы не потеряете никакую информацию.

Явное приведение должно быть выполнено вручную, поскольку вы можете потерять информацию, например точность числа. При явном приведении вы сообщаете компилятору C#, что понимаете риск и принимаете его на себя.

Явное и неявное приведение типов

Вы можете неявно привести переменную `int` к типу `double`, что безопасно, поскольку никакая информация не потеряется.

1. Создайте папку и проект консольного приложения `CastingConverting` и добавьте его в рабочую область.
2. В методе `Main` введите операторы для объявления и назначения переменных `int` и `double`, а затем неявно приведите значение целого числа при назначении его переменной `double`:

```
int a = 10;
double b = a; // тип int может быть сохранен как double
WriteLine(b);
```

3. В метод `Main` добавьте следующие операторы:

```
double c = 9.8;
int d = c; // компилятор выдаст ошибку на этой строке
WriteLine(d);
```

4. Откройте панель PROBLEMS (Проблемы), выбрав View ▸ Problems (Вид ▸ Проблемы), и обратите внимание на сообщение об ошибке (рис. 3.1).

   Если вам требуется создать ресурсы, необходимые для отображения панели PROBLEMS (Проблемы), то попробуйте закрыть и снова открыть рабочую область, выберите правильный проект для OmniSharp и нажмите кнопку Yes (Да) при появлении запроса на создание отсутствующих ресурсов, например папки .vscode. В строке состояния должен отображаться текущий активный проект, например `CastingConverting`, как на рис. 3.1 выше.

   Вы не можете неявно преобразовать переменную `double` в переменную `int`, поскольку это потенциально небезопасно и может привести к потере данных.

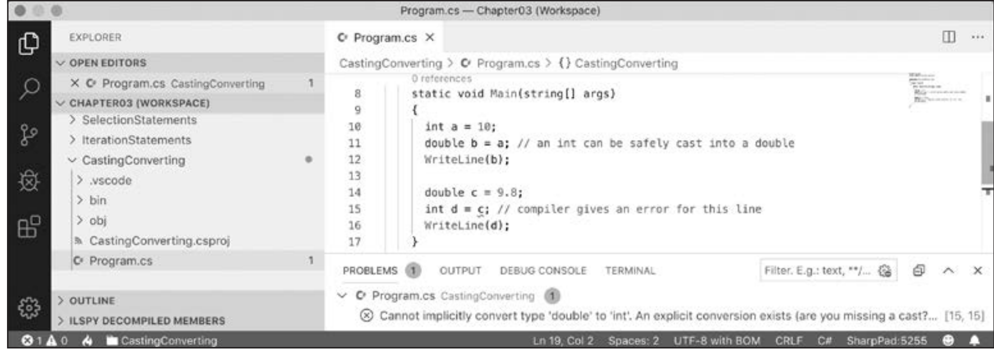


Рис. 3.1. Обратите внимание на сообщение об ошибке

5. Откройте панель TERMINAL (Терминал), введите команду запуска `dotnet` и проанализируйте сообщение об ошибке:

```
Program.cs(19,15): error CS0266: Cannot implicitly convert type 'double'
to 'int'. An explicit conversion exists (are you missing a cast?) [/Users/
markjprice/Code/Chapter03/CastingConverting/CastingConverting.csproj]
The build failed. Fix the build errors and run again.
```

   Необходимо явно привести тип `double` к типу `int`, заключив тип, к которому вы хотите привести переменную `double`, в круглые скобки. Парой круглых скобок обозначается *операция приведения типа*. И даже в этом случае вы должны остерегаться, что часть после десятичной запятой будет отброшена без предупреждения.

6. Измените оператор присваивания переменной `d`, как показано в коде, приведенном ниже:

```
int d = (int)c;
WriteLine(d); // d равняется 9 с потерей части .8
```

7. Запустите консольное приложение и проанализируйте результат:

```
10
9
```

   Следует выполнять аналогичную операцию при преобразовании значений между большими и меньшими целыми числами. Снова напомним: будьте осторожны, поскольку вы можете потерять данные, ведь слишком большое значение будет скопировано, а затем интерпретировано так, как вы этого, возможно, не ожидаете!

8. Введите операторы для объявления и присвойте 64-битную переменную типа `long` 32-битной переменной типа `int`, используя небольшое значение, которое

будет работать, и слишком большое значение, которое не будет работать, как показано ниже:

```
long e = 10;
int f = (int)e;
WriteLine($"e is {e:N0} and f is {f:N0}");

e = long.MaxValue;
f = (int)e;
WriteLine($"e is {e:N0} and f is {f:N0}");
```

9. Запустите консольное приложение и проанализируйте результат:

```
e is 10 and f is 10
e is 9,223,372,036,854,775,807 and f is -1
```

10. Измените значение переменной `e` на пять миллиардов:

```
e = 5_000_000_000;
```

11. Запустите консольное приложение, чтобы просмотреть результаты, как показано в следующем выводе:

```
e is 5,000,000,000 and f is 705,032,704
```

Использование типа System.Convert

Использование типа `System.Convert` — альтернатива применению операции приведения типа. Тип `System.Convert` может преобразовывать во все числовые типы чисел C# и из них, а также в логические значения, строки и значения даты и времени.

Рассмотрим следующий пример.

1. В начале проекта `Program.cs` статически импортируйте класс `System.Convert`, как показано ниже:

```
using static System.Convert;
```

2. Добавьте следующие операторы в конец метода `Main`, чтобы объявить и присвоить значение переменной типа `double`, затем преобразовать его в целое число, а затем записать оба значения в консоль:

```
double g = 9.8;
int h = ToInt32(g);
WriteLine($"g is {g} and h is {h}");
```

3. Запустите консольное приложение и проанализируйте результат:

```
g is 9.8 and h is 10
```

Одно из различий между приведением и преобразованием заключается в том, что при преобразовании значение `double 9,8` округляется до `10` вместо отбрасывания части после десятичной запятой.

Округление чисел

Вы убедились, что операция приведения типа отбрасывает десятичную часть вещественного числа, а при преобразовании с помощью метода `System.Convert` число округляется в большую или меньшую сторону. Но каковы правила округления?

Правила округления

В начальных школах дети учатся округлять числа следующим образом: если десятичная часть больше или равна 0,5, то число округляется в *бо́льшую* сторону, а если десятичная меньше 0,5 — то в *меньшую*.

Посмотрим, следует ли язык C# тому же правилу начальной школы.

1. В конце метода `Main` добавьте операторы для объявления массива значений типа `double` и наполнения его значениями, преобразуйте каждый элемент массива в целое число, а затем запишите результат в консоль:

```
double[] doubles = new[]
  { 9.49, 9.5, 9.51, 10.49, 10.5, 10.51 };

foreach (double n in doubles)
{
  WriteLine($"ToInt({n}) is {ToInt32(n)}");
}
```

2. Запустите консольное приложение и проанализируйте результат:

```
ToInt(9.49) is 9
ToInt(9.5) is 10
ToInt(9.51) is 10
ToInt(10.49) is 10
ToInt(10.5) is 10
ToInt(10.51) is 11
```

Обратите внимание, что правила округления в языке C# несколько иные:

- число всегда округляется в *меньшую сторону*, если десятичная часть меньше 0,5;
- число всегда округляется в *бо́льшую сторону*, если десятичная часть больше 0,5;
- число округляется в *бо́льшую сторону*, если десятичная часть равна 0,5, а целая часть — нечетная, и округляется в *меньшую сторону*, если целая часть — четная.

Эта схема известна как *«банковское округление»* и является более предпочтительной, поскольку уменьшает вероятность ошибки. К сожалению, другие языки, например JavaScript, используют правила округления из начальной школы.

Контроль правил округления

Вы можете контролировать правила округления, используя метод `Round` класса `Math`.

1. В конце метода `Main` добавьте операторы для округления каждого из значений `double`, используя правило округления «от нуля», также известное как округление в бо́льшую сторону, а затем запишите результат в консоль, как показано в следующем примере:

```
foreach (double n in doubles)
{
  WriteLine(format:
    "Math.Round({0}, 0, MidpointRounding.AwayFromZero) is {1}",
    arg0: n,
    arg1: Math.Round(value: n,
      digits: 0,
      mode: MidpointRounding.AwayFromZero));
}
```

2. Запустите консольное приложение и проанализируйте результат, как показано в следующем выводе:

```
Math.Round(9.49, 0, MidpointRounding.AwayFromZero) is 9
Math.Round(9.5, 0, MidpointRounding.AwayFromZero) is 10
Math.Round(9.51, 0, MidpointRounding.AwayFromZero) is 10
Math.Round(10.49, 0, MidpointRounding.AwayFromZero) is 10
Math.Round(10.5, 0, MidpointRounding.AwayFromZero) is 11
Math.Round(10.51, 0, MidpointRounding.AwayFromZero) is 11
```

MidpointRounding.AwayFromZero — это правило начальной школы. Более подробно о правилах округления можно прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/api/system.math.round.

Проверяйте правила округления в каждом языке программирования, с которым работаете. Они могут работать не так, как вы ожидаете!

Преобразование значения любого типа в строку

Наиболее распространено преобразование в тип `string`, поэтому все типы включают метод `ToString`, наследуемый ими от класса `System.Object`.

Метод `ToString` преобразует текущее значение любой переменной в текстовое представление. Некоторые типы не могут быть осмысленно представлены в виде текста, поэтому возвращают свое пространство имен и название типа.

Рассмотрим следующий пример.

1. В конце метода `Main` введите операторы для объявления некоторых переменных, преобразуйте их в строковое представление и запишите их в консоль, как показано ниже:

```
int number = 12;
WriteLine(number.ToString());

bool boolean = true;
WriteLine(boolean.ToString());

DateTime now = DateTime.Now;
WriteLine(now.ToString());

object me = new object();
WriteLine(me.ToString());
```

2. Запустите консольное приложение и проанализируйте результат:

```
12
True
27/01/2019 13:48:54
System.Object
```

Преобразование двоичного (бинарного) объекта в строку

Когда у вас имеется двоичный (бинарный) объект, такой как изображение или видео, который вы хотите сохранить или передать, в некоторых случаях вы не хотите отправлять необработанные биты, поскольку не знаете, как они могут быть истолкованы, например, сетевым протоколом, передающим этот объект, или другой операционной системой, которая считывает двоичный объект хранилища.

Самое безопасное — преобразовать двоичный объект в строку безопасных символов. Программисты называют это кодировкой *Base64*.

Методы `ToBase64String` и `FromBase64String` типа `Convert` выполняют такое преобразование.

Рассмотрим следующий пример:

1. Добавьте операторы в конец метода `Main`, чтобы создать массив байтов, случайно заполненный байтовыми значениями. Запишите в консоль каждый байт

в красиво отформатированном виде, а затем те же байты, преобразованные в Base64:

```
// выделение массива из 128 байт
byte[] binaryObject = new byte[128];

// заполнение массива случайными байтами
(new Random()).NextBytes(binaryObject);

WriteLine("Binary Object as bytes:");

for(int index = 0; index < binaryObject.Length; index++)
{
  Write($"{binaryObject[index]:X} ");
}
WriteLine();

// преобразование в строку Base64 и вывод в виде текста
string encoded = Convert.ToBase64String(binaryObject);
WriteLine($"Binary Object as Base64: {encoded}");
```

По умолчанию значение `int` будет выводиться в десятичной записи, то есть Base10. Вы можете использовать коды форматирования, такие как `:X`, чтобы отформатировать значение в шестнадцатеричном формате.

2. Запустите консольное приложение и проанализируйте результат:

```
Binary Object as bytes:
B3 4D 55 DE 2D E BB CF BE 4D E6 53 C3 C2 9B 67 3 45 F9 E5 20 61 7E 4F 7A 81
EC 49 F0 49 1D 8E D4 F7 DB 54 AF A0 81 5 B8 BE CE F8 36 90 7A D4 36 42 4 75
81 1B AB 51 CE 5 63 AC 22 72 DE 74 2F 57 7F CB E7 47 B7 62 C3 F4 2D 61 93
85 18 EA 6 17 12 AE 44 A8 D B8 4C 89 85 A9 3C D5 E2 46 E0 59 C9 DF 10 AF ED
EF 8AA1 B1 8D EE 4A BE 48 EC 79 A5 A 5F 2F 30 87 4A C7 7F 5D C1 D 26 EE
Binary Object as Base64: s01V3i0Ou8++TeZTw8KbZwNF+eUgYX5PeoHsSfBJHY7U99tUr6
CBBbi+zvg2kHrUNkIEdYEbq1HOBWOsInLedC9Xf8vnR7diw/QtYZOFGOoGFxKuRKgNuEyJhak81
eJG4FnJ3xCv7e+KobGN7kq+SO x5pQpfLzCHSsd/XcENJu4=
```

Разбор строк для преобразования в числа или значения даты и времени

Вторым по популярности считается преобразование из строковых переменных в числа или значения даты и времени.

Здесь вместо метода `ToString` выступает метод `Parse`. Только несколько типов поддерживают метод `Parse`, включая все числовые типы и тип `DateTime`.

Рассмотрим следующий пример.

1. Добавьте операторы в метод `Main`, чтобы извлечь целое число и значение даты и времени из строк, а затем запишите результат в консоль:

```
int age = int.Parse("27");
DateTime birthday = DateTime.Parse("4 July 1980");

WriteLine($"I was born {age} years ago.");
WriteLine($"My birthday is {birthday}.");
WriteLine($"My birthday is {birthday:D}.");
```

2. Запустите консольное приложение и проанализируйте результат:

```
I was born 27 years ago.
My birthday is 04/07/1980 00:00:00.
My birthday is 04 July 1980.
```

По умолчанию значение даты и времени выводится в коротком формате. Вы можете использовать описатели формата типа `D` (для вывода только даты в полном формате).

Существует множество других форматов даты и времени для распространенных сценариев, о которых можно прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/base-types/standard-date-and-time-format-strings.

Существует одна проблема с методом `Parse`, которая заключается в том, что он выдает ошибку, если строка не может быть преобразована.

3. Добавьте оператор в конец метода `Main`, чтобы попытаться превратить строку, содержащую буквы, в целочисленную переменную:

```
int count = int.Parse("abc");
```

4. Запустите консольное приложение и проанализируйте результат:

```
Unhandled Exception: System.FormatException: Input string was not in a
correct format.
```

Как и в предыдущем сообщении об ошибке, вы увидите трассировку стека. Я не включил тему трассировок в эту книгу, поскольку они занимают слишком много места.

Как избежать исключений с помощью метода TryParse

Во избежание ошибки вы можете использовать метод `TryParse` вместо `Parse`. `TryParse` пробует преобразовать исходную строку и возвращает `true`, если преобразование возможно, и `false` в противном случае. Ключевое слово `out` требуется для того, чтобы разрешить методу `TryParse` устанавливать переменную `count` при проведении преобразования.

Рассмотрим следующий пример.

1. Замените объявление `int count` операторами, чтобы применить метод `TryParse`, и попросите пользователя ввести число для количества яиц:

```
Write("How many eggs are there? ");
int count;
string input = ReadLine();

if (int.TryParse(input, out count))
{
  WriteLine($"There are {count} eggs.");
}
else
{
  WriteLine("I could not parse the input.");
}
```

2. Запустите консольное приложение.
3. Введите число `12` и проанализируйте результат:

```
How many eggs are there? 12
There are 12 eggs.
```

4. Снова запустите приложение.
5. Введите `twelve` — и увидите результат, показанный ниже:

```
How many eggs are there? twelve
I could not parse the input.
```

Помимо этого, вы можете использовать методы типа `System.Convert` для преобразования строковых значений в другие типы. Однако, как и метод `Parse`, он выдает ошибку, если преобразование невозможно.

Обработка исключений при преобразовании типов

Вы ознакомились с несколькими сценариями, при выполнении которых возникали ошибки. В таких ситуациях в C# вызываются исключения.

Как вы видели, в случае ошибки консольное приложение по умолчанию выводит информацию об исключении и прекращает работу.

Старайтесь не писать код, который будет выдавать исключение, когда это возможно; например, путем выполнения проверок операторов. Но иногда это невозможно. В этих сценариях вы можете перехватить исключение и обработать его лучше, чем это сделало бы поведение по умолчанию.

Оборачивание потенциально ошибочного кода в оператор try

Предполагая, что оператор может вызвать ошибку, вы должны заключить этот оператор в блок `try`. Например, анализ строки для преобразования в число может

вызвать ошибку. Любые операторы в блоке `catch` будут выполняться только в том случае, если в блоке `try` выбрасывается исключение. В блоке `catch` можно не производить никаких операций.

Рассмотрим следующий пример.

1. Создайте папку с консольным приложением `HandlingExceptions` и добавьте ее в рабочую область.
2. В метод `Main` добавьте операторы, чтобы предложить пользователю ввести свой возраст, а затем записать значение возраста в консоль:

   ```
   WriteLine("Before parsing");
   Write("What is your age? ");
   string input = ReadLine();
   try
   {
     int age = int.Parse(input);
     WriteLine($"You are {age} years old.");
   }
   catch
   {
   }
   WriteLine("After parsing");
   ```

   Данный код включает в себя два сообщения, которые нужно указывать перед синтаксическим анализом и после него, чтобы сделать процесс прохождения кода более понятным. Это будет особенно полезно, поскольку пример кода становится все более сложным.
3. Запустите консольное приложение.
4. Запустите консольное приложение и введите допустимое значение возраста, к примеру `47`, и проанализируйте результат:

   ```
   Before parsing
   What is your age? 47
   You are 47 years old.
   After parsing
   ```

5. Запустите консольное приложение еще раз.
6. Введите некорректное значение возраста, к примеру `kermit`, и проанализируйте результат:

   ```
   Before parsing
   What is your age? kermit
   After parsing
   ```

Исключение было перехвачено, но хорошо бы увидеть тип ошибки, которая произошла.

Перехват всех исключений

Чтобы получить информацию о возможном исключении любого типа, вы можете объявить переменную типа `System.Exception` в блоке `catch`.

1. Добавьте объявление переменной исключения в блок `catch` и используйте его для записи информации об исключении в консоль:

   ```
   catch(Exception ex)
   {
     WriteLine($"{ex.GetType()} says {ex.Message}");
   }
   ```

2. Запустите консольное приложение.
3. Введите некорректное значение возраста, к примеру `kermit`, и проанализируйте результат:

   ```
   Before parsing
   What is your age? kermit
   System.FormatException says Input string was not in a correct format.
   After parsing
   ```

Перехват определенных исключений

Теперь, зная тип возникшей ошибки, вы можете улучшить код, перехватывая ошибку только данного типа и изменив сообщение, которое выводится пользователю.

1. Не трогая существующий блок `catch`, выше него добавьте следующий код:

   ```
   catch (FormatException)
   {
     WriteLine("The age you entered is not a valid number format.");
   }
   catch (Exception ex)
   {
     WriteLine($"{ex.GetType()} says {ex.Message}");
   }
   ```

2. Запустите консольное приложение.
3. Введите некорректное значение возраста, к примеру `kermit`, и проанализируйте результат:

   ```
   Before parsing
   What is your age? kermit
   The age you entered is not a valid number format.
   After parsing
   ```

   Причина, по которой мы пишем универсальный блок `catch` в самом низу, состоит в том, что могут возникнуть исключения другого типа.

4. Запустите консольное приложение.

5. Введите очень большое целое число, к примеру `9876543210`, и проанализируйте результат:

```
Before parsing
What is your age? 9876543210
System.OverflowException says Value was either too large or too small
for an Int32.
After parsing
```

Добавим перехватчик для этого нового типа исключения.

6. Не трогая существующий блок `catch`, добавьте новый блок `catch`, как показано ниже (выделено полужирным шрифтом):

```
catch(OverflowException)
{
  WriteLine("Your age is a valid number format but it is either too big
or small.");
}
catch (FormatException)
{
  WriteLine("The age you entered is not a valid number format.");
}
```

7. Перезапустите приложение.
8. Введите очень большое целое число и проанализируйте результат:

```
Before parsing
What is your age? 9876543210
Your age is a valid number format but it is either too big or small.
After parsing
```

Порядок, в котором вы перехватываете исключения, важен. Правильный порядок связан с иерархией наследования исключений. С наследованием вы познакомитесь в главе 5, а пока не беспокойтесь об этом — компилятор отобразит ошибку сборки, если вы будете обрабатывать исключения в неправильном порядке.

Проверка переполнения

Ранее мы убедились, что при приведении между числовыми типами можно потерять данные, к примеру при приведении из переменной `long` к типу `int`. Если значение, хранящееся в типе, слишком велико, то возникнет переполнение.

Выброс исключений переполнения с помощью оператора checked

При переполнении оператор, обозначенный ключевым словом `checked`, передает .NET команду вызова исключения.

Мы установим начальное значение переменной `int` на максимальное значение минус один. Затем увеличим его несколько раз, выводя каждый раз значение

переменной. Обратите внимание: как только переменная достигнет своего максимального значения, произойдет переполнение и увеличение значения продолжится от минимума.

Рассмотрим следующий пример.

1. Создайте папку с консольным приложением `CheckingForOverflow` и добавьте ее в рабочую область.
2. В методе `Main` введите операторы, чтобы объявить и назначить целое число на единицу меньше его максимально возможного значения, а затем увеличьте его и запишите его значение в консоль три раза:

```
int x = int.MaxValue - 1;
WriteLine($"Initial value: {x}");
x++;
WriteLine($"After incrementing: {x}");
x++;
WriteLine($"After incrementing: {x}");
x++;
WriteLine($"After incrementing: {x}");
```

3. Запустите консольное приложение и проанализируйте результат:

```
Initial value: 2147483646
After incrementing: 2147483647
After incrementing: -2147483648
After incrementing: -2147483647
```

4. Теперь с помощью оператора `checked` научим компилятор предупреждать нас о переполнении, обернув операторы:

```
checked
{
  int x = int.MaxValue - 1;
  WriteLine($"Initial value: {x}");
  x++;
  WriteLine($"After incrementing: {x}");
  x++;
  WriteLine($"After incrementing: {x}");
  x++;
  WriteLine($"After incrementing: {x}");
}
```

5. Запустите консольное приложение и проанализируйте результат:

```
Initial value: 2147483646
After incrementing: 2147483647
Unhandled Exception: System.OverflowException: Arithmetic
operation resulted in an overflow.
```

6. Как и любое другое исключение, нам нужно поместить эти операторы в блок `try` и отобразить пользователю поясняющее сообщение об ошибке:

```
try
{
  // здесь должен находиться предыдущий код
}
catch(OverflowException)
{
  WriteLine("The code overflowed but I caught the exception.");
}
```

7. Запустите консольное приложение и проанализируйте результат:

```
Initial value: 2147483646
After incrementing: 2147483647
The code overflowed but I caught the exception.
```

Отключение проверки переполнения с помощью оператора unchecked

Для запрета проверки переполнения можно использовать ключевое слово `unchecked`. Оно отключает проверки переполнения, выполняемые компилятором в блоке кода.

1. Введите следующий оператор в конце предыдущего. Компилятор не выполнит этот оператор, поскольку он вызовет переполнение:

```
int y = int.MaxValue + 1;
```

2. Откройте панель PROBLEMS (Проблемы), выбрав View ▸ Problems (Вид ▸ Проблемы), и обратите внимание: проверка *во время компиляции* отображается как сообщение об ошибке, что показано на рис. 3.2.

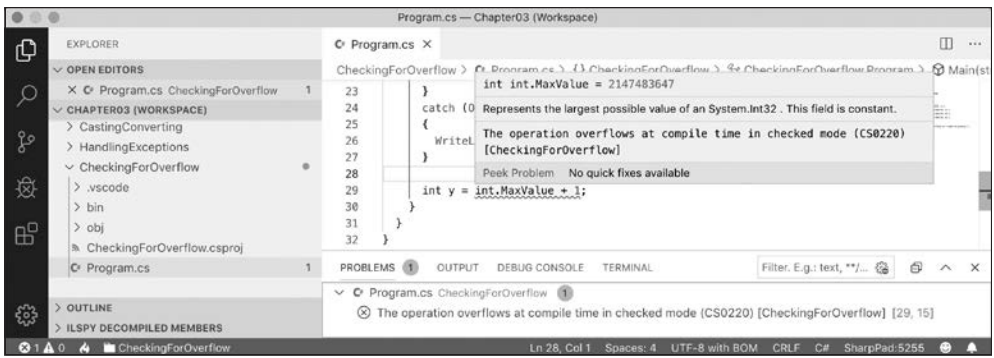


Рис. 3.2. Проверка в окне PROBLEMS (Проблемы) во время компиляции

3. Чтобы отключить проверки во время компиляции, поместите оператор в блок `unchecked`, запишите значение `y` в консоль, уменьшите его и повторите, как показано ниже:

```
unchecked
{
  int y = int.MaxValue + 1;

  WriteLine($"Initial value: {y}");
  y--;
  WriteLine($"After decrementing: {y}");
  y--;
  WriteLine($"After decrementing: {y}");
}
```

4. Запустите консольное приложение и проанализируйте результат:

```
Initial value: -2147483648
After decrementing: 2147483647
After decrementing: 2147483646
```

Разумеется, вряд ли возникнет ситуация, когда вы захотите явно отключить проверку, поскольку будет допускаться переполнение. Но, возможно, в каком-то из ваших сценариев вам понадобится реализовать подобное поведение компилятора.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 3.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Что произойдет, если разделить переменную `int` на `0`?
2. Что произойдет, если разделить переменную `double` на `0`?
3. Что происходит при переполнении переменной `int`, то есть вы присваиваете ей значения, выходящие за пределы допустимого диапазона?
4. В чем разница между операторами `x = y++;` и `x = ++y;`?
5. В чем разница между ключевыми словами `break`, `continue` и `return` при использовании в операторах цикла?
6. Из каких трех частей состоит оператор `for` и какие из них обязательны?
7. В чем разница между операторами `=` и `==`?

8. Будет ли скомпилирован следующий оператор: `for ( ; true; ) ;`?
9. Что означает символ подчеркивания _ в выражении `switch`?
10. Какой интерфейс должен реализовать объект для перечисления с помощью оператора `foreach`?

Упражнение 3.2. Циклы и переполнение

Что произойдет при выполнении кода, приведенного ниже?

```
int max = 500;
for (byte i = 0; i < max; i++)
{
  WriteLine(i);
}
```

Создайте в папке Chapter03 консольное приложение `Exercise02` и введите код из листинга, приведенного выше. Запустите консольное приложение и проанализируйте результат. Что произошло?

Какой код вы могли бы добавить (не изменяя строки предыдущего кода), чтобы получать предупреждение о проблеме?

Упражнение 3.3. Циклы и операторы

FizzBuzz — командная детская игра, обучающая делению. Игроки по очереди называют натуральные числа, заменяя те, которые делятся на 3, словом *fizz*, а те, которые делятся на 5, — *buzz*. Если встречается число, делимое и на 3, и на 5, то говорят *fizzbuzz*.

При проведении собеседований кандидатам иногда задают простые задачки в духе игры FizzBuzz. Толковый программист должен написать такую программу на бумажке или маркерной доске за пару минут.

Но вот что интересно: многие люди с профильным образованием вообще не могут справиться с этой задачей. Были даже случаи, когда кандидаты, подававшие резюме на вакансию «старший разработчик», тратили на данную программу больше 10–15 минут.

> *199 из 200 претендентов на работу программистом не могут писать код вообще. Повторяю: они не могут написать никакой код вообще.*
>
> Реджинальд Брейтуэйт

Цитата взята с сайта blog.codinghorror.com/why-cant-programmers-program/.

Дополнительную информацию вы найдете на сайте http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/.

Создайте в папке Chapter03 консольное приложение `Exercise03`, вывод которого похож на имитацию игры FizzBuzz, выполняющей счет до 100. Результат должен выглядеть примерно так, как показано на рис. 3.3.

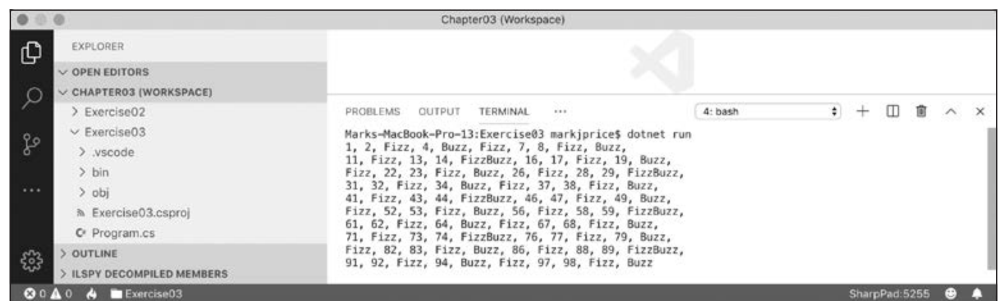


Рис. 3.3. Смоделированный вывод игры FizzBuzz

Упражнение 3.4. Обработка исключений

Создайте в папке Chapter03 консольное приложение `Exercise04`, которое запрашивает у пользователя два числа в диапазоне от 0 до 255, а затем делит первое на второе. Вывод показан ниже:

```
Enter a number between 0 and 255: 100
Enter another number between 0 and 255: 8
100 divided by 8 is 12
```

Напишите код обработчиков исключений, перехватывающих любые вызванные ошибки:

```
Enter a number between 0 and 255: apples
Enter another number between 0 and 255: bananas
FormatException: Input string was not in a correct format.
```

Упражнение 3.5. Проверка знания операций

Чему будут равны значения переменных `x` и `y` после выполнения следующих операторов?

1. `x = 3;`

   `y = 2 + ++x;`

2. `x = 3 << 2;`

   `y = 10 >> 1;`

3. `x = 10 & 8;`

   `y = 10 | 7;`

Упражнение 3.6. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- операции C#: https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/statements-expressions-operators/operators;
- побитовые операции и операции побитового сдвига: https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators;
- ключевые слова (справочник по C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/statement-keywords;
- приведение и преобразование типов (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/articles/csharp/programming-guide/types/casting-and-type-conversions.

Резюме

Из этой главы вы узнали, как разветвлять и зацикливать код, выполнять преобразование типов, обрабатывать исключения и, самое главное, как найти документацию!

Теперь вы готовы узнать, как повторно использовать блоки кода, определяя функции, как передавать значения в них и возвращать значения и вдобавок как отследить ошибки в вашем коде и удалить их!

4 Разработка, отладка и тестирование функций

Данная глава посвящена инструментам отладки, мониторингу, диагностике проблем и тестированию кода в целях отлавливания и исправления багов и обеспечения высокой производительности, стабильности и надежности.

В этой главе:

- написание функций в языке C#;
- отладка в процессе разработки;
- регистрация событий во время выполнения;
- модульное тестирование.

Написание функций в языке C#

Фундаментальный принцип программирования — *DRY* (Do not repeat yourself — «Не повторяйся»).

Во время программирования, если вы многократно пишете одни и те же операторы, их следует превращать в функцию. Функции похожи на крошечные программы, каждая из которых выполняет одну маленькую задачу. Например, вы можете написать функцию для расчета НДС, а затем повторно использовать ее многократно в финансовом приложении.

Как и программы, функции обычно имеют ввод и вывод. Их иногда называют «черными ящиками», где вы загружаете исходные данные с одной стороны, а результат появляется с другой. После создания вам не нужно думать о том, как они работают.

Допустим, вы хотите помочь своему ребенку выучить таблицу умножения. Для этого вы хотите упростить создание таблицы умножения для конкретного числа, скажем для 12[1]:

[1] В британских школах традиционно изучаются таблицы умножения до 12, а не до 10, как в России и других русскоязычных странах.

```
1 x 12 = 12
2 x 12 = 24
...
12 x 12 = 144
```

Ранее вы уже ознакомились с оператором `for`, поэтому знаете, что он может использоваться для генерации повторяющихся строк вывода, когда имеется регулярный шаблон, например таблица умножения на 12, как показано в следующем коде:

```
for (int row = 1; row <= 12; row++)
{
  Console.WriteLine($"{row} x 12 = {row * 12}");
}
```

Однако вместо вывода таблицы умножения на 12 мы хотим сделать ее более гибкой, чтобы она могла выводить результаты умножения для любого числа. Мы можем сделать это, создав функцию.

Написание функции для таблицы умножения

Познакомимся с функциями, написав код для составления таблицы умножения на 12.

1. В папке Code создайте папку Chapter04.
2. Запустите программу Visual Studio Code. Закройте открытую папку или рабочую область и сохраните текущую рабочую область в папке Chapter04 под именем `Chapter04.code-workspace`.
3. В папке Chapter04 создайте папку WritingFunctions, добавьте ее в рабочую область `Chapter04` и создайте проект консольного приложения в папке WritingFunctions.
4. Измените содержимое файла `Program.cs`:

```
using static System.Console;

namespace WritingFunctions
{
  class Program
  {
    static void TimesTable(byte number)
    {
      WriteLine($"This is the {number} times table:");
      for (int row = 1; row <= 12; row++)
      {
        WriteLine(
          $"{row} x {number} = {row * number}");
      }
```

```
      WriteLine();
    }

    static void RunTimesTable()
    {
      bool isNumber;
      do
      {
        Write("Enter a number between 0 and 255: ");

        isNumber = byte.TryParse(
          ReadLine(), out byte number);

        if (isNumber)
        {
          TimesTable(number);
        }
        else
        {
          WriteLine("You did not enter a valid number!");
        }
      }
      while (isNumber);
    }

    static void Main(string[] args)
    {
      RunTimesTable();
    }
  }
}
```

В этом коде обратите внимание на следующие моменты:

- мы статически импортировали тип `Console`, чтобы упростить вызовы его методов, таких как `WriteLine`;
- мы написали функцию `TimesTable`, которой должно быть передано байтовое (`byte`) значение с именем `number`;
- функция `TimesTable` не возвращает значение вызывающей стороне, поэтому оно объявляется с ключевым словом `void` перед его именем;
- функция `TimesTable` использует оператор `for`, чтобы вывести таблицу умножения для переданного ей числа;
- мы написали функцию `RunTimesTable`, которая предлагает пользователю ввести число, а затем вызывает функцию `TimesTable`, передавая ей введенное число. Функция будет работать до тех пор, пока пользователь вводит правильные числа;
- мы вызываем функцию `RunTimesTable` в методе `Main`.

5. Запустите консольное приложение.
6. Введите число, например `6`, а затем проанализируйте результат, как показано в следующем выводе:

```
Enter a number between 0 and 255: 6
This is the 6 times table:
1 x 6 = 6
2 x 6 = 12
3 x 6 = 18
4 x 6 = 24
5 x 6 = 30
6 x 6 = 36
7 x 6 = 42
8 x 6 = 48
9 x 6 = 54
10 x 6 = 60
11 x 6 = 66
12 x 6 = 72
```

Функции, возвращающие значение

Предыдущая функция выполняла действия (цикл и запись в консоль), но не возвращала значение. Допустим, вам нужно рассчитать налог с продаж или налог на добавленную стоимость (НДС). В Европе ставки НДС могут варьироваться от 8 % в Швейцарии до 27 % в Венгрии. В США государственные налоги с продаж могут варьироваться от 0 % в Орегоне до 8,25 % в Калифорнии.

1. Добавьте в класс `Program` функцию `CalculateTax`, используя вторую функцию для запуска приложения, как показано в приведенном ниже коде. Перед запуском кода обратите внимание на следующие моменты:
   - функция `CalculateTax` имеет два входных параметра: `amount`, который будет представлен суммой потраченных денег, и `twoLetterRegionCode`, представленный регионом, в котором потрачена сумма;
   - функция `CalculateTax` выполнит вычисление с помощью оператора `switch`, а затем вернет налог с продаж или НДС на сумму в виде десятичного значения (`decimal`). Итак, перед именем функции мы объявили тип данных возвращаемого значения;
   - функция `RunCalculateTax` предлагает пользователю ввести сумму и код региона, а затем вызывает функцию `CalculateTax` и выводит результат.

```
static decimal CalculateTax(
  decimal amount, string twoLetterRegionCode)
{
  decimal rate = 0.0M;
  switch (twoLetterRegionCode)
```

```
    {
      case "CH": // Швейцария
        rate = 0.08M;
        break;
      case "DK": // Дания
      case "NO": // Норвегия
        rate = 0.25M;
        break;
      case "GB": // Великобритания
      case "FR": // Франция
        rate = 0.2M;
        break;
      case "HU": // Венгрия
        rate = 0.27M;
        break;
      case "OR": // Орегон
      case "AK": // Аляска
      case "MT": // Монтана
        rate = 0.0M;
        break;
      case "ND": // Северная Дакота
      case "WI": // Висконсин
      case "ME": // Мэриленд
      case "VA": // Вирджиния
        rate = 0.05M;
        break;
      case "CA": // Калифорния
        rate = 0.0825M;
        break;
      default: // большинство штатов США
        rate = 0.06M;
        break;
    }
    return amount * rate;
  }

  static void RunCalculateTax()
  {
    Write("Enter an amount: ");
    string amountInText = ReadLine();

    Write("Enter a two-letter region code: ");
    string region = ReadLine();

    if (decimal.TryParse(amountInText, out decimal amount))
    {
      decimal taxToPay = CalculateTax(amount, region);
      WriteLine($"You must pay {taxToPay} in sales tax.");
    }
    else
```

```
    {
      WriteLine("You did not enter a valid amount!");
    }
  }
```

2. В методе `Main` закомментируйте вызов метода `RunTimesTable` и вызовите метод `RunSalesTax`:

```
// RunTimesTable();
RunSalesTax();
```

3. Запустите консольное приложение.
4. Введите сумму, например `149`, и код региона, например `FR`, чтобы проанализировать результат:

```
Enter an amount: 149
Enter a two-letter region code: FR
You must pay 29.8 in sales tax.
```

Можете ли вы представить какие-либо проблемы с функцией `CalculateTax`? Что произойдет, если пользователь введет код, например, `fr` или `UK`? Как вы могли бы переписать функцию, чтобы улучшить ее? Будет ли применение выражения `switch` вместо оператора `switch` более понятным?

Написание математических функций

Хотя вы, возможно, никогда не создадите приложение, которое должно обладать математической функциональностью, все изучают математику в школе, поэтому использование математических методов — обычный способ изучения функций.

Преобразование чисел из кардинальных в порядковые

Числа, которые используются для подсчета, называются *кардинальными*, например 1, 2 и 3. В то время как числа, применяемые для обозначения порядка, — *порядковые*, например, 1-е, 2-е и 3-е.

Создадим функцию для преобразования кардинальных чисел в порядковые.

1. Напишите функцию `CardinalToOrdinal`, которая преобразует кардинальное значение типа `int` в порядковое значение `string`; например, преобразует `1` в `1st`, `2` в `2nd` и т. д.:

```
static string CardinalToOrdinal(int number)
{
  switch (number)
  {
    case 11: // case с 11 по 13
```

```
      case 12:
      case 13:
        return $"{number}th";
      default:
        int lastDigit = number % 10;

        string suffix = lastDigit switch
        {
          1 => "st",
          2 => "nd",
          3 => "rd",
          _ => "th"
        };
        return $"{number}{suffix}";
    }
  }

static void RunCardinalToOrdinal()
{
  for (int number = 1; number <= 40; number++)
  {
    Write($"{CardinalToOrdinal(number)} ");
  }
  WriteLine();
}
```

В этом коде обратите внимание на следующие моменты:

- функция `CardinalToOrdinal` имеет один вход: параметр типа `int` с именем `number` — и один выход: возвращаемое значение типа `string`;
- оператор `switch` используется для обработки особых случаев: `11`, `12` и `13`;
- затем вложенный оператор `switch` обрабатывает все остальные случаи: если последней цифрой является `1`, то используется суффикс `st`, если `2` — то `nd`, если `3` — то `rd`, и если любой другой — то суффикс `th`;
- метод `RunCardinalToOrdinal` использует оператор `for` для перебора значений от 1 до 40, вызывая функцию `CardinalToOrdinal` для каждого числа и записывая возвращаемую строку в консоль, разделенную пробелом.

2. В методе `Main` закомментируйте вызов метода `RunSalesTax` и вызовите метод `RunCardinalToOrdinal`:

```
// RunTimesTable();
// RunSalesTax();
RunCardinalToOrdinal();
```

3. Запустите консольное приложение и проанализируйте результат:

```
1st 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th 17th
18th 19th 20th 21st 22nd 23rd 24th 25th 26th 27th 28th 29th 30th 31st 32nd
33rd 34th 35th 36th 37th 38th 39th 40th
```

Вычисление факториалов с помощью рекурсии

Факториал 5 равен 120. Факториал натурального числа — это число, умноженное на «себя минус один», затем на «себя минус два» и т. д. до 1. Пример: $5 \times 4 \times 3 \times 2 \times 1 = 120$.

Факториалы записываются следующим образом: 5!, где *факториал* обозначается восклицательным знаком. Пример: 5! = 120, то есть *факториал пяти равен сто двадцати*. Восклицательный знак — хорошее обозначение для факториалов, поскольку они очень быстро увеличиваются в размерах. Как вы можете видеть в предыдущем выводе, факториал 32 и выше переполняет тип `int` ввиду слишком большого своего значения.

Мы напишем функцию `Factorial`, которая вычислит факториал для `int`, переданного ей в качестве параметра. Мы будем использовать технику, называемую *рекурсией*, то есть функцию, вызывающую саму себя в своей реализации прямо или косвенно.

Рекурсия может показаться разумным решением, но она может привести к проблемам, таким как переполнение стека из-за слишком большого количества вызовов функций, поскольку при каждом ее вызове используется слишком много памяти для хранения данных. Итерация — более практичное, хотя и менее лаконичное решение в таких языках, как C#. Более подробно вы можете прочитать на сайте https://en.wikipedia.org/wiki/Recursion_(computer_science)#Recursion_versus_iteration.

1. Добавьте функцию `Factorial` и функцию для ее вызова:

```
static int Factorial(int number)
{
  if (number < 1)
  {
    return 0;
  }
  else if (number == 1)
  {
    return 1;
  }
  else
  {
    return number * Factorial(number - 1);
  }
}

static void RunFactorial()
{
```

```
  for (int i = 1; i < 15; i++)
  {
    WriteLine($"{i}! = {Factorial(i):N0}");
  }
}
```

Как и прежде, в приведенном выше коде есть несколько важных моментов.

- Если введенное число равно нулю или отрицательно, то функция `Factorial` возвращает `0`.
- Если введенное число равно `1`, то функция `Factorial` возвращает `1` и, следовательно, перестает вызывать сама себя. Если входной параметр больше единицы, то функция `Factorial` умножает число на результат вызова самой себя с на единицу меньшим числом, переданным в качестве параметра. Это делает функцию рекурсивной.
- Метод `RunFactorial` использует оператор `for` для вывода факториалов чисел от 1 до 14, вызывает функцию `Factorial`, а затем выводит результат, отформатированный с помощью кода `N0`, который означает форматирование числа и применение разделителей тысяч с нулевыми десятичными разрядами, и повторяется в цикле, пока не будет введено другое число.

2. В методе `Main` закомментируйте вызов метода `RunCardinalToOrdinal` и вызовите метод `RunFactorial`.
3. Запустите консольное приложение и проанализируйте результаты вывода:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5,040
8! = 40,320
9! = 362,880
10! = 3,628,800
11! = 39,916,800
12! = 479,001,600
13! = 1,932,053,504
14! = 1,278,945,280
```

В предыдущем выводе это не сразу очевидно, но факториалы от 13 и выше будут перекрывать тип `int`, так как они слишком большие. 12! равен 479 001 600, что составляет около полумиллиарда. Максимальное положительное значение, которое может быть сохранено в переменной типа `int`, составляет около двух миллиардов. 13! равен 6 227 020 800, что составляет около шести миллиардов, и при сохранении в виде 32-битного целого числа оно переполняется без каких-либо проблем.

Что делать, чтобы получать уведомление о переполнении? Конечно, мы смогли решить проблему, касающуюся 13! и 14!, используя длинное 64-битное целое число вместо 32-битного целого числа `int`, но мы снова быстро достигнем предела переполнения. Цель этого раздела — понять, что числа могут выходить за границы и как с этим справиться, а не как вычислять факториалы!

4. Измените функцию `Factorial` для проверки переполнения, как показано в следующем коде:

```
checked // для переполнения
{
return number * Factorial(number - 1);
}
```

5. Измените функцию `RunFactorial` для обработки исключения переполнения при вызове функции `Factorial`:

```
try
{
  WriteLine($"{i}! = {Factorial(i):N0}");
}
catch (System.OverflowException)
{
  WriteLine($"{i}! is too big for a 32-bit integer.");
}
```

6. Запустите консольное приложение и проанализируйте результаты вывода:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5,040
8! = 40,320
9! = 362,880
10! = 3,628,800
11! = 39,916,800
12! = 479,001,600
13! is too big for a 32-bit integer.
14! is too big for a 32-bit integer.
```

Документирование функций с помощью XML-комментариев

По умолчанию при вызове такой функции, как `CardinalToOrdinal`, среда разработки Visual Studio Code отображает всплывающую подсказку с основной информацией (рис. 4.1).

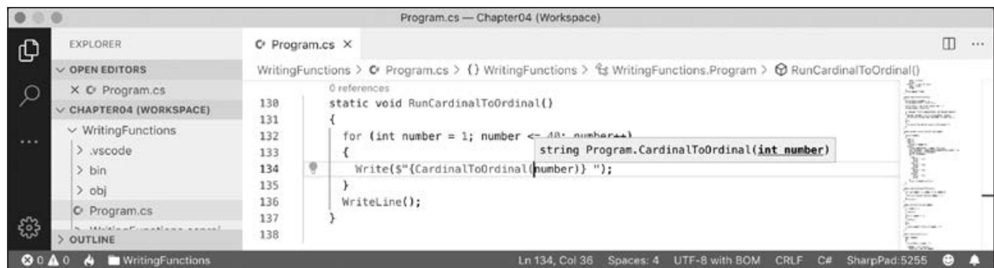


Рис. 4.1. Всплывающая подсказка, по умолчанию отображающая запись простого метода

Усовершенствуем всплывающую подсказку, добавив дополнительную информацию.

1. Если вы еще не установили расширение *C# XML Documentation Comments*, то сделайте это сейчас. Инструкции по установке расширений для программы Visual Studio Code были рассмотрены в главе 1.
2. В строке над функцией `CardinalToOrdinal` введите три косые черты и обратите внимание, что расширение превратит это в комментарий XML и распознает наличие у метода одного параметра с именем `number`.
3. Введите подходящую информацию для комментария к документации XML, как показано в коде ниже и на рис. 4.2:

```
/// <summary>
/// Передайте 32-битное целое число, и оно будет преобразовано
/// в его порядковый эквивалент.
/// </summary>
/// <param name="number"> Число — это кардинальное значение,
/// например 1, 2, 3 и т. д.</param>
/// <returns> Число как порядковое значение,
/// например 1-й, 2-й, 3-й и т. д.</returns>
```

4. Теперь при вызове функции вы увидите более подробную информацию (рис. 4.3).

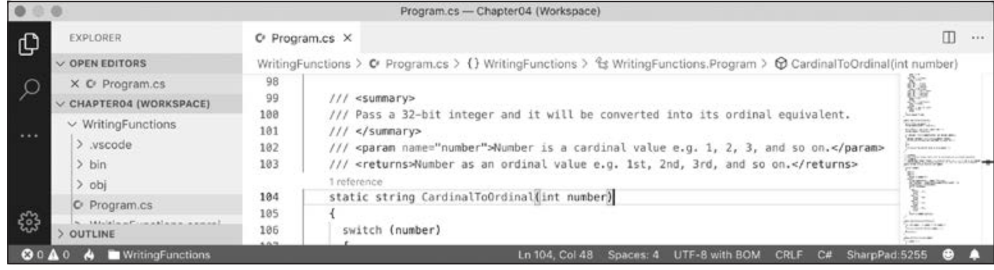


Рис. 4.2. XML-комментарии

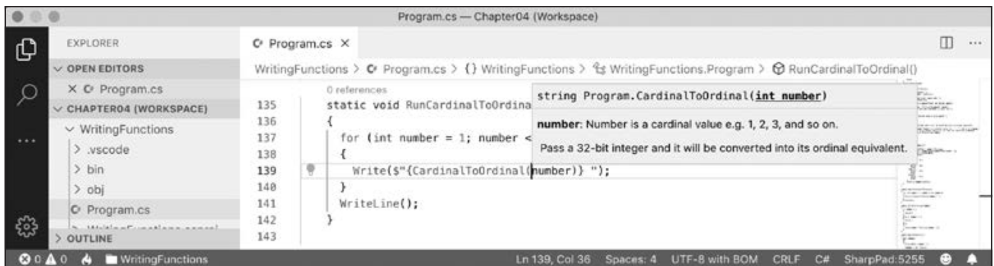


Рис. 4.3. Всплывающая подсказка, отображающая более подробную сигнатуру метода

Добавляйте XML-комментарии в код для всех ваших функций.

Использование лямбда-выражений в реализациях функций

F# — первый строго типизированный функциональный язык программирования Microsoft, который, как и C#, для выполнения .NET компилируется в IL. Функциональные языки произошли от лямбда-исчисления. Можно выделить следующие особенности: вычислительная система, основанная только на функциях; код больше похож на математические функции, чем на пошаговый рецепт.

Некоторые из важных особенностей функциональных языков определены в следующем списке.

- **Модульность.** Это же преимущество определения функций в C# применимо и к функциональным языкам. Разделите большой сложный код на более мелкие части.
- **Неизменяемость.** Переменных в контексте C# не существует. Любое значение данных внутри функции не может измениться. Вместо этого новое значение данных может быть создано из существующего. Это уменьшает количество ошибок.
- **Сопровождаемость.** Код чище и понятнее (для склонных к математике программистов!).
- Начиная с версии C# 6, компания Microsoft работает над добавлением функций в язык для поддержки более функционального подхода. Например, добавление кортежей и сопоставление с шаблоном в версии C# 7, ненулевые ссылочные типы в C# 8, улучшение сопоставления с шаблоном и добавление записей, то есть неизменяемых объектов класса, в C# 9.

В C# 6 компания Microsoft добавила поддержку членов функций, содержащих выражения.

Последовательность чисел Фибоначчи всегда начинается с 0 и 1. Затем остальная часть последовательности генерируется с использованием правила сложения двух предыдущих чисел:

```
0 1 1 2 3 5 8 13 21 34 65 ...
```

Следующим членом в последовательности будет 34 + 65, то есть 89.

Чтобы продемонстрировать разницу между императивной и декларативной реализацией функции, мы будем использовать последовательность Фибоначчи.

1. Добавьте императивную функцию `FibImperative` и функцию для ее вызова внутри оператора `for`, выполняющий цикл от 1 до 30:

```
static int FibImperative(int term)
{
  if (term == 1)
  {
    return 0;
  }
  else if (term == 2)
  {
    return 1;
  }
  else
  {
    return FibImperative(term - 1) + FibImperative(term - 2);
  }
}

static void RunFibImperative()
{
  for (int i = 1; i <= 30; i++)
  {
    WriteLine("The {0} term of the Fibonacci sequence is {1:N0}.",
      arg0: CardinalToOrdinal(i),
      arg1: FibImperative(term: i));
  }
}
```

2. В методе `Main` закомментируйте вызовы других методов и вызовите метод `RunFibImperative`.
3. Запустите консольное приложение и проанализируйте результаты вывода.

```
The 1st term of the Fibonacci sequence is 0.
The 2nd term of the Fibonacci sequence is 1.
```

```
The 3rd term of the Fibonacci sequence is 1.
The 4th term of the Fibonacci sequence is 2.
The 5th term of the Fibonacci sequence is 3.
The 6th term of the Fibonacci sequence is 5.
The 7th term of the Fibonacci sequence is 8.
The 8th term of the Fibonacci sequence is 13.
The 9th term of the Fibonacci sequence is 21.
The 10th term of the Fibonacci sequence is 34.
The 11th term of the Fibonacci sequence is 55.
The 12th term of the Fibonacci sequence is 89.
The 13th term of the Fibonacci sequence is 144.
The 14th term of the Fibonacci sequence is 233.
The 15th term of the Fibonacci sequence is 377.
The 16th term of the Fibonacci sequence is 610.
The 17th term of the Fibonacci sequence is 987.
The 18th term of the Fibonacci sequence is 1,597.
The 19th term of the Fibonacci sequence is 2,584.
The 20th term of the Fibonacci sequence is 4,181.
The 21st term of the Fibonacci sequence is 6,765.
The 22nd term of the Fibonacci sequence is 10,946.
The 23rd term of the Fibonacci sequence is 17,711.
The 24th term of the Fibonacci sequence is 28,657.
The 25th term of the Fibonacci sequence is 46,368.
The 26th term of the Fibonacci sequence is 75,025.
The 27th term of the Fibonacci sequence is 121,393.
The 28th term of the Fibonacci sequence is 196,418.
The 29th term of the Fibonacci sequence is 317,811.
The 30th term of the Fibonacci sequence is 514,229.
```

4. Добавьте декларативную функцию `FibFunctional` и функцию для ее вызова внутри оператора `for`, выполняющего цикл от 1 до 30, как показано в следующем коде:

```
static int FibFunctional(int term) =>
  term switch
  {
    1 => 0,
    2 => 1,
    _ => FibFunctional(term - 1) + FibFunctional(term - 2)
  };

static void RunFibFunctional()
{
  for (int i = 1; i <= 30; i++)
  {
    WriteLine("The {0} term of the Fibonacci sequence is {1:N0}.",
      arg0: CardinalToOrdinal(i),
    arg1: FibFunctional(term: i));
  }
}
```

5. В методе `Main` закомментируйте вызов метода `RunFibImperative` и вызовите метод `RunFibFunctional`.
6. Запустите консольное приложение и проанализируйте результаты вывода (которые будут такими же, как и раньше).

Отладка в процессе разработки

В данном разделе вы научитесь отлаживать ошибки во время разработки.

Настройка отладчика OmniSharp для Visual Studio Code может оказаться сложной задачей. Если у вас возникли проблемы, вы можете изучить документацию на сайте https://github.com/OmniSharp/omnisharp-vscode/blob/master/debugger.md.

Преднамеренное добавление ошибок в код

Рассмотрим отладку, создав консольное приложение с преднамеренной ошибкой, для отслеживания и исправления которой мы затем будем использовать специальный инструментарий.

1. В папке Chapter04 создайте папку Debugging, добавьте ее в рабочую область и создайте в этой папке консольное приложение.
2. Выберите View ▸ Command Palette (Вид ▸ Палитра команд), введите и выберите OmniSharp: Select Project, а затем выберите проект `Debugging`.
3. Когда вы увидите всплывающее предупреждающее сообщение о том, что требуемые ресурсы отсутствуют, то для их добавления нажмите Yes (Да).
4. В папке Debugging откройте и измените файл `Program.cs`, чтобы определить функцию с преднамеренной ошибкой, и вызовите ее в методе `Main`, как показано ниже:

```
using static System.Console;

namespace Debugging
{
  class Program
  {
    static double Add(double a, double b)
    {
      return a * b; // преднамеренная ошибка!
    }

    static void Main(string[] args)
    {
```

```
            double a = 4.5; // или используйте var
            double b = 2.5;
            double answer = Add(a, b);
            WriteLine($"{a} + {b} = {answer}");
            ReadLine(); // ожидание нажатия клавиши Enter
        }
    }
}
```

5. Запустите консольное приложение и проанализируйте результат:

```
4.5 + 2.5 = 11.25
```

6. Нажмите клавишу Enter, чтобы закрыть консольное приложение.

В этом коде обнаружена следующая ошибка: `4.5` плюс `2.5` должно в результате давать `7`, но никак не `11.25`!

Мы воспользуемся инструментами отладки, чтобы расправиться с этой ошибкой.

Установка точек останова

Точки останова позволяют помечать строки кода, на которых следует приостановить выполнение программы, чтобы найти ошибки.

1. Щелкните кнопкой мыши на открывающей фигурной скобке в начале метода `Main`.
2. Выберите Debug ▸ Toggle Breakpoint (Выполнить ▸ Переключить точку останова) или нажмите клавишу F9. Слева на полях появится красный кружок, указывающий на то, что точка останова была установлена (рис. 4.4).

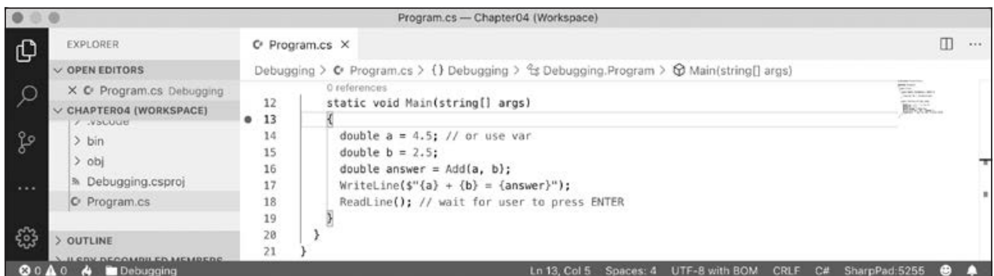


Рис. 4.4. Переключение точек останова

Точки останова можно устанавливать и убирать с помощью клавиши F9. Для тех же целей можно щелкнуть кнопкой мыши на полях. А если щелкнуть на точке останова правой кнопкой мыши, то вы увидите дополнительные команды, помимо удаления, например выключение или изменение параметров точки останова.

3. В программе Visual Studio Code выберите View ▸ Run[1] (Вид ▸ Выполнить) либо нажмите сочетание клавиш Ctrl+Shift+D или Cmd+Shift+D. Или вы можете нажать кнопку Run (Выполнить) (значок воспроизведения и значок ошибки), расположенную на левой панели навигации.
4. В верхней части панели DEBUG (Отладка)[2] нажмите раскрывающийся список справа от кнопки Start Debugging (Начать отладку) (зеленая треугольная кнопка) и выберите пункт .NET Core Launch (console) (Debugging) (рис. 4.5).

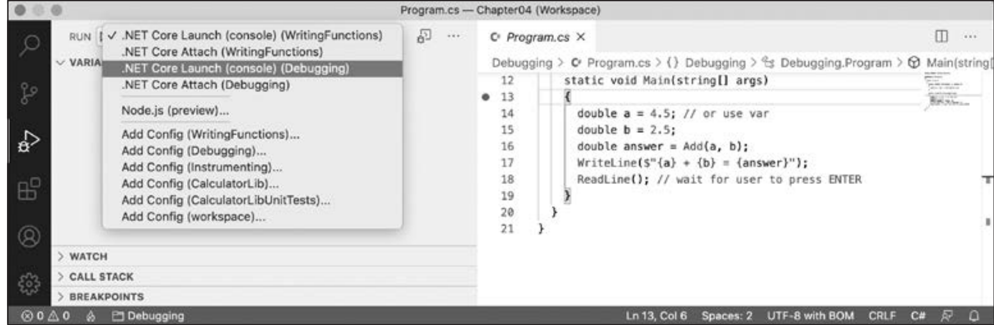


Рис. 4.5. Отладка

5. В верхней части панели DEBUG (Отладка) нажмите кнопку Start Debugging (Начать отладку) (зеленая треугольная кнопка) или нажмите клавишу F5. Среда разработки Visual Studio Code запустит консольное приложение, а затем приостановит выполнение при обнаружении точки останова. Это называется режимом приостановки выполнения. Строка кода, которая будет выполнена следующей, подсвечивается желтым и на нее указывает стрелка такого же цвета, размещенная на поле (рис. 4.6).

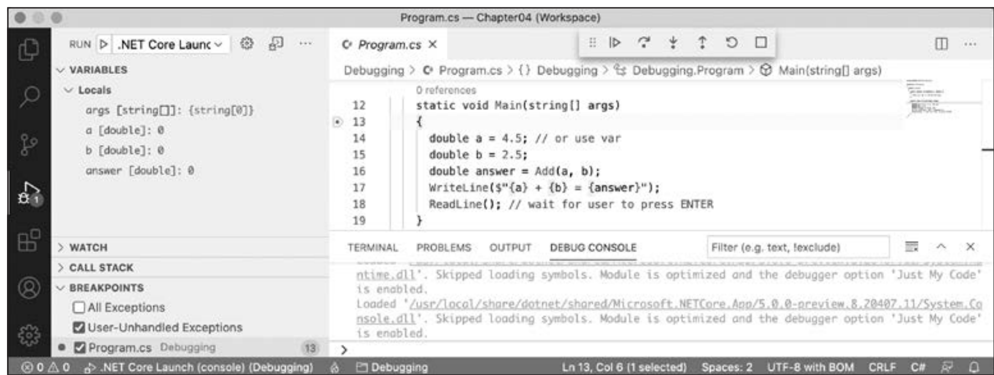


Рис. 4.6. Режим приостановки

[1] В новой версии программы используется команда View ▸ Run (Вид ▸ Выполнить).

[2] В новой версии программы панель носит название Run (Выполнить).

Навигация с помощью панели средств отладки

Программа Visual Studio Code содержит специальную панель с дополнительными элементами управления, упрощающими доступ к средствам отладки. Ниже перечислены некоторые из элементов управления.

- Continue (Продолжить)/F5 (синяя полоса и зеленый треугольник) — запускает код с текущей позиции до достижения следующей точки останова.
- Step Over (Перешагнуть)/F10, Step Into (Шагнуть внутрь)/F11 и Step Out (Шагнуть наружу)/Shift+F11 (синие стрелки над синими точками) — различными способами пошагово выполняют код (будет рассмотрено далее).
- Restart (Перезапустить)/Ctrl+Shift+F5 или Cmd+Shift+F5 (круглая черная стрелка) — завершает выполнение и сразу вновь запускает программу.
- Stop (Остановить)/Shift+F5 (красный квадрат) — прекращает выполнение программы.

Панели отладки

Панель RUN (Выполнить) в левой части окна позволяет отслеживать полезную информацию, такую как переменные, пока вы отлаживаете свой код. Эта панель состоит из четырех разделов.

- VARIABLES (Переменные), включая Locals (Локальные), отображает имена, значение и типы всех используемых локальных переменных. Отслеживайте содержимое этой панели, когда отлаживаете свой код.
- WATCH (Наблюдение) отображает значения переменных и выражений, которые вы вводите вручную.
- CALL STACK (Стек вызовов) отображает стек вызовов функций.
- BREAKPOINTS (Точки останова) отображает все ваши точки останова, что позволяет лучше контролировать их.

В режиме приостановки в нижней части области редактирования также имеется полезное окно:

- DEBUG CONSOLE (Консоль отладки) обеспечивает интерактивное взаимодействие с вашим кодом. Здесь вы можете задать вопрос и получить на него ответ. К примеру, чтобы спросить: «Сколько будет 1 + 2?» — надо набрать `1+2` и нажать клавишу Enter (рис. 4.7).

Рис. 4.7. Запрос состояния программы

Пошаговое выполнение кода

Рассмотрим несколько способов пошагового выполнения кода.

1. Выберите Run[1] ▸ Step Into (Выполнить ▸ Шагнуть внутрь) или нажмите кнопку Step Into (Шагнуть внутрь) на панели инструментов либо нажмите клавишу F11. Желтая подсветка переместится на одну строку кода.
2. Выберите Run ▸ Step Over (Выполнить ▸ Перешагнуть) или нажмите кнопку Step Over (Перешагнуть) на панели инструментов либо нажмите клавишу F10. Желтая подсветка переместится на одну строку кода. На данный момент вы можете видеть, что нет никакой разницы между использованием команды Step Into (Шагнуть внутрь) или Step Over (Перешагнуть).
3. Снова нажмите клавишу F10, чтобы желтая подсветка оказалась в строке вызова метода Add (рис. 4.8).

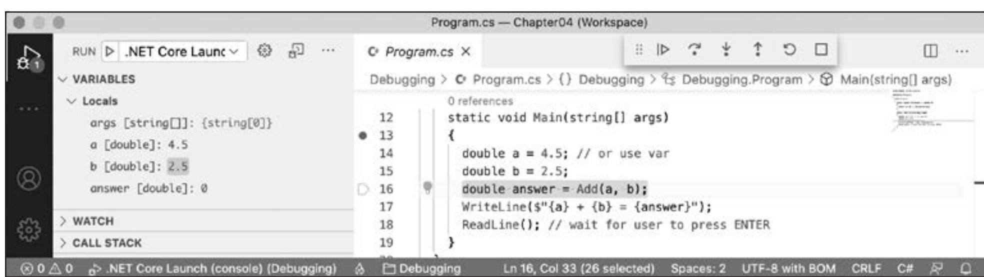


Рис. 4.8. Пошаговое выполнение кода

Разницу между командами Step Into (Шагнуть внутрь) и Step Over (Перешагнуть) можно увидеть, когда следующим оператором является вызов метода.

- Если вы нажмете кнопку Step Into (Шагнуть внутрь), то отладчик войдет в метод, так что вы сможете пройти по каждой строке этого метода.
- Если вы нажмете кнопку Step Over (Перешагнуть), то отладчик выполнит метод *целиком*, а затем остановит выполнение в первой строке, расположенной вне метода.

4. Нажмите кнопку Step Into (Шагнуть внутрь).
5. Выберите выражение a * b, щелкните на нем правой кнопкой мыши и выберите Debug:Add to Watch.

   Выражение добавляется в окно Watch, показывая, что эта операция умножает a на b и выдает результат 11.25. Обнаружилась ошибка (рис. 4.9).

[1] В новой версии программы вместо меню Debug (Отладка) используется меню Run (Выполнить).

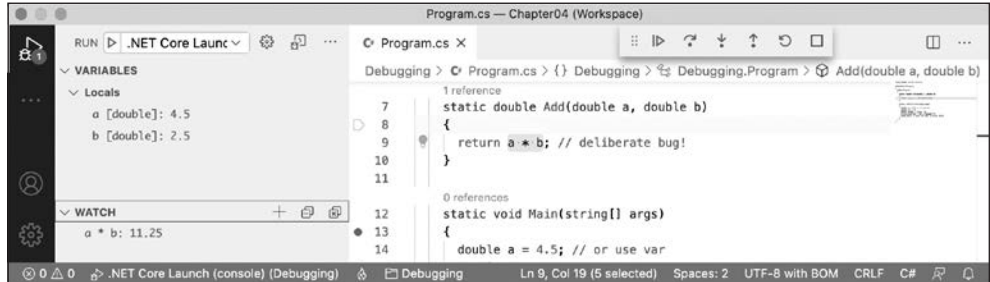


Рис. 4.9. Добавление элементов в окно WATCH

Если вы наведете указатель мыши на параметры a или b в окне редактирования кода, то появится всплывающая подсказка с текущим значением.

6. Исправьте ошибку, сменив операцию * на +.
7. Остановите, перекомпилируйте и перезапустите приложение. Для этого нажмите зеленую круглую стрелку Restart (Перезагрузить) либо сочетание клавиш Ctrl+Shift+F5 или Cmd+Shift+F5.
8. Шагните внутрь функции, при этом заметив, что она теперь правильно рассчитывается, затем нажмите кнопку Continue (Продолжить) или клавишу F5 и обратите внимание, что при записи в консоль во время отладки вывод отображается на панели DEBUG CONSOLE (Консоль отладки) вместо окна TERMINAL (Терминал), как показано на рис. 4.9.

Настройка точек останова

Вы можете легко настроить более сложные точки останова.

1. Если вы все еще выполняете отладку кода, то нажмите кнопку Stop (Стоп), или выберите Run ▸ Stop Debugging (Выполнить ▸ Остановить отладку), либо нажмите комбинацию клавиш Shift+F5.
2. В окне BREAKPOINTS (Точки останова) нажмите последнюю кнопку на мини-панели инструментов, чтобы удалить все точки останова, или выберите Run ▸ Remove All Breakpoints (Выполнить ▸ Удалить все точки останова).
3. Щелкните кнопкой мыши на операторе `WriteLine`.
4. Установите точку останова, нажав клавишу F9 или выбрав Debug ▸ Toggle Breakpoint (Отладка ▸ Переключить точку останова).
5. Щелкните правой кнопкой мыши на точке останова и выберите Edit Breakpoint (Изменить точку останова).

6. Введите выражение, например: переменная `answer` должна быть больше `9` — и обратите внимание, что для активации точки останова выражение должно иметь значение `true` (рис. 4.10).

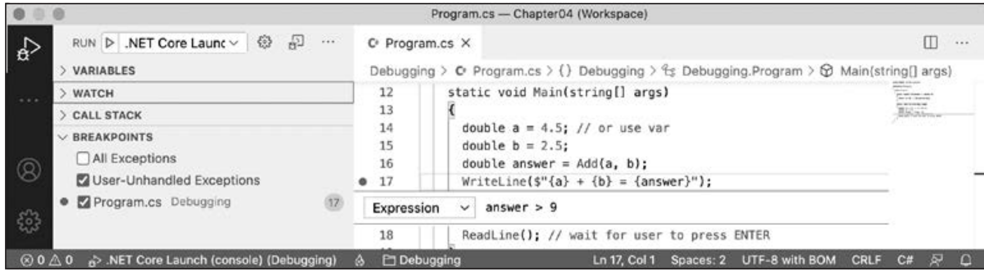


Рис. 4.10. Настройка точек останова

7. Начните отладку и обратите внимание, что точка останова не достигнута.
8. Остановите отладку.
9. Измените точку останова и установите ее выражение в значение меньше 9.
10. Начните отладку и обратите внимание, что точка останова достигнута.
11. Остановите отладку.
12. Измените точку останова и выберите пункт Hit Count (Количество обращений), затем введите число, например, `3`. Это значит, что вам нужно будет достичь точки останова три раза, прежде чем она активируется.
13. Установите указатель мыши на красный круг точки останова, чтобы увидеть итог (рис. 4.11).

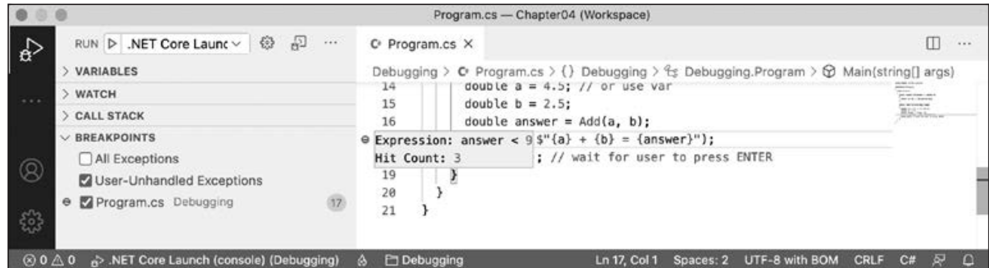


Рис. 4.11. Настроенные точки останова

Теперь вы исправили ошибку, используя отдельные средства отладки, и увидели некоторые расширенные возможности для установки точек останова.

Более подробную информацию вы можете получить на сайте https://code.visualstudio.com/docs/editor/debugging.

Регистрация событий[1] во время разработки и выполнения проекта

Если вы считаете, что все ошибки были удалены из вашего кода, то должны скомпилировать релизную версию и развернуть приложение, чтобы люди могли его использовать. Однако не существует кода без ошибок, и во время выполнения они могут возникать непредвиденно.

Известно, что конечные пользователи плохо запоминают и неохотно и неточно описывают свои действия в момент, когда произошла ошибка. Поэтому вам не следует полагаться на то, что они предоставят полезную информацию для воспроизведения проблемы, чтобы понять ее причину, а затем исправить ее.

Добавляйте в свое приложение код для регистрации событий, особенно при возникновении исключений, чтобы вы могли просматривать события и использовать их в целях отслеживания и устранения проблемы.

Существует два типа, которые можно использовать для добавления простой регистрации в ваш код: `Debug` и `Trace`.

Прежде чем мы углубимся в данные типы более подробно, коротко рассмотрим каждый из них:

- `Debug` (отладка) — используется для добавления события, которое записывается во время разработки;
- `Trace` (трассировка) — служит для добавления события, которое записывается как во время разработки, так и во время выполнения.

Работа с типами Debug и Trace

Вы видели, как используется тип `Console` и его метод `WriteLine` для предоставления вывода на консоль или на панели TERMINAL (Терминал) или DEBUG CONSOLE (Консоль отладки) в среде разработки Visual Studio Code.

[1] Также применяется термин «логирование» (от англ. logging) или «журналирование».

Существуют и более гибкие типы: `Debug` и `Trace`.

Более подробную информацию о классе Debug можно найти на сайте https://docs.microsoft.com/ru-ru/dotnet/api/system.diagnostics.debug.

Классы `Debug` и `Trace` могут вести записи в любой *прослушиватель трассировки* (trace listener). Это тип, который можно настроить для записи результатов в любом необходимом для вас месте, когда вызывается метод `Trace.WriteLine`. .NET предоставляет несколько прослушивателей трассировки, и вы даже можете создать собственный, унаследовав тип `TraceListener`.

Увидеть список прослушивателей трассировки, полученных из TraceListener, можно, перейдя по ссылке https://docs.microsoft.com/ru-ru/dotnet/api/system.diagnostics.tracelistener.

Прослушиватель трассировки

Один из прослушивателей трассировки, класс `DefaultTraceListener`, настраивается автоматически и ведет запись на панель DEBUG CONSOLE (Консоль отладки) в среде разработки Visual Studio Code. В коде вы можете вручную добавить другие прослушиватели трассировки.

1. В папке Chapter04 создайте папку Instrumenting, добавьте ее в рабочую область и создайте в этой папке проект консольного приложения.
2. Измените файл `Program.cs`:

```
using System.Diagnostics;

namespace Instrumenting
{
  class Program
  {
    static void Main(string[] args)
    {
      Debug.WriteLine("Debug says, I am watching!");
      Trace.WriteLine("Trace says, I am watching!");
    }
  }
}
```

3. Перейдите к панели RUN (Выполнить).
4. Начните отладку, запустив консольное приложение `Instrumenting`, и обратите внимание, что на панели DEBUG CONSOLE (Консоль отладки) отображаются два сообщения синего цвета, смешанные с другой информацией отладки наподобие загруженных библиотек DLL (оранжевого цвета) (рис. 4.12).

Рис. 4.12. DEBUG CONSOLE отображает два сообщения синего цвета

Настройка прослушивателей трассировки

Теперь мы настроим другой прослушиватель трассировки, который будет производить запись в текстовый файл.

1. Измените код, добавив оператор для импорта пространства имен `System.IO`, создайте текстовый файл для записи зарегистрированных событий и включите автоматическую запись буфера, как показано ниже (выделено полужирным шрифтом):

```
using System.Diagnostics;
using System.IO;

namespace Instrumenting
{
  class Program
  {
    static void Main(string[] args)
    {
      // запись в текстовый файл, расположенный в файле проекта
      Trace.Listeners.Add(new TextWriterTraceListener(
        File.CreateText("log.txt")));

      // модуль записи текста буферизируется, поэтому данная опция
      // вызывает функцию Flush() для всех прослушивателей после записи
      Trace.AutoFlush = true;

      Debug.WriteLine("Debug says, I am watching!");
      Trace.WriteLine("Trace says, I am watching!");
    }
  }
}
```

Любой тип, представляющий файл, обычно реализует буфер для повышения производительности. Вместо немедленной записи в файл данные записываются в буфер и только после заполнения буфера записываются в один фрагмент

в файл. Такое поведение может сбить с толку при отладке, поскольку мы не сразу видим результаты! Включение AutoFlush означает, что метод Flush вызывается автоматически после каждой записи.

2. Запустите консольное приложение, введя следующую команду на панели TERMINAL (Терминал) для проекта Instrumenting и обратите внимание, что ничего не произошло:

```
dotnet run --configuration Release
```

3. На панели EXPLORER (Проводник) откройте файл log.txt и обратите внимание, что он содержит сообщение "Trace says, I am watching!".
4. Запустите консольное приложение, введя следующую команду на панели TERMINAL (Терминал) для проекта Instrumenting:

```
dotnet run --configuration Debug
```

5. На панели EXPLORER (Проводник) откройте файл log.txt и обратите внимание, что он содержит оба сообщения: "Debug says, I am watching!" и "Trace says, I am watching!".

При работе в режиме Debug активны классы и Debug, и Trace, и их результаты будут отображаться на панели DEBUG CONSOLE (Консоль отладки). При работе с режимом конфигурации Release отображается только вывод Trace. Поэтому вы можете свободно использовать вызовы Debug.WriteLine по всему коду, зная, что они будут автоматически удалены при создании релизной версии вашего приложения.

Переключение уровней трассировки

Вызовы Trace.WriteLine остаются в вашем коде даже после выпуска. Таким образом, было бы здорово контролировать их вывод. Это то, что мы можем сделать с помощью *переключателя трассировки*.

Значение переключателя трассировки можно установить с помощью числа или слова. Например, число 3 можно заменить словом Info, как показано в табл. 4.1.

Таблица 4.1. Переключение уровней трассировки

Число	Слово	Описание
0	Off	Ничего не выводит
1	Error	Выводит только ошибки
2	Warning	Выводит ошибки и предупреждения
3	Info	Выводит ошибки, предупреждения и информацию
4	Verbose	Выводит все уровни

Рассмотрим использование переключателей трассировки. Нам потребуется добавить несколько пакетов, чтобы разрешить загрузку настроек конфигурации из JSON-файла настроек `appsettings`. Более подробно эту тему мы рассмотрим в главе 7.

1. Перейдите к панели TERMINAL (Терминал).
2. Введите команду:

```
dotnet add package Microsoft.Extensions.Configuration
```

3. Введите команду:

```
dotnet add package Microsoft.Extensions.Configuration.Binder
```

4. Введите команду:

```
dotnet add package Microsoft.Extensions.Configuration.Json
```

5. Введите команду:

```
dotnet add package Microsoft.Extensions.Configuration.FileExtensions
```

6. Откройте файл `Instrumenting.csproj` и создайте дополнительный раздел `<ItemGroup>` с дополнительными пакетами, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.Extensions.Configuration"
      Version="5.0.0" />
    <PackageReference
      Include="Microsoft.Extensions.Configuration.Binder"
      Version="5.0.0" />
    <PackageReference
      Include="Microsoft.Extensions.Configuration.FileExtensions"
      Version="5.0.0" />
    <PackageReference
      Include="Microsoft.Extensions.Configuration.Json"
      Version="5.0.0" />
  </ItemGroup>
</Project>
```

7. Добавьте файл `appsettings.json` в папку Instrumenting.
8. Измените содержимое файла `appsettings.json`:

```
{
  "PacktSwitch": {
    "Level": "Info"
  }
}
```

9. В проекте `Program.cs` импортируйте пространство имен `Microsoft.Extensions.Configuration`.
10. Добавьте несколько операторов в конец метода `Main`, чтобы создать строитель конфигурации, который ищет в текущей папке файл `appsettings.json`. Скомпонуйте конфигурацию, создайте переключатель трассировки, установите уровень, привязав его к конфигурации, и затем выведите четыре уровня переключения трассировки:

```
var builder = new ConfigurationBuilder()
  .SetBasePath(Directory.GetCurrentDirectory())
  .AddJsonFile("appsettings.json",
    optional: true, reloadOnChange: true);

IConfigurationRoot configuration = builder.Build();

var ts = new TraceSwitch(
  displayName: "PacktSwitch",
  description: "This switch is set via a JSON config.");

configuration.GetSection("PacktSwitch").Bind(ts);

Trace.WriteLineIf(ts.TraceError, "Trace error");
Trace.WriteLineIf(ts.TraceWarning, "Trace warning");
Trace.WriteLineIf(ts.TraceInfo, "Trace information");
Trace.WriteLineIf(ts.TraceVerbose, "Trace verbose");
```

11. Установите точку останова на операторе `Bind`.
12. Начните отладку консольного приложения `Instrumenting`.
13. На панели VARIABLES (Переменные) разверните наблюдение за переменной `ts` и обратите внимание, что ее `Level` (уровень) установлен в значение `Off` (Выкл.), а значения `TraceError`, `TraceWarning` и т. д. равны `false`.
14. Перейдите к вызову метода `Bind`, нажав кнопку Step Into (Шагнуть внутрь), или Step Over (Перешагнуть), или клавиши F11 или F10, и обратите внимание, что переменная `ts` следит за обновлениями вплоть до уровня `Info`.
15. Перейдите к четырем вызовам метода `Trace.WriteLineIf` и обратите внимание, что все уровни вплоть до `Info` записываются в DEBUG CONSOLE (Консоль отладки), но не уровень `Verbose` (рис. 4.13).
16. Остановите отладку.
17. Измените содержимое файла `appsettings.json`, чтобы установить уровень `2` (то есть предупреждение), как показано в следующем файле JSON:

```
{
  "PacktSwitch": {
    "Level": "2"
  }
}
```

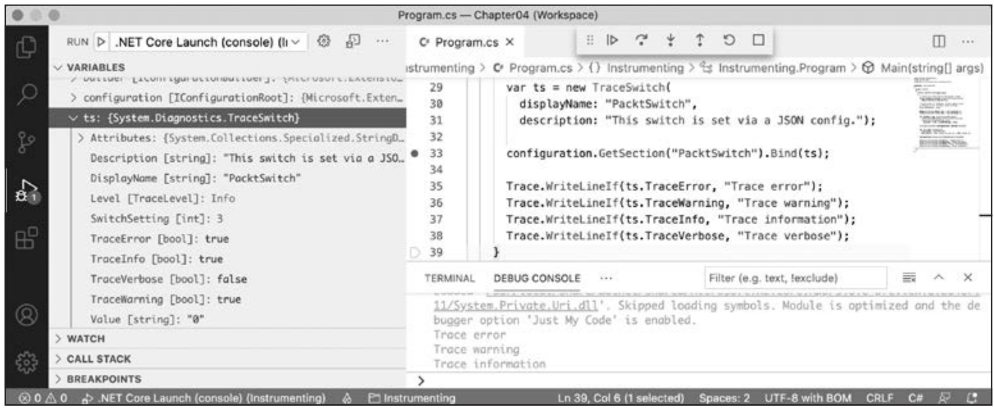


Рис. 4.13. Различные уровни трассировки, отображаемые в консоли DEBUG CONSOLE

18. Сохраните изменения.
19. Запустите консольное приложение `Instrumenting`, введя следующую команду на панели TERMINAL (Терминал):

```
dotnet run --configuration Release
```

20. Откройте файл `log.txt` и обратите внимание, что на сей раз из четырех потенциальных уровней трассировки выводится только трассировка с уровнями «ошибка» и «предупреждение», как показано в следующем текстовом файле:

```
Trace says, I am watching!
Trace error
Trace warning
```

Если аргумент не передан, то уровень переключателя трассировки по умолчанию считается `Off` (`0`), поэтому не выводится ни один из уровней переключателя.

Модульное тестирование функций

Исправление ошибок в коде стоит дорого. Чем раньше будет обнаружена ошибка в процессе разработки, тем дешевле будет ее исправить.

Модульное тестирование прекрасно подходит для поиска ошибок на ранних стадиях разработки. Некоторые разработчики даже следуют принципу, согласно которому программисты должны создавать модульные тесты, прежде чем писать код, и это называется *разработкой на основе тестирования* (Test-Driven Development, TDD).

Более подробную информацию о TDD можно узнать на сайте https://en.wikipedia.org/wiki/Test-driven_development.

Корпорация Microsoft имеет собственную систему модульного тестирования, известную как *MS Test*. Однако мы будем использовать сторонний фреймворк *xUnit.net*.

Создание библиотеки классов, требующей тестирования

Сначала мы создадим функцию, которую необходимо тестировать.

1. В папке Chapter04 создайте две подпапки с именами CalculatorLib и CalculatorLibUnitTests и добавьте их в рабочую область.
2. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и `CalculatorLib`.
3. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new classlib
```

4. Переименуйте файл `Class1.cs` в `Calculator.cs`.
5. Измените файл, чтобы определить класс `Calculator` (с преднамеренной ошибкой!):

```
namespace Packt
{
  public class Calculator
  {
    public double Add(double a, double b)
    {
      return a * b;
    }
  }
}
```

6. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet build
```

7. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и `CalculatorLibUnitTests`.
8. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new xunit
```

9. Выберите файл `CalculatorLibUnitTests.csproj` и измените конфигурацию, добавив группу элементов со ссылкой на проект `CalculatorLib`, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <IsPackable>false</IsPackable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk"
      Version="16.8.0" />
    <PackageReference Include="xunit"
      Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio"
      Version="2.4.3" />
    <PackageReference Include="coverlet.collector"
      Version="1.3.0" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference
      Include="..\CalculatorLib\CalculatorLib.csproj" />
  </ItemGroup>
</Project>
```

Просмотреть ленту Microsoft NuGet для последней версии Microsoft.NET.Test.Sdk и других пакетов можно на сайте https://www.nuget.org/.

10. Переименуйте файл `UnitTest1.cs` в `CalculatorUnitTests.cs`.
11. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet build
```

Разработка модульных тестов

Правильно разработанный модульный тест будет состоять из трех частей:

- *размещение* — объявляет и создает экземпляры переменных для ввода и вывода;
- *действие* — выполняет тестируемый модуль. В нашем случае это означает вызов метода, который необходимо протестировать;
- *утверждение* — создает одно или несколько утверждений о выводе. Если оно не соответствует действительности, то тест считается неудачным. Например, при сложении 2 и 2 мы ожидаем, что результат будет 4[1].

Теперь мы напишем модульные тесты для класса `Calculator`.

1. Откройте `CalculatorUnitTests.cs`, переименуйте класс в `CalculatorUnitTests`, импортируйте пространство имен `Packt` и измените его, чтобы иметь два метода тестирования для сложения 2 и 2 и 2 и 3:

[1] Этот принцип называется AAA, от английского Arrange — Act — Assert.

```
using Packt;
using Xunit;

namespace CalculatorLibUnitTests
{
  public class CalculatorUnitTests
  {
    [Fact]
    public void TestAdding2And2()
    {
      // размещение
      double a = 2;
      double b = 2;
      double expected = 4;
      var calc = new Calculator();

      // действие
      double actual = calc.Add(a, b);

      // утверждение
      Assert.Equal(expected, actual);
    }

    [Fact]
    public void TestAdding2And3()
    {
      // размещение
      double a = 2;
      double b = 3;
      double expected = 5;
      var calc = new Calculator();

      // действие
      double actual = calc.Add(a, b);

      // утверждение
      Assert.Equal(expected, actual);
    }
  }
}
```

Выполнение модульных тестов

Теперь мы готовы запустить модульные тесты и посмотреть результаты.

1. На панели `TERMINAL` (Терминал) приложения `CalculatorLibUnitTest` введите следующую команду:

```
dotnet test
```

2. Обратите внимание: по результатам два теста были выполнены, один тест пройден и один тест не пройден, как показано на рис. 4.14.

Рис. 4.14. Результаты модульного теста

3. Исправьте ошибку в методе `Add`.
4. Снова запустите модульные тесты, чтобы убедиться в устранении ошибки.
5. Закройте рабочую область.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 4.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Что означает в языке C# ключевое слово `void`?
2. В чем разница между императивным и функциональным стилями программирования?
3. Какова разница между сочетаниями клавиш F5, Ctrl или Cmd+F5, Shift+F5 и Ctrl или Cmd+Shift+F5 в программе Visual Studio Code?
4. Куда записывает выходные данные метод `Trace.WriteLine`?

5. Каковы пять уровней трассировки?
6. В чем разница между классами `Debug` и `Trace`?
7. Как расшифровывается принцип AAA в модульном тестировании?
8. Каким атрибутом вы должны дополнять методы тестирования при написании модульного теста с помощью `xUnit`?
9. Какая команда `dotnet` выполняет тесты `xUnit`?
10. Что такое TDD?

Упражнение 4.2. Функции, отладка и модульное тестирование

Простые множители — это комбинация наименьших простых чисел, которые при умножении в результате дают исходное число. Рассмотрим следующий пример:

- простой множитель 4: 2×2;
- простой множитель 7: 7;
- простой множитель 30: $5 \times 3 \times 2$;
- простой множитель 40: $5 \times 2 \times 2 \times 2$;
- простой множитель 50: $5 \times 5 \times 2$.

Создайте рабочую область; библиотеку классов с методом `PrimeFactors`, которая при передаче переменной `int` в качестве параметра возвращает строку, показывающую ее простые множители; проект модульных тестов и консольное приложение для его использования.

Для простоты можно предположить, что наибольшее введенное число будет равно `1000`.

Используйте инструменты отладки и напишите модульные тесты с целью убедиться, что ваша функция работает правильно с несколькими входными параметрами и возвращает правильные выходные параметры.

Упражнение 4.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- отладка в программе Visual Studio Code: https://code.visualstudio.com/docs/editor/debugging;
- инструкция по настройке отладчика .NET: https://github.com/OmniSharp/omnisharp-vscode/blob/master/debugger.md;

- пространство имен `System.Diagnostics`: https://docs.microsoft.com/ru-ru/dotnet/core/api/system.diagnostics;
- модульное тестирование в .NET: https://docs.microsoft.com/ru-ru/dotnet/core/testing/;
- xUnit.net: http://xunit.github.io/.

Резюме

Вы изучили, как в среде разработки Visual Studio Code писать многоразовые функции как в императивном, так и в функциональном стиле, использовать средства отладки, диагностики и модульного тестирования кода.

Далее вы узнаете, как создавать пользовательские типы с помощью технологий объектно-ориентированного программирования.

5 Создание пользовательских типов с помощью объектно-ориентированного программирования

Данная глава посвящена созданию пользовательских типов с помощью принципов *объектно-ориентированного программирования (ООП)*. Вы узнаете о различных категориях элементов, которые может иметь тип, в том числе о полях для хранения данных и методах для выполнения действий. Вы будете применять концепции ООП, такие как агрегирование и инкапсуляция. Вы изучите языковые функции, такие как поддержка синтаксиса кортежей и переменные `out`, литералы для значений по умолчанию и автоматически определяемые имена кортежей.

В этой главе:

- коротко об ООП;
- сборка библиотек классов;
- хранение данных в полях;
- запись и вызов методов;
- управление доступом с помощью свойств и индексаторов;
- сопоставление шаблонов с объектами;
- работа с записями.

Коротко об объектно-ориентированном программировании

Объект в реальном мире — это предмет, например автомобиль или человек. Объект в программировании часто представляет нечто в реальном мире, например товар или банковский счет, но может быть и чем-то более абстрактным.

В языке C# используются классы `class` (обычно) или структуры `struct` (редко) для определения каждого типа объекта. О разнице между классами и структурами вы узнаете в главе 6. Можно представить тип как шаблон объекта.

Ниже кратко описаны концепции объектно-ориентированного программирования.

- *Инкапсуляция* — комбинация данных и действий, связанных с объектом. К примеру, тип `BankAccount` может иметь такие данные, как `Balance` и `AccountName`, а также действия, такие как `Deposit` и `Withdraw`. При инкапсуляции часто возникает необходимость управлять тем, кто и что может получить доступ к этим действиям и данным, например ограничение доступа к внутреннему состоянию объекта или его изменению извне.
- *Композиция* — то, из чего состоит объект. К примеру, автомобиль состоит из разных частей, таких как четыре колеса, несколько сидений, двигатель и т. д.
- *Агрегирование* касается всего, что может быть объединено с объектом. Например, человек, не будучи частью автомобиля, может сидеть на водительском сиденье, а затем стать водителем. Два отдельных объекта объединены, чтобы сформировать новый компонент.
- *Наследование* — многократное использование кода с помощью подклассов, производных от базовых классов или суперклассов. Все функциональные возможности базового класса становятся доступными в производном классе. Например, базовый или суперкласс `Exception` имеет несколько членов, которые имеют одинаковую реализацию во всех исключениях. Подкласс же или производный класс `SqlException` наследует эти члены и имеет дополнительные, имеющие отношение только к тем случаям, когда возникает исключение базы данных SQL — например, свойство, содержащее информацию о подключении к базе данных.
- *Абстракция* — передача основной идеи объекта и игнорирование его деталей или особенностей. Язык C# имеет ключевое слово `abstract`, которое формализует концепцию. Если класс не явно абстрактный, то его можно описать как конкретный. Базовые классы часто абстрактны, например, суперкласс `Stream` — абстрактный, а его подклассы, такие как `FileStream` и `MemoryStream`, — конкретные. Абстракция — сложный баланс. Если вы сделаете класс слишком абстрактным, то большее количество классов сможет наследовать его, но количество возможностей для совместного использования уменьшится.
- *Полиморфизм* заключается в переопределении производным классом унаследованных методов для реализации собственного поведения.

Разработка библиотек классов

Сборки библиотек классов группируют типы в легко развертываемые модули (DLL-файлы). Не считая раздела, где вы изучали модульное тестирование, до сих пор вы создавали только консольные приложения, содержащие ваш код. Но чтобы он стал доступен для других проектов, его следует помещать в сборки библиотек классов, как это делают сотрудники корпорации Microsoft.

Создание библиотек классов

Первая задача — создать повторно используемую библиотеку классов .NET.

1. В существующей папке Code создайте папку Chapter05 с подпапкой PacktLibrary.
2. В программе Visual Studio Code выберите File ▸ Save Workspace As (Файл ▸ Сохранить рабочую область как), введите имя `Chapter05`, выберите папку Chapter05 и нажмите кнопку Save (Сохранить).
3. Выберите команду меню File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область), выберите папку PacktLibrary и нажмите кнопку Add (Добавить).
4. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new classlib
```

5. Откройте файл `PacktLibrary.csproj` и обратите внимание, что по умолчанию библиотеки классов нацелены на .NET 5 и, следовательно, могут работать только с другими сборками, совместимыми с .NET 5, как показано ниже в коде:

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

6. Измените целевую платформу для поддержки .NET Standard 2.0, как показано ниже в коде:

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

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

7. Сохраните и закройте файл.
8. На панели TERMINAL (Терминал) скомпилируйте проект, используя следующую команду: `dotnet build`.

Чтобы использовать новейшие функции языка C# и платформы .NET, поместите типы в библиотеку классов .NET 5. Для поддержки устаревших платформ .NET, таких как .NET Core, .NET Framework и Xamarin, поместите типы, которые повторно можно использовать, в библиотеку классов .NET Standard 2.0.

Определение классов

Следующая задача — определить класс, который будет представлять человека.

1. На панели EXPLORER (Проводник) переименуйте файл `Class1.cs` в `Person.cs`.
2. Щелкните кнопкой мыши на файле `Person.cs`, чтобы открыть его, и измените имя класса на `Person`.
3. Измените название пространства имен на `Packt.Shared`.

Мы делаем это, поскольку важно поместить ваши классы в логически именованное пространство имен. Лучшее имя пространства имен будет специфичным для домена, например System.Numerics для типов, связанных с расширенными числовыми функциями, но в нашем случае мы создадим типы Person, BankAccount и WondersOfTheWorld, и у них нет общего домена.

Ваш файл класса теперь должен выглядеть следующим образом:

```
using System;

namespace Packt.Shared
{
  public class Person
  {
  }
}
```

Обратите внимание, что ключевое слово `public` языка C# указывается перед словом `class`. Это ключевое слово называется *модификатором доступа*, управляющим тем, как осуществляется доступ к данному классу.

Если вы явно не определили доступ к классу с помощью ключевого слова `public` (публичный), то он будет доступен только в определяющей его сборке. Это следствие того, что неявный модификатор доступа для класса считается `internal` (внутренний). Нам же нужно, чтобы класс был доступен за пределами сборки, поэтому необходимо ключевое слово `public`.

Члены

У этого типа еще нет членов, инкапсулированных в него. Скоро мы их создадим. Членами могут быть поля, методы или специализированные версии их обоих. Их описание представлено ниже.

- *Поля* используются для хранения данных. Существует три специализированных категории полей:
  - *константы* — данные в них никогда не меняются. Компилятор буквально копирует данные в любой код, который их читает;

  - *поля, доступные только для чтения,* — данные в таких полях не могут измениться после создания экземпляра класса, но могут быть рассчитаны или загружены из внешнего источника во время создания экземпляра;
  - *события* — данные ссылаются на один или несколько методов, вызываемых автоматически при возникновении определенной ситуации, например при нажатии кнопки. Тема событий будет рассмотрена в главе 6.
- *Методы* используются для выполнения операторов. Вы уже ознакомились с некоторыми примерами в главе 4. Существует четыре специализированных метода:
  - *конструкторы* выполняются, когда вы используете ключевое слово `new` для выделения памяти и создания экземпляра класса;
  - *свойства* выполняются, когда необходимо получить доступ к данным. Они обычно хранятся в поле, но могут храниться извне или рассчитываться во время выполнения. Использование свойств — предпочтительный способ инкапсуляции полей, если только не требуется выдать наружу адрес памяти поля;
  - *индексаторы* выполняются, когда необходимо получить доступ к данным с помощью синтаксиса массива `[]`;
  - *операции* выполняются, когда необходимо применить операции типа `+` и `/` для операндов вашего типа.

Создание экземпляров классов

В этом подразделе мы создадим *экземпляр* класса `Person` (данный процесс описывается как *инстанцирование* класса).

Ссылка на сборку

Прежде чем мы сможем создать экземпляр класса, нам нужно сослаться на сборку, которая его содержит.

1. Создайте подпапку PeopleApp в папке Chapter05.
2. В программе Visual Studio Code выберите File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область), выберите папку PeopleApp и нажмите кнопку Add (Добавить).
3. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и выберите пункт `PeopleApp`.
4. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new console
```

5. На панели EXPLORER (Проводник) щелкните кнопкой мыши на файле People-App.csproj.
6. Добавьте ссылку на проект в PacktLibrary, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="../PacktLibrary/PacktLibrary.csproj" />
  </ItemGroup>
</Project>
```

7. На панели TERMINAL (Терминал) введите команду для компиляции проекта PeopleApp и его зависимого проекта PacktLibrary, как показано в следующей команде:

```
dotnet build
```

8. Выберите PeopleApp в качестве активного проекта для OmniSharp.

Импорт пространства имен для использования типа

Теперь мы готовы написать операторы для работы с классом Person.

1. В программе Visual Studio Code в папке PeopleApp откройте проект Program.cs.
2. В начале файла Program.cs введите операторы для импорта пространства имен для нашего класса People и статически импортируйте класс Console, как показано ниже:

```
using Packt.Shared;
using static System.Console;
```

3. В методе Main введите операторы для:
   - создания экземпляра типа Person;
   - вывода экземпляра, используя его текстовое описание.

   Ключевое слово new выделяет память для объекта и инициализирует любые внутренние данные. Мы могли бы использовать Person вместо ключевого слова var, но применение последнего требует меньше ввода и по-прежнему понятно, как показано ниже:

```
var bob = new Person();
WriteLine(bob.ToString());
```

Вы можете спросить: «Почему у переменной `bob` имеется метод `ToString`? Класс `Person` пуст!» Не беспокойтесь, скоро вы все узнаете!

4. Запустите приложение, введя команду `dotnet run` на панели TERMINAL (Терминал), а затем проанализируйте результат:

```
Packt.Shared.Person
```

Управление несколькими файлами

Если требуется одновременная работа с несколькими файлами, то вы можете размещать их рядом друг с другом по мере их редактирования.

1. На панели EXPLORER (Проводник) разверните два проекта.
2. Откройте файлы `Person.cs` и `Program.cs`.
3. Нажав и удерживая кнопку мыши, перетащите вкладку окна редактирования для одного из ваших открытых файлов, чтобы расположить их так, чтобы вы могли одновременно видеть оба файла `Person.cs` и `Program.cs`.

Вы можете нажать кнопку Split Editor Right (Разделить редактор) или нажать сочетание клавиш Ctrl+\ или Cmd+\, чтобы разместить два окна файла друг рядом с другом.

Более подробно о работе с пользовательским интерфейсом Visual Studio Code вы можете прочитать на сайте: https://code.visualstudio.com/docs/getstarted/userinterface.

Работа с объектами

Хотя наш класс `Person` явно не наследуется ни от какого типа, все типы косвенно наследуются от специального типа `System.Object`. Реализация метода `ToString` в типе `System.Object` выдает полные имена пространства имен и типа.

Возвращаясь к исходному классу `Person`, мы могли бы явно сообщить компилятору, что `Person` наследуется от типа `System.Object`:

```
public class Person : System.Object
```

Когда класс Б *наследуется* от класса А, мы говорим, что А — *базовый класс* или суперкласс, а Б — *производный класс*. В нашем случае `System.Object` — базовый класс (суперкласс), а `Person` — производный.

Мы также можем использовать в C# псевдоним типа — ключевое слово `object`:

```
public class Person : object
```

Наследование System.Object

Сделаем наш класс явно наследуемым от `object`, а затем рассмотрим, какие члены имеют все объекты.

1. Измените свой класс `Person` для явного наследования от `object`.
2. Затем установите указатель мыши внутри ключевого слова `object` и нажмите клавишу F12 или щелкните правой кнопкой мыши на ключевом слове `object` и выберите Go to Definition (Перейти к определению).

Вы увидите определение типа `System.Object` и его членов. Вам не нужно разбираться во всем этом определении, но обратите внимание на метод `ToString`, показанный на рис. 5.1.

```
#region Assembly netstandard, Version=2.0.0.0, Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51
// netstandard.dll
#endregion

namespace System
{
  public class Object
  {
    public Object();

    ~Object();

    public static bool Equals(Object objA, Object objB);
    public static bool ReferenceEquals(Object objA, Object objB);
    public virtual bool Equals(Object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
    protected Object MemberwiseClone();
  }
}
```

Рис. 5.1. Определение класса System.Object

Будьте уверены: другие программисты знают, что, если наследование не указано, класс наследуется от System.Object.

Хранение данных в полях

Теперь определим в классе несколько полей для хранения информации о человеке.

Определение полей

Допустим, мы решили, что информация о человеке включает имя и дату рождения. Мы инкапсулируем оба значения в классе `Person` и также сделаем поля общедоступными.

В классе Person напишите операторы для объявления двух общедоступных полей для хранения имени и даты рождения человека:

```
public class Person : object
{
  // поля
  public string Name;
  public DateTime DateOfBirth;
}
```

Вы можете использовать любые типы для полей, включая массивы и коллекции, такие как списки и словари. Они вам будут полезны при необходимости хранить несколько значений в одном именованном поле. В данном примере информация о человеке содержит только одно имя и одну дату рождения.

Модификаторы доступа

При реализации инкапсуляции важно выбрать, насколько видны элементы.

Обратите внимание: работая с классом, мы использовали ключевое слово public по отношению к созданным полям. Если бы мы этого не сделали, то поля были бы закрытыми, то есть доступными только в пределах класса.

Доступны четыре ключевых слова для модификаторов доступа, каждое из которых вы можете применить к члену класса, например к полю или методу, как показано в табл. 5.1.

Таблица 5.1. Модификаторы доступа

Модификатор доступа	**Описание**
private	Доступ ограничен содержащим типом. Используется по умолчанию
internal	Доступ ограничен содержащим типом и любым другим типом в текущей сборке
protected	Доступ ограничен содержащим типом или типами, производными от содержащего типа
public	Неограниченный доступ
internal protected	Доступ ограничен содержащим типом и любым другим типом в текущей сборке, а также типами, производными от содержащего класса. Аналогичен вымышленному модификатору доступа internal_or_protected (то есть дает доступ по принципу «internal ИЛИ protected»).
private protected	Доступ ограничен содержащим типом и любым другим типом, который наследуется от типа и находится в той же сборке. Аналогичен вымышленному модификатору доступа internal_and_protected (то есть дает доступ по принципу «internal И protected»). Эта комбинация доступна только для версии C# 7.2 или более поздних

Следует явно применять один из модификаторов доступа ко всем членам типа, даже если будет использоваться модификатор по умолчанию, private. Кроме того, поля, как правило, должны быть private или protected и вам следует создать общедоступные свойства public в целях получения или установки значений полей для контроля доступа. Позже в данной главе мы рассмотрим эту тему.

Установка и вывод значений полей

Теперь мы будем использовать эти поля в консольном приложении.

1. Убедитесь, что в начале проекта `Program.cs` импортировано пространство имен `System`.
2. Внутри метода `Main` измените операторы, чтобы задать имя и дату рождения человека, а затем выведите эти поля в надлежащем формате:

```
var bob = new Person();
bob.Name = "Bob Smith";
bob.DateOfBirth = new DateTime(1965, 12, 22);

WriteLine(
  format: "{0} was born on {1:dddd, d MMMM yyyy}",
  arg0: bob.Name,
  arg1: bob.DateOfBirth);
```

   Мы могли бы также использовать интерполяцию строк, но длинные строки будут переноситься на несколько строк, что может усложнить чтение кода в данной книге. В примерах кода, описанных в книге, помните: `{0}` — это заполнитель для `arg0` и т. д.

3. Запустите приложение и проанализируйте результат, как показано в следующем выводе:

```
Bob Smith was born on Wednesday, 22 December 1965
```

   Код формата для `arg1` состоит из нескольких частей: `dddd` означает день недели, `d` — день месяца, `MMMM` — название месяца. Строчные буквы `m` используются для значений времени в минутах, `yyyy` означает полное число года, `yy` будет означать число года с двумя цифрами.

   Вы также можете инициализировать поля, используя краткий синтаксис с фигурными скобками. Рассмотрим пример.

4. Под существующим кодом добавьте код, показанный ниже, чтобы создать запись еще об одном человеке. Обратите внимание на иной формат вывода в консоль даты рождения:

```
var alice = new Person
{
  Name = "Alice Jones",
```

```
  DateOfBirth = new DateTime(1998, 3, 7)
};

WriteLine(
  format: "{0} was born on {1:dd MMM yy}",
  arg0: alice.Name,
  arg1: alice.DateOfBirth);
```

5. Запустите приложение и проанализируйте результат:

```
Bob Smith was born on Wednesday, 22 December 1965
Alice Jones was born on 07 Mar 98
```

Помните, что ваш вывод может выглядеть по-разному в зависимости от вашего региона, то есть языка и культуры.

Хранение значения с помощью типа-перечисления

Иногда значение должно соответствовать одному из вариантов ограниченного списка. Например, у человека может быть любимое чудо света — одно из семи. А иногда значение должно соответствовать нескольким вариантам ограниченного списка. Например, человек может хотеть посетить несколько определенных чудес света из числа тех семи.

Мы можем сохранять такие данные, используя перечисления — тип `enum`.

Тип `enum` весьма эффективен по части хранения одного или нескольких значений списка (перечислителей), поскольку внутри в нем используются значения `int` в сочетании с поисковой таблицей описаний строк.

1. Добавьте новый класс в библиотеку классов, выбрав `PacktLibrary`, нажав кнопку New File (Новый файл) на мини-панели инструментов и введя имя `WondersOfTheAncientWorld.cs`.
2. Измените код в файле `WondersOfTheAncientWorld.cs`:

```
namespace Packt.Shared
{
  public enum WondersOfTheAncientWorld
  {
    GreatPyramidOfGiza,
    HangingGardensOfBabylon,
    StatueOfZeusAtOlympia,
    TempleOfArtemisAtEphesus,
    MausoleumAtHalicarnassus,
    ColossusOfRhodes,
    LighthouseOfAlexandria
  }
}
```

3. В классе `Person` добавьте следующий оператор в свой список полей:

```
public WondersOfTheAncientWorld FavoriteAncientWonder;
```

4. В методе `Main` файла `Program.cs` добавьте следующие операторы:

```
bob.FavoriteAncientWonder =
  WondersOfTheAncientWorld.StatueOfZeusAtOlympia;

WriteLine(format:
  "{0}'s favorite wonder is {1}. It's integer is {2}.",
  arg0: bob.Name,
  arg1: bob.FavoriteAncientWonder,
  arg2: (int)bob.FavoriteAncientWonder);
```

5. Запустите приложение и проанализируйте результат, как показано в следующем выводе:

```
Bob Smith's favorite wonder is StatueOfZeusAtOlympia. Its integer is 2.
```

В целях эффективности значение `enum` хранится как `int`. Значения `int` присваиваются автоматически, начиная с `0`, поэтому третье чудо света в нашем перечислении имеет значение 2. Вы можете присваивать значения `int`, которые не перечислены в списке. Они будут выводиться как значение `int` вместо имени, поскольку совпадение не будет найдено.

Хранение группы значений с помощью типа enum

Для списка перечислений мы могли бы создать коллекцию экземпляров `enum` (коллекции будут объяснены позже в этой главе), но есть способ получше. Мы можем скомбинировать несколько вариантов в одно значение с помощью *флагов*.

1. Измените перечисление, дополнив его атрибутом `[System.Flags]`.
2. Явно установите байтовое значение для каждого чуда света, представляющего разные битовые столбцы:

```
namespace Packt.Shared
{
  [System.Flags]
  public enum WondersOfTheAncientWorld : byte
  {
    None                     = 0b_0000_0000, // то есть 0
    GreatPyramidOfGiza       = 0b_0000_0001, // т. е. 1
    HangingGardensOfBabylon  = 0b_0000_0010, // т. е. 2
    StatueOfZeusAtOlympia    = 0b_0000_0100, // т. е. 4
    TempleOfArtemisAtEphesus = 0b_0000_1000, // т. е. 8
    MausoleumAtHalicarnassus = 0b_0001_0000, // т. е. 16
```

```
    ColossusOfRhodes         = 0b_0010_0000, // т. е. 32
    LighthouseOfAlexandria   = 0b_0100_0000  // т. е. 64
  }
}
```

Мы явно присваиваем значения каждому варианту, которые не будут перекрывать друг друга при просмотре битов, хранящихся в памяти. Мы также должны пометить перечисление атрибутом `System.Flags`. Обычно тип `enum` использует переменные `int`, но, поскольку нам не нужны настолько большие значения, мы можем уменьшить требования к памяти на 75 % (то есть 1 байт на значение вместо 4 байт), указав ему использовать переменные типа `byte`.

Если требуется указать, что наш список перечислений включает в себя *Висячие сады* и *Мавзолей в Галикарнасе*, то биты 16 *и* 2 должны быть установлены равными 1. Другими словами, мы сохранили бы значение `18` (табл. 5.2).

Таблица 5.2. Значения битов

64	32	16	8	4	2	1
0	0	1	0	0	1	0

В классе `Person` добавьте следующий оператор к списку полей:

```
public WondersOfTheAncientWorld BucketList;
```

3. Вернитесь к методу `Main` проекта `PeopleApp` и добавьте в него следующие операторы, применив к списку перечислений оператор `|` (логическое ИЛИ) для комбинирования значений `enum`. Мы также могли бы установить значение, приведя число `18` к типу `enum`, как показано в комментарии, но мы не должны этого делать, поскольку это затруднит понимание кода:

```
bob.BucketList =
  WondersOfTheAncientWorld.HangingGardensOfBabylon
  | WondersOfTheAncientWorld.MausoleumAtHalicarnassus;

// bob.BucketList = (WondersOfTheAncientWorld)18;

WriteLine($"{bob.Name}'s bucket list is {bob.BucketList}");
```

4. Запустите приложение и проанализируйте результат:

```
Bob Smith's bucket list is HangingGardensOfBabylon, MausoleumAtHalicarnassus
```

Используйте значения enum для сохранения комбинаций отдельных вариантов. Наследуйте enum от byte, если вариантов не более 8, от ushort — если не более 16, от uint — если не более 32 и от ulong — если не более 64.

Хранение нескольких значений с помощью коллекций

Добавим поле для хранения данных о детях человека. Перед вами пример агрегирования, поскольку дети — экземпляр класса, связанного с этим человеком, но не часть самого человека. Мы будем использовать дженерик коллекции `List<T>`.

1. Импортируйте пространство имен `System.Collections.Generic` в начало файла класса `Person.cs`:

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

   Более подробно о коллекциях вы узнаете в главе 8. Сейчас просто продолжайте набирать код.

2. Объявите новое поле в классе `Person`:

   ```
   public List<Person> Children = new List<Person>();
   ```

Код `List<Person>` читается как «список `Person`», например, «тип свойства `Children` — это список экземпляров `Person`». Следует убедиться, что коллекция инициализируется новым экземпляром списка, прежде чем мы сможем добавить элементы в коллекцию, иначе поле будет `null` и выдаст исключения времени выполнения.

Угловые скобки в названии типа `List<T>` — это функциональность языка C#, называемая *«дженерики»*, которая была добавлена в 2005 году в версии языка C# 2.0. Это просто красивый способ сделать коллекцию сильно типизированной, а именно такой, чтобы компилятор знал максимально точно, какие типы объектов могут храниться в коллекции. Дженерики улучшают производительность и правильность кода.

Сильная типизированность отличается от *статической типизированности*. Старые типы `System.Collection` статически типизированы и содержат слабо типизированные элементы `System.Object`. Новые типы `System.Collection.Generic` статически типизированы и содержат сильно типизированные экземпляры `<T>`. По иронии судьбы термин *«дженерики»* (от англ. generic — «обобщенный») означает, что мы можем использовать более конкретный статический тип!

1. В методе `Main` добавьте операторы для добавления двух детей Боба, а затем выведите количество его детей и их имена:

   ```
   bob.Children.Add(new Person { Name = "Alfred" });
   bob.Children.Add(new Person { Name = "Zoe" });

   WriteLine(
     $"{bob.Name} has {bob.Children.Count} children:");

   for (int child = 0; child < bob.Children.Count; child++)
   ```

```
{
  WriteLine($" {bob.Children[child].Name}");
}
```

Мы также можем использовать оператор `foreach`. В качестве дополнительной задачи измените оператор `for` для вывода той же информации с помощью `foreach`.

2. Запустите приложение и проанализируйте результат:

```
Bob Smith has 2 children:
Alfred
Zoe
```

Создание статического поля

Поля, которые мы создавали до сих пор, были членами *экземпляра*; это значит, для каждого созданного экземпляра класса существует разное значение каждого поля. Переменные `bob` и `alice` имеют разные значения поля `Name`.

Иногда требуется определить поле, в котором есть только одно значение, общее для всех экземпляров. Это так называемые *статические* члены, поскольку поля не единственные члены, которые могут быть статическими. Для обозначения статического члена класса используется модификатор `static`.

Далее рассмотрим, чего можем достичь с помощью статических полей.

1. В проекте `PacktLibrary` создайте файл класса `BankAccount.cs`.
2. Измените класс, предоставив ему три поля, два поля экземпляра и одно статическое поле:

```
namespace Packt.Shared
{
  public class BankAccount
  {
    public string AccountName;           // член экземпляра
    public decimal Balance;              // член экземпляра
    public static decimal InterestRate; // общедоступный член
  }
}
```

Каждый экземпляр класса `BankAccount` будет иметь собственные поля `AccountName` и `Balance`, но все экземпляры станут содержать общее значение `InterestRate`.

3. В проекте `Program.cs` и его методе `Main` добавьте операторы, чтобы установить общую процентную ставку, а затем создайте два экземпляра типа `BankAccount`, как показано ниже:

```
BankAccount.InterestRate = 0.012M; // хранение общего значения

var jonesAccount = new BankAccount();
jonesAccount.AccountName = "Mrs. Jones";
jonesAccount.Balance = 2400;

WriteLine(format: "{0} earned {1:C} interest.",
  arg0: jonesAccount.AccountName,
  arg1: jonesAccount.Balance * BankAccount.InterestRate);

var gerrierAccount = new BankAccount();
gerrierAccount.AccountName = "Ms. Gerrier";
gerrierAccount.Balance = 98;

WriteLine(format: "{0} earned {1:C} interest.",
  arg0: gerrierAccount.AccountName,
  arg1: gerrierAccount.Balance * BankAccount.InterestRate);
```

`:C` — это код формата чисел на платформе .NET, который определяет формат перевода числа в строку, специфический для денежных величин. В главе 8 вы узнаете, как управлять региональными настройками, определяющими символ валюты. На данный момент будет использоваться значение по умолчанию для вашей операционной системы. Я живу в Лондоне, в Великобритании, поэтому в моем результате будут отображаться британские фунты (`£`).

4. Запустите приложение и проанализируйте результат:

```
Mrs. Jones earned £28.80 interest.
Ms. Gerrier earned £1.18 interest.
```

Создание константного поля

Если значение поля *никогда не* должно меняться, то вы можете использовать ключевое слово `const` и присваивать значение во время компиляции.

1. В класс `Person` добавьте код, приведенный ниже:

```
// константы
public const string Species = "Homo Sapien";
```

2. В методе `Main` добавьте оператор для записи имени Боба и вида:

```
WriteLine($"{p1.Name} is a {Person.Species}");
```

Обратите внимание: чтобы прочитать константное поле, вы должны написать имя класса, а не экземпляра класса.

3. Запустите приложение и проанализируйте результат:

```
Bob Smith is a Homo Sapien
```

Примеры полей `const` в типах Microsoft включают `System.Int32.MaxValue` и `System.Math.PI`, поскольку ни одно из этих значений никогда не изменится, как вы можете видеть на рис. 5.2.

Рис. 5.2. Примеры констант

Констант следует избегать по двум важным причинам: значение должно быть известно во время компиляции и выражаемо в виде литеральной строки, логического или числового значения. Каждая ссылка в поле const во время компиляции заменяется литеральным значением, что, следовательно, не будет изменено, если значение изменится в будущей версии, а вы не перекомпилируете сборки, которые ссылаются на это значение.

Создание поля только для чтения

Лучший способ создания полей, значения которых не должны меняться, заключается в пометке их как «только для чтения».

1. В классе `Person` добавьте оператор для объявления экземпляра доступного только для чтения поля для хранения информации о домашней планете человека, как показано ниже:

```
// поля только для чтения
public readonly string HomePlanet = "Earth";
```

   Вы также можете объявить статические поля `static readonly` (только для чтения), значения которых будут общими для всех экземпляров типа.

2. В методе Main добавьте оператор для записи имени Боба и домашней планеты в консоль:

```
WriteLine($"{bob.Name} was born on {bob.HomePlanet}");
```

3. Запустите приложение и проанализируйте результат:

```
Bob Smith was born on Earth
```

Используйте поля только для чтения вместо константных полей по двум важным причинам: значение может быть вычислено или загружено во время выполнения и выражено с помощью любого исполняемого оператора. Таким образом, поле только для чтения может быть установлено с применением конструктора. Каждая ссылка в поле — «живая», поэтому любые будущие изменения будут правильно отражены при вызове кода.

Инициализация полей с помощью конструкторов

Поля часто необходимо инициализировать во время выполнения. Это делается в конструкторе, который будет вызываться при создании экземпляра класса с помощью ключевого слова new. Конструкторы выполняются до того, как любые поля будут установлены кодом, использующим этот тип.

1. В класс Person добавьте следующий выделенный полужирным шрифтом код после существующего поля только для чтения HomePlanet:

```
// поля только для чтения
public readonly string HomePlanet = "Earth";
public readonly DateTime Instantiated;

// конструкторы
public Person()
{
  // установка значений по умолчанию для полей,
  // включая поля только для чтения
  Name = "Unknown";
  Instantiated = DateTime.Now;
}
```

2. В методе Main добавьте операторы для создания экземпляра нового человека, а затем выведите начальные значения его полей:

```
var blankPerson = new Person();

WriteLine(format:
  "{0} of {1} was created at {2:hh:mm:ss} on a {2:dddd}.",
  arg0: blankPerson.Name,
```

```
  arg1: blankPerson.HomePlanet,
  arg2: blankPerson.Instantiated);
```

3. Запустите приложение и проанализируйте результат:

```
Unknown of Earth was created at 11:58:12 on a Sunday
```

Вы можете иметь несколько конструкторов в типе. Это особенно помогает побудить разработчиков устанавливать начальные значения для полей.

4. В класс `Person` добавьте операторы, чтобы определить второй конструктор, который позволяет разработчику устанавливать начальные значения для имени человека и домашней планеты:

```
public Person(string initialName, string homePlanet)
{
  Name = initialName;
  HomePlanet = homePlanet;
  Instantiated = DateTime.Now;
}
```

5. В метод `Main` добавьте код, приведенный ниже:

```
var gunny = new Person("Gunny", "Mars");

WriteLine(format:
  "{0} of {1} was created at {2:hh:mm:ss} on a {2:dddd}.",
  arg0: gunny.Name,
  arg1: gunny.HomePlanet,
  arg2: gunny.Instantiated);
```

6. Запустите приложение и проанализируйте результат:

```
Gunny of Mars was created at 11:59:25 on a Sunday
```

Установка значения поля с использованием литерала для значения по умолчанию

Одной из функций языка, представленных в C# 7.1, стал *литерал для значения по умолчанию*. В главе 2 вы уже узнали о ключевом слове `default(type)`.

Напомню, что если у вас в классе были поля, которые вы хотели инициализировать значениями типа по умолчанию в конструкторе, то вы могли использовать `default(type)`, начиная с версии C# 2.0.

1. В папке PacktLibrary создайте проект `ThingOfDefaults.cs`.
2. В проекте `ThingOfDefaults.cs` добавьте операторы для объявления класса с четырьмя полями различных типов и установите для них значения по умолчанию в конструкторе:

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

namespace Packt.Shared
{
  public class ThingOfDefaults
  {
    public int Population;
    public DateTime When;
    public string Name;
    public List<Person> People;

    public ThingOfDefaults()
    {
      Population = default(int); // C# 2.0 и более поздние версии
      When = default(DateTime);
      Name = default(string);
      People = default(List<Person>);
    }
  }
}
```

Может показаться, что компилятор должен уметь определять, какой тип мы имеем в виду, без явного указания, и вы были бы правы, но в течение первых 15 лет жизни компилятора C# этого не произошло. Наконец, в версии C# 7.1 и выше данная возможность появилась.

3. Упростите операторы, устанавливающие значения по умолчанию, как показано ниже (выделено полужирным шрифтом):

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

namespace Packt.Shared
{
  public class ThingOfDefaults
  {
    public int Population;
    public DateTime When;
    public string Name;
    public List<Person> People;

    public ThingOfDefaults()
    {
      Population = default; // C# 7.1 и более поздние версии
      When = default;
      Name = default;
      People = default;
    }
  }
}
```

Конструкторы — это особая категория *методов*. Далее рассмотрим методы более подробно.

Запись и вызов методов

Методы — это члены типа, которые выполняют блок операторов.

Возвращение значений из методов

Методы могут возвращать одно значение или ничего не возвращать.

- Если указать тип возврата `void` перед именем метода, то метод выполнит определенные действия, но не вернет значение.
- Если перед именем метода указать тип возвращаемого значения, то метод выполнит определенные действия и вернет значение.

В качестве примера вам предстоит создать два метода:

- `WriteToConsole` — будет выполнено действие (запись строки), но ничего не будет возвращено из метода, определенного со словом `void`;
- `GetOrigin` — вернет строковое значение, поскольку перед именем метода указано ключевое слово `string`.

Теперь напишем код.

1. В классе `Person` статически импортируйте тип `System.Console`.
2. Добавьте операторы для определения двух методов:

```
// методы
public void WriteToConsole()
{
  WriteLine($"{Name} was born on a {DateOfBirth:dddd}.");
}

public string GetOrigin()
{
  return $"{Name} was born on {HomePlanet}.";
}
```

3. В методе `Main` добавьте операторы для вызова двух методов:

```
bob.WriteToConsole();
WriteLine(bob.GetOrigin());
```

4. Запустите приложение и проанализируйте результат:

```
Bob Smith was born on a Wednesday.
Bob Smith was born on Earth.
```

Возвращение нескольких значений с помощью кортежей

Методы могут возвращать только одно значение одного типа. Этот тип может быть простым, таким как `string` в предыдущем примере, сложным, например `Person`, или коллекцией, например `List<Person>`.

Представьте, что нужно определить метод, который возвращает оба значения: и `string`, и `int`. Мы могли бы определить новый класс `TextAndNumber` с полями `string` и `int` и вернуть экземпляр этого сложного типа:

```
public class TextAndNumber
{
  public string Text;
  public int Number;
}

public class Processor
{
  public TextAndNumber GetTheData()
  {
    return new TextAndNumber
    {
      Text = "What's the meaning of life?",
      Number = 42
    };
  }
}
```

Однако определять класс просто для объединения двух значений не нужно, поскольку в современных версиях C# мы можем использовать *кортежи*. Кортежи — это эффективный способ объединить два и более значения в одно целое.

Кортежи давно используются в некоторых языках, таких как F# (с первой версии), но на платформе .NET поддержка была реализована только с выпуском версии 4.0 (был добавлен тип `System.Tuple`).

Только в C# 7.0 была добавлена поддержка синтаксиса языка для кортежей, использующих круглые скобки `()`. Наряду с началом поддержки кортежей в языке C# 7 на платформе .NET стал доступен новый тип `System.ValueTuple`, который

более эффективен в некоторых распространенных ситуациях, чем старый тип `System.Tuple` из версии .NET 4.0.

Рассмотрим кортежи на примере.

1. В классе `Person` добавьте операторы, чтобы определить метод, который возвращает кортеж со значениями типа `string` и `int`:

```
public (string, int) GetFruit()
{
  return ("Apples", 5);
}
```

2. В методе `Main` добавьте операторы для вызова метода `GetFruit`, а затем выведите поля кортежа:

```
(string, int) fruit = bob.GetFruit();
WriteLine($"{fruit.Item1}, {fruit.Item2} there are.");
```

3. Запустите приложение и проанализируйте результат:

```
Apples, 5 there are.
```

Присвоение имен полям кортежа

Для доступа к полям кортежа используются имена по умолчанию, например `Item1`, `Item2` и т. д.

Вы можете явно указать имена полей.

1. В классе `Person` добавьте операторы, чтобы определить метод, который возвращает кортеж с именованными полями:

```
public (string Name, int Number) GetNamedFruit()
{
  return (Name: "Apples", Number: 5);
}
```

2. В методе `Main` добавьте операторы для вызова метода и выведите именованные поля кортежа:

```
var fruitNamed = bob.GetNamedFruit();
WriteLine($"There are {fruitNamed.Number} {fruitNamed.Name}.");
```

3. Запустите приложение и проанализируйте результат, как показано в следующем выводе:

```
There are 5 Apples.
```

Автоматическое определение имен элементов кортежей

Если вы создаете кортеж из другого объекта, то можете использовать функцию, представленную в C# 7.1, которая называется *автоматическим определением имени элемента кортежа*.

В методе `Main` создайте два кортежа, каждый из которых состоит из значений `string` и `int`:

```
var thing1 = ("Neville", 4);
WriteLine($"{thing1.Item1} has {thing1.Item2} children.");

var thing2 = (bob.Name, bob.Children.Count);
WriteLine($"{thing2.Name} has {thing2.Count} children.");
```

В версии C# 7.0 оба кортежа будут использовать схемы именования `Item1` и `Item2`. В версии C# 7.1 и более поздних версиях во втором примере можно использовать имена `Name` и `Count`.

Деконструкция кортежей

Вы также можете деконструировать кортежи в отдельные переменные. Объявление деконструкции имеет тот же синтаксис, что и именование полей, но без имени переменной для самого кортежа:

```
// сохранение возвращаемого значения в переменной типа кортеж с двумя полями
(string name, int age) tupleWithNamedFields = GetPerson();
// tupleWithNamedFields.name
// tupleWithNamedFields.age

// деконструирование возвращаемого значения на две отдельные переменные
(string name, int age) = GetPerson();
// name
// age
```

Это приводит к разделению кортежа на части и назначению указанных частей новым переменным.

1. В метод `Main` добавьте код, приведенный ниже:

   ```
   (string fruitName, int fruitNumber) = bob.GetFruit();
   WriteLine($"Deconstructed: {fruitName}, {fruitNumber}");
   ```

2. Запустите приложение и проанализируйте результат:

   ```
   Deconstructed: Apples, 5
   ```

Операция деконструкции применима не только к кортежам. Любой тип может быть деконструирован, если содержит метод Deconstruct. Прочитать о деконструкции можно на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/deconstruct.

Определение и передача параметров в методы

Методы могут иметь параметры, переданные им для изменения их поведения. Параметры определяются по аналогии с объявлением переменных, только внутри круглых скобок кода метода.

1. В класс `Person` добавьте код, показанный ниже, чтобы определить два метода — первый без параметров и второй с одним параметром в скобках:

```
public string SayHello()
{
  return $"{Name} says 'Hello!'";
}

public string SayHelloTo(string name)
{
  return $"{Name} says 'Hello {name}!'";
}
```

2. В методе `Main` добавьте операторы для вызова двух методов и запишите возвращаемое значение в консоль:

```
WriteLine(bob.SayHello());
WriteLine(bob.SayHelloTo("Emily"));
```

3. Запустите приложение и проанализируйте результат:

```
Bob Smith says 'Hello!'
Bob Smith says 'Hello Emily!'
```

При вводе оператора, который вызывает метод, технология IntelliSense отображает всплывающую подсказку с именем и типом параметров, а также типом возвращаемого методом значения (рис. 5.3).

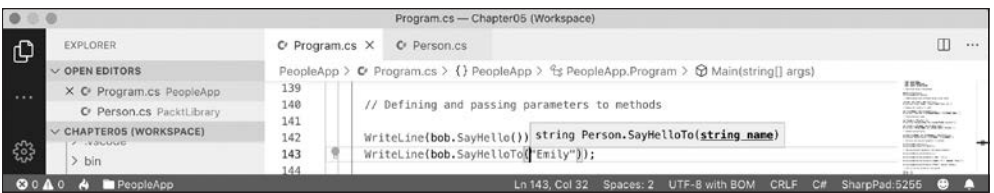


Рис. 5.3. Всплывающая подсказка IntelliSense

Перегрузка методов

Вместо двух методов с разными именами мы могли бы присвоить обоим методам одно и то же имя. Это допустимо, поскольку каждому методу присуща собственная сигнатура.

Сигнатура метода — список типов параметров, которые могут быть переданы при вызове метода (а также тип возвращаемого значения).

1. В классе `Person` измените имя метода `SayHelloTo` на `SayHello`.
2. В `Main` измените вызов метода `SayHelloTo` на метод `SayHello` и обратите внимание, что краткая информация о методе говорит вам о наличии у него одной дополнительной перегрузки, 1/2, а также 2/2 (рис. 5.4).

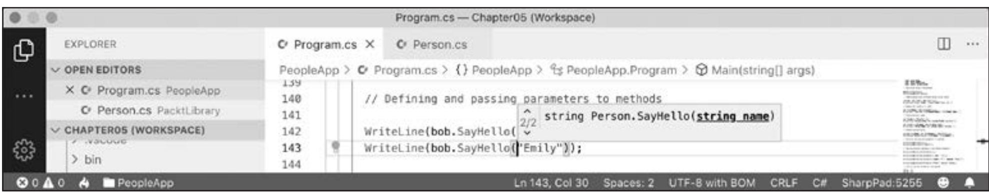


Рис. 5.4. Всплывающая подсказка IntelliSense

Используйте перегруженные методы для упрощения классов так, чтобы казалось, что у них меньше методов.

Передача необязательных параметров и именованных аргументов

Другой способ упростить методы заключается в использовании необязательных параметров. Вы определяете параметр как необязательный, назначая значение по умолчанию в списке параметров метода. Необязательные параметры всегда должны указываться последними в списке параметров.

Существует одно исключение из правила, что необязательные параметры всегда следуют последними. В C# есть ключевое слово params, которое позволяет передавать в виде массива список параметров любой длины через запятую. Прочитать о параметрах можно, перейдя по следующей ссылке: https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/params.

Сейчас вы создадите метод с тремя необязательными параметрами.

1. В классе `Person` добавьте операторы для определения метода, как показано ниже:

```
public string OptionalParameters(
  string command = "Run!",
  double number = 0.0,
  bool active = true)
{
  return string.Format(
    format: "command is {0}, number is {1}, active is {2}",
    arg0: command, arg1: number, arg2: active);
}
```

2. В методе `Main` добавьте оператор для вызова метода и запишите его возвращаемое значение в консоль:

```
WriteLine(bob.OptionalParameters());
```

3. При вводе кода появится меню IntelliSense с краткими сведениями, в которых указаны три необязательных параметра со значениями по умолчанию (рис. 5.5).

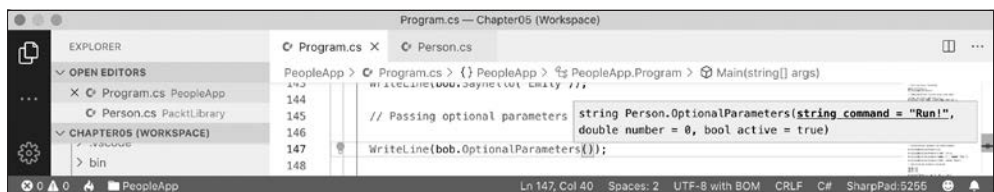


Рис. 5.5. IntelliSense отображает необязательные параметры при вводе кода

4. Запустите приложение и проанализируйте результат:

```
command is Run!, number is 0, active is True
```

5. В метод `Main` добавьте следующую строку кода, который передает значение `string` для параметра `command` и `double` — для параметра `number`:

```
WriteLine(bob.OptionalParameters("Jump!", 98.5));
```

6. Запустите приложение и проанализируйте результат, как показано в следующем выводе:

```
command is Jump!, number is 98.5, active is True
```

Значения по умолчанию для параметров `command` и `number` были заменены, но значение по умолчанию для параметра `active` по-прежнему `True`.

Необязательные параметры при вызове метода часто сочетаются с именованными, поскольку именование параметра позволяет передавать значения в порядке, отличном от того, в котором они были объявлены.

7. В метод `Main` добавьте следующую строку кода, которая так же передает значение `string` для параметра `command` и `double` — для параметра `number`, но с использованием именованных параметров, поэтому порядок, в котором они передаются, может быть заменен:

```
WriteLine(bob.OptionalParameters(
number: 52.7, command: "Hide!"));
```

8. Запустите приложение и проанализируйте вывод:

```
command is Hide!, number is 52.7, active is True
```

Вы даже можете использовать именованные параметры, чтобы пропустить необязательные параметры.

9. В метод `Main` добавьте следующую строку кода, которая передает значение `string` для параметра `command` в соответствии с порядком, пропускает параметр `number` и включает именованный параметр `active`:

```
WriteLine(bob.OptionalParameters("Poke!", active: false));
```

10. Запустите приложение и проанализируйте вывод:

```
command is Poke!, number is 0, active is False
```

Управление передачей параметров

В метод можно передать параметр одним из трех способов:

- по *значению* (по умолчанию) — можно представить как режим *«только вход»*;
- по *ссылке* в качестве параметра `ref` — можно представить как режим *«вход-выход»*;
- как параметр `out` — можно представить как режим *«только выход»*.

Рассмотрим несколько примеров передачи параметров.

1. В классе `Person` добавьте операторы, чтобы определить метод с тремя параметрами: один входной, один `ref` и один `out`, как показано в следующем методе:

```
public void PassingParameters(int x, ref int y, out int z)
{
  // параметры out не могут иметь значение по умолчанию
  // и должны быть инициализированы в методе
  z = 99;
```

```
  // инкрементирование каждого параметра
  x++;
  y++;
  z++;
}
```

2. В метод `Main` добавьте операторы для объявления нескольких переменных `int` и передачи их в метод:

```
int a = 10;
int b = 20;
int c = 30;

WriteLine($"Before: a = {a}, b = {b}, c = {c}");

bob.PassingParameters(a, ref b, out c);

WriteLine($"After: a = {a}, b = {b}, c = {c}");
```

3. Запустите приложение и проанализируйте вывод:

```
Before: a = 10, b = 20, c = 30
After: a = 10, b = 21, c = 100
```

При передаче переменной как параметра по умолчанию передается текущее *значение*, а не сама переменная. Следовательно, `x` — копия переменной `a`. Данная переменная сохраняет свое первоначальное значение, `10`. При передаче переменной в качестве параметра `ref` в метод передается *ссылка* на переменную.

Следовательно, `y` — ссылка на переменную `b`. Данная переменная инкрементируется при инкрементировании параметра `y`. При передаче переменной в качестве параметра `out` в метод передается *ссылка* на переменную.

Следовательно, `z` — ссылка на переменную `c`. Данная переменная заменяется кодом, выполняемым в методе. Мы могли бы упростить код в методе `Main`, не присваивая значение `30` переменной `c`, поскольку оно всегда будет заменено.

В языке C# 7.0 можно упростить код, использующий переменные `out`.

4. В метод `Main` добавьте операторы для объявления еще нескольких переменных, включая параметр `out` с именем `f`, объявленный прямо в операторе вызова метода (`inline`):

```
int d = 10;
int e = 20;

WriteLine(
  $"Before: d = {d}, e = {e}, f doesn't exist yet!");

// упрощенный синтаксис параметров out в C# 7.0
bob.PassingParameters(d, ref e, out int f);

WriteLine($"After: d = {d}, e = {e}, f = {f}");
```

Ключевое слово ref

В версии C# 7.0 ключевое слово `ref` используется не только для передачи параметров в метод, но и для возвращаемого значения. Это позволяет внешней переменной ссылаться на внутреннюю и изменять ее значение после вызова метода. Это может быть полезно в сложных сценариях, например для передачи пустых контейнеров, требующих последующего заполнения, в больших структурах данных, и выходит за рамки данной книги.

Существует одно исключение из правила, что необязательные параметры всегда следуют последними. В C# есть ключевое слово params, которое позволяет передавать в виде массива список параметров любой длины через запятую.

Разделение классов с помощью ключевого слова partial

При командной работе над большими проектами удобно разделять определения сложных классов на несколько файлов. Сделать это можно с помощью ключевого слова `partial`.

Допустим, нам необходимо добавить в класс `Person` операторы, которые автоматически создаются инструментом, подобным ORM (object-relational mapper, объектно-реляционный отображатель), считывающим информацию о схеме из базы данных. Если класс определен как `partial`, то мы можем разбить его на необходимое количество отдельных файлов.

1. В код класса `Person` добавьте ключевое слово `partial`, как показано в коде ниже (выделено полужирным шрифтом):

```
namespace Packt.Shared
{
  public partial class Person
  {
```

2. На панели EXPLORER (Проводник) нажмите кнопку New File (Новый файл) в папке PacktLibrary и введите имя `PersonAutoGen.cs`.
3. Добавьте операторы в новый файл:

```
namespace Packt.Shared
{
  public partial class Person
  {
  }
}
```

Остальная часть кода, который мы напишем в этой главе, будет находиться в файле `PersonAutoGen.cs`.

Управление доступом с помощью свойств и индексаторов

Ранее вы создали метод `GetOrigin`, который возвращал строковое значение, содержащее имя и происхождение человека. В таких языках, как Java, этот способ используется часто. В C# есть подход получше: *свойства*.

Свойство — обычный метод (или пара методов), который действует и выглядит как поле, когда требуется получить или установить значение, тем самым упрощая синтаксис.

Определение свойств только для чтения

Свойство только для чтения имеет только реализацию `get`.

1. В класс `Person` в файле `PersonAutoGen.cs` добавьте код, показанный ниже, чтобы определить три свойства:
   - первое играет ту же роль, что и метод `GetOrigin`, используя синтаксис `property`, допустимый во всех версиях языка C# (хотя задействует синтаксис интерполяции строк из версии C# 6 и более поздней);
   - второе возвращает приветственное сообщение, используя синтаксис лямбда-выражения (`=>`) из версии C# 6 и более поздней;
   - третье вычисляет возраст человека.

   Код представлен ниже:

```
// свойство, определенное с синтаксисом из C# 1–5
public string Origin
{
  get
  {
    return $"{Name} was born on {HomePlanet}";
  }
}

// два свойства, определенные с помощью синтаксиса
// лямбда-выражения из C# 6+
public string Greeting => $"{Name} says 'Hello!'";

public int Age => System.DateTime.Today.Year - DateOfBirth.Year;
```

Очевидно, это не лучший способ подсчета возраста, но мы не учимся рассчитывать возраст по датам рождения. Как сделать это правильно, можно узнать на сайте https://stackoverflow.com/questions/9/how-do-i-calculate-someones-age-in-c.

2. В метод `Main` добавьте код, показанный ниже:

```
var sam = new Person
{
  Name = "Sam",
  DateOfBirth = new DateTime(1972, 1, 27)
};

WriteLine(sam.Origin);
WriteLine(sam.Greeting);
WriteLine(sam.Age);
```

3. Запустите приложение и проанализируйте результат:

```
Sam was born on Earth
Sam says 'Hello!'
48
```

В результате отображается `48`, поскольку я запустил консольное приложение 15 августа 2020 года, когда Сэму было 48 лет.

Определение изменяемых свойств

Чтобы создать изменяемое свойство, вы должны использовать старый синтаксис и предоставить пару методов: не только часть `get`, но и часть `set`.

1. В файле `PersonAutoGen.cs` добавьте следующий код, чтобы определить свойство `string`, которое имеет методы `get` и `set` (также известные как геттер и сеттер):

   ```
   public string FavoriteIceCream { get; set; } // автосинтаксис
   ```

   Несмотря на то что вы вручную не создали поле для хранения названия любимого мороженого человека, оно создается автоматически компилятором.

   Иногда требуется дополнительный контроль над установкой свойств. В этом случае следует использовать более детализированный синтаксис и вручную создать закрытое поле для хранения значения свойства.

2. В файле `PersonAutoGen.cs` добавьте операторы, чтобы определить поле `string` и свойство `string`, которое имеет и `get`, и `set`:

```
private string favoritePrimaryColor;

public string FavoritePrimaryColor
```

```
{
  get
  {
    return favoritePrimaryColor;
  }
  set
  {
    switch (value.ToLower())
    {
      case "red":
      case "green":
      case "blue":
        favoritePrimaryColor = value;
        break;
      default:
        throw new System.ArgumentException(
          $"{value} is not a primary color. " +
          "Choose from: red, green, blue.");
    }
  }
}
```

3. В методе `Main` добавьте операторы, чтобы задать любимое мороженое и цвет Сэма, а затем запишите их в консоль:

```
sam.FavoriteIceCream = "Chocolate Fudge";

WriteLine($"Sam's favorite ice-cream flavor is {sam.FavoriteIceCream}.");

sam.FavoritePrimaryColor = "Red";

WriteLine($"Sam's favorite primary color is {sam.FavoritePrimaryColor}.");
```

4. Запустите приложение и проанализируйте результат:

```
Sam's favorite ice-cream flavor is Chocolate Fudge.
Sam's favorite primary color is Red.
```

Если вы попытаетесь присвоить в качестве цвета любое значение, отличное от красного, зеленого или синего, то код вызовет исключение. Затем вызывающий код может воспользоваться оператором `try` для отображения сообщения об ошибке.

Используйте свойства вместо полей, если хотите провалидировать сохраняемое значение, привязать данные в XAML (эта тема рассматривается в главе 21) и считывать и записывать поля без методов, таких как GetAge и SetAge.

Узнать больше об инкапсуляции полей можно на сайте https://stackoverflow.com/questions/1568091/why-use-getters-and-setters-accessors.

Определение индексаторов

Индексаторы позволяют вызывающему коду использовать синтаксис массива, чтобы получить доступ к свойству. К примеру, тип `string` определяет *индексатор*, поэтому вызывающий код может обращаться к отдельным символам в строке по отдельности.

Мы определим индексатор, чтобы упростить доступ к информации о детях человека.

1. В файле `PersonAutoGen.cs` добавьте код, показанный ниже, чтобы определить индексатор, получающий и устанавливающий ребенка на основе индекса (позиции) ребенка:

```
// индексаторы
public Person this[int index]
{
  get
  {
    return Children[index];
  }
  set
  {
    Children[index] = value;
  }
}
```

   Вы можете перегружать индексаторы так, чтобы можно было использовать разные типы в качестве их параметров. Например, так же, как вы передаете `int`, вы можете передавать и значение типа `string`.

2. В метод `Main` добавьте код, показанный ниже. После этого мы получим доступ к первому и второму дочерним элементам, используя более длинное поле `Children` и укороченный синтаксис индексатора:

```
sam.Children.Add(new Person { Name = "Charlie" });
sam.Children.Add(new Person { Name = "Ella" });

WriteLine($"Sam's first child is {sam.Children[0].Name}");
WriteLine($"Sam's second child is {sam.Children[1].Name}");
WriteLine($"Sam's first child is {sam[0].Name}");
WriteLine($"Sam's second child is {sam[1].Name}");
```

3. Запустите приложение и проанализируйте результат:

```
Sam's first child is Charlie
Sam's second child is Ella
Sam's first child is Charlie
Sam's second child is Ella
```

Сопоставление шаблонов с объектами

В главе 3 вы познакомились с основами сопоставления шаблонов. В следующем разделе мы более подробно рассмотрим данную тему.

Создание и работа с библиотеками классов .NET 5

Расширенные функции сопоставления с образцом доступны только в библиотеках классов .NET 5, поддерживающих C# 9 или более поздние версии. Во-первых, мы увидим, какие функции сопоставления с шаблоном были доступны до усовершенствования версии C# 9.

1. Создайте каталог PacktLibrary9 в папке Chapter05.
2. В Visual Studio Code выберите команду меню File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область), выберите пункт PacktLibrary9 и нажмите кнопку Add (Добавить).
3. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал), а затем выберите пункт PacktLibrary9.
4. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new classlib
```

5. На панели EXPLORER (Проводник) в папке PeopleApp выберите файл PeopleApp.csproj.
6. Добавьте элемент версии языка, чтобы принудительно использовать компилятор C# 8. Добавьте также ссылку на проект в PacktLibrary9, как показано ниже в коде:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <LangVersion>8</LangVersion>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="../PacktLibrary/PacktLibrary.csproj" />
    <ProjectReference Include="../PacktLibrary9/PacktLibrary9.csproj" />
  </ItemGroup>
</Project>
```

7. Выберите команду меню Terminal ▸ New TERMINAL (Терминал ▸ Новый терминал), а затем и выберите пункт PeopleApp.
8. На панели TERMINAL (Терминал) введите команду для компиляции проекта PeopleApp и его зависимых проектов:

```
dotnet build
```

Определение пассажиров при полете

В следующем примере мы определим некоторые классы, которые представляют различные типы пассажиров при полете, а затем для определения стоимости полета мы будем использовать выражение переключения с сопоставлением с шаблоном.

1. На панели EXPLORER (Проводник) в папке `PacktLibrary9` удалите файл `Class1.cs` и добавьте новый файл `FlightPatterns.cs`.
2. В файле `FlightPatterns.cs` добавьте операторы для определения трех типов пассажиров с разными свойствами:

```
namespace Packt.Shared
{
  public class BusinessClassPassenger
  {
    public override string ToString()
    {
      return $"Business Class";
    }
  }

  public class FirstClassPassenger
  {
    public int AirMiles { get; set; }

    public override string ToString()
    {
      return $"First Class with {AirMiles:N0} air miles";
    }
  }

  public class CoachClassPassenger
  {
    public double CarryOnKG { get; set; }

    public override string ToString()
    {
      return $"Coach Class with {CarryOnKG:N2} KG carry on";
    }
  }
}
```

3. В папке `PeopleApp` откройте файл `Program.cs` и добавьте операторы в конец метода `Main`, чтобы определить массив объектов, содержащий пять пассажиров различных типов и значений свойств. Затем перечислите их, выведите стоимость их полета:

```
object[] passengers = {
  new FirstClassPassenger { AirMiles = 1_419 },
  new FirstClassPassenger { AirMiles = 16_562 },
  new BusinessClassPassenger(),
```

```
    new CoachClassPassenger { CarryOnKG = 25.7 },
    new CoachClassPassenger { CarryOnKG = 0 },
  };

  foreach (object passenger in passengers)
  {
    decimal flightCost = passenger switch
    {
      FirstClassPassenger p when p.AirMiles > 35000 => 1500M,
      FirstClassPassenger p when p.AirMiles > 15000 => 1750M,
      FirstClassPassenger _                         => 2000M,
      BusinessClassPassenger _                      => 1000M,
      CoachClassPassenger p when p.CarryOnKG < 10.0 => 500M,
      CoachClassPassenger _                         => 650M,
      _                                             => 800M
    };

    WriteLine($"Flight costs {flightCost:C} for {passenger}");
  }
```

Просматривая предыдущий код, обратите внимание на следующее.

- Для сопоставления с шаблоном свойств объекта вы должны определить локальную переменную, которую затем можно будет использовать в выражении, таком как p.
- Чтобы сопоставить шаблон только с типом, вы можете использовать _, чтобы отбросить локальную переменную.
- Выражение switch также использует _ для обозначения своей ветви по умолчанию.

4. Запустите приложение и проанализируйте результаты вывода.

```
Flight costs £2,000.00 for First Class with 1,419 air miles
Flight costs £1,750.00 for First Class with 16,562 air miles
Flight costs £1,000.00 for Business Class
Flight costs £650.00 for Coach Class with 25.70 KG carry on
Flight costs £500.00 for Coach Class with 0.00 KG carry on
```

Изменения сопоставления с шаблоном в C# 9

Предыдущие примеры работали с версией C# 8. Теперь мы рассмотрим некоторые изменения в версии C# 9 и более поздних версиях. Во-первых, для исключения при сопоставлении типов вам больше не нужно использовать символ подчеркивания.

1. В папке PeopleApp откройте файл Program.cs и удалите символ _ из одной из веток.
2. На панели TERMINAL (Терминал) введите команду dotnet build, чтобы скомпилировать консольное приложение, и обратите внимание на ошибку компиляции, которая объясняет, что эта функция не поддерживается версией C# 8.0.

3. Откройте файл `PeopleApp.csproj` и удалите элемент языковой версии C# 8.0.
4. В папке `PeopleApp` откройте файл `Program.cs` и измените ветви для пассажиров первого класса, чтобы использовать вложенное выражение `switch` и новую поддержку условных выражений, таких как `>`, как показано в следующем коде:

```
decimal flightCost = passenger switch
{
  /* Синтаксис C# 8
  FirstClassPassenger p when p.AirMiles > 35000 => 1500M,
  FirstClassPassenger p when p.AirMiles > 15000 => 1750M,
  FirstClassPassenger                           => 2000M, */

  // Синтаксис C# 9
  FirstClassPassenger p => p.AirMiles switch
  {
    > 35000 => 1500M,
    > 15000 => 1750M,
    _       => 2000M
  },

  BusinessClassPassenger                        => 1000M,
  CoachClassPassenger p when p.CarryOnKG < 10.0 => 500M,
  CoachClassPassenger                           => 650M,
  _                                             => 800M
};
```

5. Запустите консольное приложение и проанализируйте результаты вывода.

Более подробно о сопоставлении шаблонов вы можете узнать на сайте https://stackoverflow.com/questions/1568091/why-use-getters-and-setters-accessors.

Работа с записями

Прежде чем мы начнем подробно изучать новую функцию языка записей в C# 9, давайте рассмотрим некоторые другие связанные с ней новые функции.

Свойства только для инициализации

В этой главе использован синтаксис инициализации объекта для создания экземпляров объектов и установки начальных свойств. Эти свойства также можно изменить после создания экземпляра.

Иногда будет возникать необходимость обрабатывать свойства как `readonly`, чтобы их можно было установить во время создания экземпляра, но не после него. Новое

ключевое слово `init` позволяет это сделать. Его можно использовать вместо ключевого слова `set`.

1. В папке `PacktLibrary9` добавьте новый файл `Records.cs`.
2. В файле `Records.cs` определите неизменяемый класс `person`, как показано в следующем коде:

```
namespace Packt.Shared
{
  public class ImmutablePerson
  {
    public string FirstName { get; init; }
    public string LastName { get; init; }
  }
}
```

3. В файле `Program.cs` в конце метода `Main` добавьте операторы для создания экземпляра нового пассажира, а затем попробуйте изменить одно из его свойств, как показано ниже в коде:

```
var jeff = new ImmutablePerson
{
  FirstName = "Jeff",
  LastName = "Winger"
};

jeff.FirstName = "Geoff";
```

4. Скомпилируйте консольное приложение и обратите внимание на ошибку компиляции:

```
Program.cs(254,7): error CS8852: Init-only property or indexer
'ImmutablePerson.FirstName' can only be assigned in an object initializer,
or on 'this' or 'base' in an instance constructor or an 'init' accessor.
[/Users/markjprice/Code/Chapter05/PeopleApp/PeopleApp.csproj]
```

5. Закомментируйте попытку установить свойство `LastName` после создания экземпляра.

Записи

Свойства, доступные только для инициализации, обеспечивают некоторую неизменность C#. Вы можете развить эту концепцию с помощью *записей*. Записи определяются с помощью ключевого слова `record` вместо ключевого слова `class`. Это делает весь объект неизменным, поэтому он работает как значение.

Записи не должны содержать состояние (свойства и поля), которое изменяется после создания экземпляра. Вместо них вы создаете новые записи из существующих

с любым измененным состоянием. Это называется *недеструктивной мутацией*. Для этого в языке C# 9 введено ключевое слово `with`.

1. Откройте файл `Records.cs` и добавьте запись с именем `ImmutableVehicle`, как показано в следующем коде:

```
public record ImmutableVehicle
{
  public int Wheels { get; init; }
  public string Color { get; init; }
  public string Brand { get; init; }
}
```

2. Откройте файл `Program.cs` и в конце метода `Main` добавьте операторы для создания автомобиля, а затем его мутированной копии, как показано в следующем коде:

```
var car = new ImmutableVehicle
{
  Brand = "Mazda MX-5 RF",
  Color = "Soul Red Crystal Metallic",
  Wheels = 4
};

var repaintedCar = car with { Color = "Polymetal Grey Metallic" };

WriteLine("Original color was {0}, new color is {1}.",
  arg0: car.Color, arg1: repaintedCar.Color);
```

3. Запустите приложение, проанализируйте результаты вывода и обратите внимание на изменение цвета автомобиля в измененной копии:

```
Original color was Soul Red Crystal Metallic, new color is Polymetal Grey
Metallic.
```

Упрощение членов данных

В следующем классе `Age` является приватным полем, поэтому к нему можно получить доступ только внутри класса:

```
public class Person
{
  int Age; // приватное поле по умолчанию
}
```

Но с ключевым словом `record` поле становится общедоступным свойством только для инициализации:

```
public record Person
{
```

```
  int Age; // свойство public эквивалентно:
  // public int Age { get; init; }
}
```

Это сделано для того, чтобы сделать определение записей кратким и понятным.

Позиционирование записей

Вместо использования синтаксиса инициализации объекта с фигурными скобками иногда вы можете предоставить конструктор с позиционными параметрами, как вы уже изучали ранее в этой главе. Вы также можете комбинировать это с деконструктором для разделения объекта на отдельные части, как показано в следующем коде:

```
public record ImmutableAnimal
{
  string Name; // то есть общедоступное только для инициализации
  string Species;

  public ImmutableAnimal(string name, string species)
  {
    Name = name;
    Species = species;
  }

  public void Deconstruct(out string name, out string species)
  {
    name = Name;
    species = Species;
  }
}
```

Для вас могут быть сгенерированы свойства, конструктор и деструктор.

1. В файле `Records.cs` добавьте операторы для определения другой записи, как показано в следующем коде:

   ```
   // более простой способ определить запись
   public data class ImmutableAnimal(string Name, string Species);
   ```

2. В файле `Program.cs` добавьте операторы для создания и деконструкции животных:

   ```
   var oscar = new ImmutableAnimal("Oscar", "Labrador");
   var (who, what) = oscar; // вызов деструктора
   WriteLine($"{who} is a {what}.");
   ```

3. Запустите приложение и проанализируйте результаты вывода:

   ```
   Oscar is a Labrador.
   ```

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 5.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какие шесть модификаторов доступа вы знаете и для чего они используются?
2. В чем разница между ключевыми словами `static`, `const` и `readonly`?
3. Для чего используется конструктор?
4. Зачем с ключевым словом `enum` используется атрибут `[Flags]`, если требуется хранить комбинированные значения?
5. В чем польза ключевого слова `partial`?
6. Что вы знаете о кортежах?
7. Для чего служит ключевое слово C# `ref`?
8. Что такое перегрузка?
9. В чем разница между полем и свойством?
10. Как сделать параметр метода необязательным?

Упражнение 5.2. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- поля (руководство по программированию в C#): https://docs.microsoft.com/ru-ru/dotnet/articles/csharp/programming-guide/classes-and-structs/fields;
- модификаторы доступа (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/articles/csharp/language-reference/keywords/access-modifiers;
- перечисляемые типы (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/enum;
- конструкторы (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/articles/csharp/programming-guide/classes-and-structs/constructors;
- методы (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/articles/csharp/methods;
- свойства (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/articles/csharp/properties.

Резюме

Вы познакомились с созданием пользовательских типов с помощью объектно-ориентированного программирования. Вы узнали о ряде категорий членов, которые могут иметь тип, включая поля для хранения данных и методы для выполнения действий. Вы использовали концепции объектно-ориентированного программирования, такие как агрегирование и инкапсуляция. В главе были продемонстрированы примеры использования функций версии C# 9, таких как усовершенствования сопоставления с шаблоном объектов, свойства, доступные только для инициализации, и записи.

Далее вы прибегнете к этим концепциям, чтобы определять делегаты и события, реализовывать интерфейсы и наследовать существующие классы.

6 Реализация интерфейсов и наследование классов

Данная глава посвящена созданию новых типов на основе существующих с использованием *объектно-ориентированного программирования (ООП)*. Вы узнаете о том, как определять операции и локальные функции для выполнения простых действий, как определять делегаты и события для обмена сообщениями между типами, как реализовывать интерфейсы для разработки общей функциональности, и о дженериках. Кроме того, узнаете о разнице между ссылочными типами и типами значений, о том, как создавать новые классы, наследуя их от базовых для многократного применения функциональности, как переопределять член типа, использовать полиморфизм, создавать методы расширения и выполнять приведение классов в иерархии наследования.

В этой главе:

- настройка библиотеки классов и консольного приложения;
- упрощение методов;
- вызов и обработка событий;
- реализация интерфейсов;
- оптимизация типов под многократное использование с помощью дженериков;
- управление памятью с помощью ссылочных типов и типов значений;
- наследование классов;
- приведение в иерархиях наследования;
- наследование и расширение типов .NET.

Настройка библиотеки классов и консольного приложения

Мы начнем с определения рабочей области с двумя проектами, подобного созданному в главе 5. Если вы выполнили все упражнения в той главе, то можете открыть созданную тогда рабочую область и продолжить работу.

В противном случае следуйте приведенным ниже инструкциям.

1. В существующей папке Code создайте папку Chapter06 с двумя подпапками, PacktLibrary и PeopleApp, как показано в следующей иерархии:
   - Chapter06
     - PacktLibrary
     - PeopleApp
2. Запустите программу Visual Studio Code.
3. Выберите команду меню File ▸ Save Workspace As (Файл ▸ Сохранить рабочую область как), введите имя `Chapter06` и нажмите кнопку Save (Сохранить).
4. Выберите команду меню File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область), выберите папку PacktLibrary и нажмите кнопку Add (Добавить).
5. Выберите команду меню File ▸ Add Folder to Workspace (Файл ▸ Добавить папку в рабочую область), выберите папку PeopleApp и нажмите кнопку Add (Добавить).
6. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и выберите `PacktLibrary`.
7. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new classlib
```

8. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и выберите `PeopleApp`.
9. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new console
```

10. На панели EXPLORER (Проводник) в проекте `PacktLibrary` переименуйте файл `Class1.cs` в `Person.cs`.
11. Измените содержимое файла:

```
using System;

namespace Packt.Shared
{
  public class Person
  {
  }
}
```

12. На панели EXPLORER (Проводник) разверните папку PeopleApp и щелкните кнопкой мыши на файле `PeopleApp.csproj`.

13. Добавьте ссылку на проект в PacktLibrary:

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

<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>netcoreapp3.0</TargetFramework>
</PropertyGroup>

<ItemGroup>
  <ProjectReference
    Include="..\PacktLibrary\PacktLibrary.csproj" />
</ItemGroup>

</Project>
```

14. На панели TERMINAL (Терминал) для папки PeopleApp введите команду dotnet build и обратите внимание на выведенную информацию, указывающую, что оба проекта были успешно созданы.
15. Добавьте операторы в класс Person для определения трех полей и метода, как показано ниже:

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace Packt.Shared
{
  public class Person
  {
    // поля
    public string Name;
    public DateTime DateOfBirth;
    public List<Person> Children = new List<Person>();

    // методы
    public void WriteToConsole()
    {
      WriteLine($"{Name} was born on a {DateOfBirth:dddd}.");
    }
  }
}
```

Упрощение методов

Сделаем так, чтобы два экземпляра человека произвели потомство. Мы можем реализовать это, написав методы. Методы экземпляра — действия, которые объект выполняет для себя; статические методы — действия, которые выполняет сам тип.

Ваш выбор зависит от того, что больше всего подходит для действия.

Имеет смысл использовать статические методы и методы экземпляров для выполнения аналогичных действий. Например, тип string имеет как статический метод Compare, так и метод экземпляра CompareTo. Это позволяет программистам, применяющим ваш тип, выбрать, как задействовать функциональность, тем самым увеличивая гибкость.

Реализация функционала с помощью методов

Начнем с реализации некоторых функций, используя методы.

1. Добавьте один метод экземпляра и один статический метод в класс `Person`, который позволит двум объектам `Person` производить потомство:

```
// статический метод «размножения»
public static Person Procreate(Person p1, Person p2)
{
  var baby = new Person
  {
    Name = $"Baby of {p1.Name} and {p2.Name}"
  };

  p1.Children.Add(baby);
  p2.Children.Add(baby);

  return baby;

}

// метод «размножения» экземпляра класса
public Person ProcreateWith(Person partner)
{
  return Procreate(this, partner);
}
```

Обратите внимание на следующие моменты:

- в статическом методе `static` с именем `Procreate` объекты `Person` для создания передаются как параметры с именами `p1` и `p2`;
- новый класс `Person` с именем `baby` создается с именем, составленным из комбинации двух человек, которые произвели потомство;
- объект `baby` добавляется в коллекцию `Children` обоих родителей, а затем возвращается. Классы — это ссылочные типы, то есть добавляется ссылка на объект `baby`, сохраненный в памяти, но не копия объекта;

- в методе экземпляра ProcreateWith объект Person, для которого производится потомство, передается в качестве параметра partner, и он вместе с this передается статическому методу Procreate для повторного использования реализации метода. Ключевое слово this ссылается на текущий экземпляр класса.

Метод, который создает новый объект или модифицирует существующий объект, должен возвращать ссылку на этот объект, чтобы вызывающая сторона могла видеть результаты.

2. В проекте PeopleApp в начале файла Program.cs импортируйте пространство имен для нашего класса и статически импортируйте тип Console, как показано ниже:

```
using System;
using Packt.Shared;
using static System.Console;
```

3. В методе Main создайте трех человек, у которых появятся дети, отметив, что для добавления символа двойной кавычки в строку необходимо поставить перед ним символ обратной косой черты с кавычкой, \":

```
var harry = new Person { Name = "Harry" };
var mary = new Person { Name = "Mary" };
var jill = new Person { Name = "Jill" };

// вызов метода экземпляра
var baby1 = mary.ProcreateWith(harry);

// вызов статического метода
var baby2 = Person.Procreate(harry, jill);

WriteLine($"{harry.Name} has {harry.Children.Count} children.");
WriteLine($"{mary.Name} has {mary.Children.Count} children.");
WriteLine($"{jill.Name} has {jill.Children.Count} children.");

WriteLine(
  format: "{0}'s first child is named \"{1}\".",
  arg0: harry.Name,
  arg1: harry.Children[0].Name);
```

4. Запустите приложение и проанализируйте результат:

```
Harry has 2 children.
Mary has 1 children.
Jill has 1 children.
Harry's first child is named "Gary".
```

Реализация функционала с помощью операций

Класс `System.String` имеет статический метод `Concat`, который объединяет два строковых значения и возвращает результат:

```
string s1 = "Hello ";
string s2 = "World!";
string s3 = string.Concat(s1, s2);
WriteLine(s3); // => Hello World!
```

Вызов метода, подобного `Concat`, работает, но программисту может быть привычнее использовать символ `+` для сложения двух строковых значений:

```
string s1 = "Hello ";
string s2 = "World!";
string s3 = s1 + s2;
WriteLine(s3); // => Hello World!
```

Хорошо известная библейская фраза «Плодитесь и размножайтесь», то есть «производите потомство». Напишем код так, чтобы символ `*` (умножение) позволял создавать два объекта `Person`.

Мы делаем это путем определения статической операции для символа *. Синтаксис скорее похож на метод, поскольку, по сути, операция — это метод, но вместо имени метода используется символ, что сокращает синтаксис.

Символ * — лишь один из многих символов, которые можно использовать в качестве операции. Полный список символов приведен на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/statements-expressions-operators/overloadable-operators.

1. В проекте `PacktLibrary` в классе `Person` создайте статическую операцию для символа *:

   ```
   // операция "размножения"
   public static Person operator *(Person p1, Person p2)
   {
     return Person.Procreate(p1, p2);
   }
   ```

В отличие от методов, операции не отображаются в списках IntelliSense для типа. Для каждой операции, которую вы определяете, также создайте метод, поскольку программисту может быть неочевидно, что операция доступна. Реализация операции может затем вызвать метод, повторно используя написанный вами код. Вторая причина предоставления метода заключается в том, что операции поддерживаются не каждым языковым компилятором.

2. В методе `Main` после вызова статического метода `Procreate` создайте еще один объект `baby` с помощью операции *:

```
// вызов статического метода
var baby2 = Person.Procreate(harry, jill);

// вызов операции
var baby3 = harry * mary;
```

3. Запустите приложение и проанализируйте результат:

```
Harry has 3 children.
Mary has 2 children.
Jill has 1 children.
Harry's first child is named "Gary".
```

Реализация функционала с помощью локальных функций

К новым возможностям языка C# 7.0 относятся *локальные функции*.

Это, по сути, перенос идеи локальных переменных на методы. Другими словами, локальными функциями являются методы, которые видимы и могут быть вызваны только в пределах метода, в котором были определены. В других языках программирования такие функции иногда называются *вложенными* или *внутренними*.

Локальные функции могут быть определены где угодно внутри метода: в начале, в конце или даже где-то посередине!

Мы воспользуемся локальной функцией для вычисления факториала.

1. В классе `Person` добавьте операторы, чтобы определить функцию `Factorial`, которая использует локальную функцию внутри себя для вычисления результата, как показано ниже:

```
// метод с локальной функцией
public int Factorial(int number)
{
  if (number < 0)
  {
    throw new ArgumentException(
      $"{nameof(number)} cannot be less than zero.");
  }
  return localFactorial(number);

  int localFactorial(int localNumber) // локальная функция
  {
    if (localNumber < 1) return 1;
```

```
    return localNumber * localFactorial(localNumber - 1);
  }
}
```

2. В проекте `Program.cs` в методе `Main` добавьте оператор для вызова функции `Factorial` и запишите возвращаемое значение в консоль:

   ```
   WriteLine($"5! is {harry.Factorial(5)}");
   ```

3. Запустите консольное приложение и проанализируйте результат:

   ```
   5! is 120
   ```

Вызов и обработка событий

Методы часто описываются как *действия, которые может выполнять объект*. К примеру, класс `List` может добавить в себя элемент или очистить сам себя, а `File` может создавать или удалять файл в файловой системе.

События часто описываются как *действия, которые происходят с объектом*. Например, в пользовательском интерфейсе `Button` есть событие `Click`, определяющее, что происходит с кнопкой при щелчке на ней. События можно охарактеризовать еще и как способ обмена сообщениями между двумя объектами.

События построены на делегатах, поэтому начнем с рассмотрения того, как работают делегаты.

Вызов методов с помощью делегатов

Вы уже знакомы с наиболее распространенным способом вызова или выполнения метода, который заключается в синтаксисе с точкой для доступа к методу по его имени. Например, метод `Console.WriteLine` дает типу `Console` задание выводить сообщения с помощью метода `WriteLine`.

Другой способ вызова или выполнения метода заключается в использовании *делегата*. Если вы использовали языки, поддерживающие *указатели на функции*, то делегат можно представить, как *типобезопасный указатель на метод*.

Другими словами, делегат — это адрес метода в памяти, имеющего ту же сигнатуру, что и делегат, чтобы его можно было безопасно вызвать.

Для примера представьте, что в классе `Person` существует метод, который должен иметь тип `string`, передаваемый как единственный параметр, и возвращать `int`:

```
public int MethodIWantToCall(string input)
{
  return input.Length; // неважно, что здесь выполняется
}
```

Я мог бы вызвать этот метод на экземпляре `Person` с именем `p1` следующим образом:

```
int answer = p1.MethodIWantToCall("Frog");
```

Как вариант, я мог бы определить делегат с совпадающей сигнатурой для косвенного вызова метода. Обратите внимание: имена параметров могут не совпадать. Должны совпадать только типы параметров и возвращаемые значения, как показано в коде ниже:

```
delegate int DelegateWithMatchingSignature(string s);
```

Теперь я могу создать экземпляр делегата, указать его на метод и, наконец, вызвать делегат (который вызывает метод!):

```
// создание экземпляра делегата, который указывает на метод
var d = new DelegateWithMatchingSignature(p1.MethodIWantToCall);

// вызов делегата, который вызывает метод
int answer2 = d("Frog");
```

Вероятно, вы задались вопросом: «А в чем смысл?» Отвечу лаконично — в гибкости.

Например, мы могли бы использовать делегаты для создания очереди методов, которые нужно вызвать по порядку. Действия в очереди, которые необходимо выполнить, распространены в сервисах для обеспечения улучшенной масштабируемости.

Другой пример — разрешить параллельное выполнение нескольких действий. Делегаты имеют встроенную поддержку асинхронных операций, которые выполняются в другом потоке, что может обеспечить улучшенную скорость отклика. Вы узнаете, как это сделать, в главе 13.

Делегаты позволяют реализовывать события для отправки сообщений между различными объектами, которые не должны знать друг о друге.

Делегаты и события — одни из самых продвинутых функций языка C#, и обучение им может занять некоторое время, поэтому не беспокойтесь, если сразу не поняли принцип их работы!

Определение и обработка делегатов

Microsoft предоставляет два предопределенных делегата для использования в качестве событий. Они выглядят следующим образом:

```
public delegate void EventHandler(
  object sender, EventArgs e);

public delegate void EventHandler<TEventArgs>(
  object sender, TEventArgs e);
```

Если хотите определить событие в собственном типе, то вам следует использовать один из этих двух предопределенных делегатов.

1. Добавьте код, показанный ниже, в класс Person и обратите внимание на следующие моменты:
   - код определяет поле делегата EventHandler с именем Shout;
   - код определяет поле int для хранения AngerLevel;
   - код определяет метод Poke;
   - каждый раз, когда человека толкают (Poke), уровень его раздражения (AngerLevel) растет. Когда уровень раздражения достигает 3, поднимается событие Shout (возмущенный возглас), но только если делегат события указывает на метод, определенный где-либо еще в коде, то есть не null.

   ```
   // событие
   public event EventHandler Shout;

   // поле
   public int AngerLevel;

   // метод
   public void Poke()
   {
     AngerLevel++;
     if (AngerLevel >= 3)
     {
       // если что-то слушает...
       if (Shout != null)
       {
         // ...то вызывается делегат
         Shout(this, EventArgs.Empty);
       }
     }
   }
   ```

   Проверка, является ли объект null до вызова одного из его методов, выполняется очень часто. Версия C# 6.0 и более поздние версии позволяют упростить проверки на null, вместив их в одну строку:

   ```
   Shout?.Invoke(this, EventArgs.Empty);
   ```

2. В Program добавьте метод с соответствующей сигнатурой, который получает ссылку на объект Person из параметра sender и выводит некоторую информацию о них:

   ```
   private static void Harry_Shout(object sender, EventArgs e)
   {
     Person p = (Person)sender;
   ```

```
  WriteLine($"{p.Name} is this angry: {p.AngerLevel}.");
}
```

ObjectName_EventName — соглашение Microsoft для имен методов, которые обрабатывают события.

3. В методе Main добавьте оператор для назначения метода полю делегата, как показано ниже:

```
harry.Shout = Harry_Shout;
```

4. Добавьте операторы для вызова метода Poke четыре раза после назначения метода для события Shout:

```
harry.Shout = Harry_Shout;
harry.Poke();
harry.Poke();
harry.Poke();
harry.Poke();
```

Делегаты — многоадресные; это значит, вы можете назначить несколько делегатов одному полю делегата. Вместо оператора присваивания = мы могли бы использовать оператор +=, чтобы добавить больше методов к тому же полю делегата. При вызове делегата вызываются все назначенные методы, хотя вы не можете контролировать порядок их вызова.

5. Запустите приложение и проанализируйте результат, как показано в следующем выводе, и обратите внимание, что Гарри первое время только злится и начинает кричать после того, как его толкнут не менее трех раз:

```
Harry is this angry: 3.
Harry is this angry: 4.
```

Определение и обработка событий

Теперь вы увидели, как делегаты реализуют наиболее важную функциональность событий: возможность определить сигнатуру для метода, который может быть реализован совершенно другим фрагментом кода, а затем вызвать этот метод и любые другие, подключенные к полю делегатов.

Но как насчет событий? Как ни странно, мы уже почти все обсудили.

При назначении метода для поля делегата не следует использовать простой оператор присваивания, как мы делали в предыдущем примере и как показано ниже:

```
harry.Shout = Harry_Shout;
```

Если поле делегата Shout уже ссылается на один или несколько методов, то назначение нового метода перекроет все остальные. С делегатами, которые используются

для событий, мы обычно хотим убедиться, что программист применяет только оператор += или оператор -= для назначения и удаления методов.

1. Для этого добавьте ключевое слово event в объявление поля делегата, как показано ниже:

```
public event EventHandler Shout;
```

2. На панели TERMINAL (Терминал) введите команду dotnet build и запишите сообщение об ошибке компилятора:

```
Program.cs(41,13): error CS0079: The event 'Person.Shout' can only appear
on the left hand side of += or -=
```

Это (почти) все, что делает ключевое слово event! Если у вас никогда не будет более одного метода, назначенного для поля делегата, то вам не нужны «события».

3. Измените присвоение метода с помощью использования оператора +=, как показано ниже:

```
harry.Shout += Harry_Shout;
```

Запустите приложение и обратите внимание, что оно работает как раньше.

Вы можете определить собственные пользовательские типы, наследуясь от EventArgs, чтобы можно было передавать дополнительную информацию в метод обработчика событий. Более подробная информация доступна на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/events/how-to-raise-and-consume-events.

Реализация интерфейсов

С помощью интерфейсов разные типы соединяются, чтобы создать новые элементы. В качестве примеров интерфейсов можно привести шпильки на детальках лего, которые позволяют им соединяться, или стандарты для электрических вилок и розеток.

Если тип реализует интерфейс, то гарантирует остальной части .NET, что поддерживает определенную функцию.

Универсальные интерфейсы

В табл. 6.1 представлено несколько универсальных интерфейсов, которые могут реализовать ваши типы.

Таблица 6.1. Универсальные интерфейсы

Интерфейс	**Метод (-ы)**	**Описание**
IComparable	CompareTo(other)	Определяет метод сравнения, который тип реализует для упорядочения или сортировки экземпляров
IComparer	Compare(first, second)	Определяет метод сравнения, который вторичный тип реализует для упорядочения или сортировки экземпляров первичного типа
IDisposable	Dispose()	Предоставляет механизм для освобождения неуправляемых ресурсов
IFormattable	ToString(format, culture)	Определяет метод, поддерживающий региональные параметры, для форматирования значения объекта в строковое представление
IFormatter	Serialize(stream, object) и Deserialize(stream)	Определяет методы преобразования объекта в поток байтов и из него для хранения или передачи
IFormat Provider	GetFormat(type)	Определяет метод для форматирования ввода на основе настроек языка и региона

Сравнение объектов при сортировке

Один из наиболее распространенных интерфейсов, который вы можете реализовать, — это `IComparable`. Он позволяет сортировать массивы и коллекции вашего типа.

1. В метод `Main` добавьте операторы, которые создают массив экземпляров `Person`, выводят массив, пытаются его сортировать и затем выводят отсортированный массив:

```
Person[] people =
{
  new Person { Name = "Simon" },
  new Person { Name = "Jenny" },
  new Person { Name = "Adam" },
  new Person { Name = "Richard" }
};

WriteLine("Initial list of people:");
foreach (var person in people)
{
  WriteLine($"{person.Name}");
}
```

```
WriteLine("Use Person's IComparable implementation to sort:");
Array.Sort(people);
foreach (var person in people)
{
  WriteLine($"{person.Name}");
}
```

2. Запустите приложение, и вы увидите следующую ошибку выполнения:

```
Unhandled Exception: System.InvalidOperationException: Failed to compare
two elements in the array. --->
System.ArgumentException: At least one object must implement IComparable.
```

   Как следует из текста ошибки, чтобы устранить проблему, наш тип должен реализовать интерфейс `IComparable`.

3. В проекте `PacktLibrary` добавьте в конец описания класса `Person` двоеточие и введите `IComparable<Person>`:

```
public class Person : IComparable<Person>
```

   Программа Visual Studio Code подчеркнет новый код, предупреждая о том, что вы еще не реализовали обещанный метод. Программа может сгенерировать каркас реализации, если вы установите указатель мыши на кнопку в виде лампочки и выберете в контекстном меню команду Implement interface (Реализовать интерфейс).

   Интерфейсы могут быть реализованы неявно и явно. Неявные реализации проще. Явные необходимы только в том случае, если тип должен иметь несколько методов с одинаковым именем и сигнатурой. Например, как `IGamePlayer`, так и `IKeyHolder` могут иметь метод `Lose` с одинаковыми параметрами. В типе, который должен реализовывать оба интерфейса, неявным методом может быть только одна реализация `Lose`. Если оба интерфейса могут использовать одну и ту же реализацию, то это работает, но в противном случае другой метод `Lose` должен быть реализован по-другому и вызываться явно.

Более подробно о явных реализациях интерфейса можно прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/interfaces/explicit-interface-implementation.

4. Прокрутите код вниз, чтобы найти сгенерированный метод, и удалите оператор, выбрасывающий ошибку `NotImplementedException`.

5. Добавьте оператор для вызова метода `CompareTo` поля `Name`, в котором используется реализация `CompareTo` для строкового типа, как показано ниже (выделено полужирным шрифтом):

```
public int CompareTo(Person other)
{
  return Name.CompareTo(other.Name);
}
```

Мы решили сравнить два экземпляра `Person`, сравнив их поля `Name`. Это значит, `Person` будут сортироваться в алфавитном порядке по имени.

Для простоты я не добавил проверки на `null` во всех этих примерах.

6. Запустите приложение. На этот раз все работает:

```
Initial list of people:
Simon
Jenny
Adam
Richard
Use Person's IComparable implementation to sort:
Adam
Jenny
Richard
Simon
```

Если кто-то захочет отсортировать массив или коллекцию экземпляров вашего типа, то реализуйте интерфейс IComparable.

Сравнение объектов с помощью отдельных классов

В некоторых случаях вы можете не иметь доступа к исходному коду типа и он может не реализовывать интерфейс `IComparable`. К счастью, есть еще один способ сортировать экземпляры типа: создать вторичный тип, реализующий несколько иной интерфейс под названием `IComparer`.

1. В проекте `PacktLibrary` добавьте класс `PersonComparer`, реализующий интерфейс `IComparer`, который сравнивает двух людей, то есть два экземпляра `Person`, сравнивая длину их поля `Name`, или, если имена имеют одинаковую длину, сравнивая имена в алфавитном порядке:

```
using System.Collections.Generic;

namespace Packt.Shared
{
  public class PersonComparer : IComparer<Person>
  {
    public int Compare(Person x, Person y)
    {
```

```
        // сравнение длины имени...
        int result = x.Name.Length
          .CompareTo(y.Name.Length);

        // ...если равны...
        if (result == 0)
        {
          // ...затем сравниваем по именам...
          return x.Name.CompareTo(y.Name);
        }
        else
        {
          // ...в противном случае сравниваем по длине
          return result;
        }
      }
    }
  }
```

2. В приложении `PeopleApp` в классе `Program` в методе `Main` добавьте операторы для сортировки массива, используя эту альтернативную реализацию, как показано ниже:

```
WriteLine("Use PersonComparer's IComparer implementation to sort:");
Array.Sort(people, new PersonComparer());
foreach (var person in people)
{
  WriteLine($"{person.Name}");
}
```

3. Запустите приложение и проанализируйте результат:

```
Initial list of people:
Simon
Jenny
Adam
Richard
Use Person's IComparable implementation to sort:
Adam
Jenny
Richard
Simon
Use PersonComparer's IComparer implementation to sort:
Adam
Jenny
Simon
Richard
```

На сей раз, сортируя массив `people`, мы явно просим алгоритм сортировки использовать тип `PersonComparer`, поэтому сначала люди сортируются по длине имен, а если длины двух или более имен равны, то по алфавиту.

Определение интерфейсов с реализациями по умолчанию

Функция языка, представленная в C# 8.0, — *реализация интерфейса по умолчанию*.

1. В проекте `PacktLibrary` создайте файл `IPlayable.cs`. Если у вас установлено расширение *C# Extensions*, то можете щелкнуть правой кнопкой мыши на папке PacktLibrary и выбрать пункт New C# Interface (Новый интерфейс C#) в контекстном меню.
2. Измените код, чтобы определить общедоступный интерфейс `IPlayable` с двумя методами `Play` и `Pause`:

```
using static System.Console;

namespace Packt.Shared
{
  public interface IPlayable
  {
    void Play();

    void Pause();
  }
}
```

3. В проекте `PacktLibrary` создайте файл `DvdPlayer.cs`.
4. Измените операторы в файле для реализации интерфейса `IPlayable`:

```
using static System.Console;

namespace Packt.Shared
{
  public class DvdPlayer : IPlayable
  {
    public void Pause()
    {
      WriteLine("DVD player is pausing.");
    }

    public void Play()
    {
      WriteLine("DVD player is playing.");
    }
  }

}
```

Представленное выше полезно, но что, если мы решим добавить третий метод, `Stop`? До выхода версии C# 8.0 это было бы невозможно в случае, если хотя бы

один тип реализует исходный интерфейс. Одно из основных свойств интерфейса — это то, что он предоставляет фиксированный контракт.

Версия C# 8.0 позволяет добавлять в интерфейс новые члены после его выпуска, если они имеют реализацию по умолчанию. Пуристам C# не нравится эта идея, но по практическим соображениям она полезна, и другие языки, такие как Java и Swift, поддерживают аналогичные приемы.

Прочитать об архитектурных решениях, использующих реализации интерфейса по умолчанию, можно на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods.

Поддержка реализаций интерфейса по умолчанию требует некоторых фундаментальных изменений в базовой платформе, поэтому они поддерживаются только в версии C#, если целевая платформа —.NET 5 или более поздние версии, .NET Core 3.0 или .NET Standard 2.1, и поэтому эта функциональность не поддерживается в .NET Framework.

5. Измените интерфейс `IPlayable`, добавив метод `Stop` с реализацией по умолчанию:

```
using static System.Console;

namespace Packt.Shared
{
  public interface IPlayable
  {
    void Play();
    void Pause();

    void Stop() // реализация интерфейса по умолчанию
    {
      WriteLine("Default implementation of Stop.");
    }
  }
}
```

6. На панели `TERMINAL` (Терминал) скомпилируйте проект `PeopleApp`, введя команду `dotnet build`, и обратите внимание, что компиляция успешно выполнена.

Прочитать руководство по обновлению интерфейсов с помощью членов интерфейса по умолчанию можно на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/tutorials/default-interface-members-versions.

Обеспечение безопасности многократного использования типов с помощью дженериков

В 2005 году в версии C# 2.0 и .NET Framework 2.0 корпорация Microsoft представила функцию под названием *«дженерики»* (или обобщения), которая позволяет вашим типам быть более безопасными для повторного использования и более эффективными. Это позволяет программисту передавать типы в качестве параметров, аналогично тому как вы можете передавать объекты в качестве параметров.

Сначала рассмотрим пример не дженерика, чтобы вы могли понять проблему, для решения которой предназначены дженерики.

1. В проекте `PacktLibrary` создайте класс `Thing`, как показано в коде ниже, и обратите внимание на следующие моменты:
   - у класса `Thing` есть поле `object` с именем `Data`;
   - у класса `Thing` есть метод `Process`, который принимает входной параметр `object` и возвращает значение `string`.

```
using System;

namespace Packt.Shared
{
  public class Thing
  {
    public object Data = default(object);

    public string Process(object input)
    {
      if (Data == input)
      {
        return "Data and input are the same.";
      }
      else
      {
        return "Data and input are NOT the same.";
      }
    }
  }
}
```

2. В проекте `PeopleApp` добавьте несколько операторов в конец метода `Main`, как показано ниже:

```
var t1 = new Thing();
t1.Data = 42;
WriteLine($"Thing with an integer: {t1.Process(42)}");

var t2 = new Thing();
t2.Data = "apple";
WriteLine($"Thing with a string: {t2.Process("apple")}");
```

3. Запустите приложение и проанализируйте результат, как показано в следующем выводе:

```
Thing with an integer: Data and input are NOT the same.
Thing with a string: Data and input are the same.
```

В настоящее время класс `Thing` достаточно гибок, поскольку для поля `Data` и параметра `input` может быть установлен любой тип. Но здесь нет проверки типов, поэтому внутри метода `Process` мы не можем безопасно выполнять большое количество кода, так как результаты иногда бывают неправильными. Например, при передаче значений `int` в параметр `object`!

Это связано с тем, что значение `42`, хранящееся в свойстве `Data`, хранится в месте памяти, отличном от места хранения значения `42`, переданного в качестве параметра, и при сравнении ссылочных типов они равны, только если хранятся по одному и тому же адресу в памяти, то есть они — это один и тот же объект, даже если их значения равны. Эту проблему мы можем решить, используя дженерики.

Работа с типами-дженериками

Мы можем решить эту проблему, используя дженерики.

1. В проекте `PacktLibrary` создайте класс `GenericThing`:

```
using System;

namespace Packt.Shared
{
  public class GenericThing<T> where T : IComparable
  {
    public T Data = default(T);

    public string Process(T input)
    {
      if (Data.CompareTo(input) == 0)
      {
        return "Data and input are the same.";
      }
      else
      {
```

```
            return "Data and input are NOT the same.";
          }
        }
      }
    }
```

Обратите внимание на следующие моменты:

- `GenericThing` имеет параметр типа-дженерика с именем `T`, который может быть любого типа, реализующего `IComparable`, поэтому должен иметь метод `CompareTo`, возвращающий в результате `0`, если два объекта равны. Существует конвенция о том, что имя параметра типа должно быть `T` при условии, что существует только один такой параметр;
- `GenericThing` имеет поле типа `T` с именем `Data`;
- `GenericThing` имеет метод `Process`, который принимает входной параметр типа `T` и возвращает строковое значение.

2. В проекте `PeopleApp` добавьте несколько операторов в конец метода `Main`, как показано ниже:

```
var gt1 = new GenericThing<int>();
gt1.Data = 42;
WriteLine($"GenericThing with an integer:
{gt1.Process(42)}");

var gt2 = new GenericThing<string>();
gt2.Data = "apple";
WriteLine($"GenericThing with a string:
{gt2.Process("apple")}");
```

Обратите внимание на следующие моменты:

- при создании экземпляра типа-дженерика разработчик должен передать параметр типа. В данном примере мы передаем `int` как параметр типа для `gt1`, а `string` — как параметр типа для `gt2`. Поэтому, где бы `T` ни появлялся в классе `GenericThing`, он заменяется на `int` и `string`;
- при установке поля `Data` и передаче входного параметра компилятор принудительно использует значение `int`, например `42`, для переменной `gt1` и строковое значение, например `"apple"`, для переменной `gt2`.

3. Запустите приложение, проанализируйте результат и обратите внимание, что логика метода `Process` правильно работает для `GenericThing` как для `int`, так и для строковых значений:

```
Thing with an integer: Data and input are NOT the same.
Thing with a string: Data and input are the same.
GenericThing with an integer: Data and input are the same.
GenericThing with a string: Data and input are the same.
```

Работа с методами-дженериками

Дженериками могут быть не только типы, но и методы даже внутри типа, который не является дженериком.

1. В методе `PacktLibrary` создайте класс `Squarer` с помощью метода-дженерика `Square`:

```
using System;
using System.Threading;

namespace Packt.Shared
{
  public static class Squarer
  {
    public static double Square<T>(T input)
      where T : IConvertible
    {
      // конвертирует, используя текущие настройки
      double d = input.ToDouble(
        Thread.CurrentThread.CurrentCulture);

      return d * d;
    }
  }
}
```

   Обратите внимание на следующие моменты:

   - класс `Squarer` не является дженериком;
   - метод `Square` — дженерик, и его параметр типа `T` должен реализовывать параметр `IConvertible`, поэтому компилятор убедится в наличии метода `ToDouble`;
   - `T` используется в качестве типа для параметра `input`;
   - метод `ToDouble` требует параметр, который реализует параметр `IFormatProvider`, чтобы понять формат чисел для региональных настроек. Мы можем передать свойство `CurrentCulture` текущего потока, чтобы указать язык и регион для вашего компьютера. О межрегиональных настройках вы узнаете в главе 8;
   - возвращаемое значение — это параметр `input`, умноженный на себя, то есть в квадрате.

2. В приложении `PeopleApp` в классе `Program` в конце метода `Main` добавьте следующий код. Обратите внимание: при вызове метода-дженерика вы можете указать параметр типа, чтобы сделать его более понятным, как показано в первом примере, хотя компилятор может выполнить его самостоятельно, как показано во втором примере:

```
string number1 = "4";
WriteLine("{0} squared is {1}",
  arg0: number1,
  arg1: Squarer.Square<string>(number1));

byte number2 = 3;
WriteLine("{0} squared is {1}",
  arg0: number2,
  arg1: Squarer.Square(number2));
```

3. Запустите приложение и проанализируйте результат, как показано в следующем выводе:

```
4 squared is 16
3 squared is 9
```

Я уже упоминал ссылочные типы. Давайте рассмотрим их подробнее.

Управление памятью с помощью ссылочных типов и типов значений

Существует две категории памяти: *стек* и *куча*. В современных операционных системах оба этих объекта могут находиться где угодно в физической или виртуальной памяти.

Стек работает быстрее (поскольку управляется непосредственно процессором и использует механизм «первым вошел — первым вышел», а значит, его данные с большей вероятностью будут храниться в кэш-памяти `L1` или `L2`), но его размер ограничен, в то время как куча более медленная, но увеличивается динамически. Например, на моем macOS на панели TERMINAL (Терминал) я могу ввести команду: `ulimit -a` и увидеть, что размер стека ограничен 8192 Кбайт, а другая память «не ограничена». Вот почему так легко получить «переполнение стека».

Доступны два ключевых слова C#, которые пригодны для создания объектов: `class` и `struct`. Оба могут иметь одинаковые члены (поля и методы, например). Разница между ними заключается в распределении памяти.

При использовании класса вы определяете *ссылочный тип*. Это значит, что память для самого объекта выделяется в куче и в стеке хранится только адрес объекта в памяти (и некоторые служебные данные).

Более подробно о внутренней структуре памяти для типов в .NET можно прочитать на сайте https://adamsitnik.com/Value-Types-vs-Reference-Types/.

Если вы определяете тип с помощью `struct` (то есть определяете структуру), то определяете *тип значения*. Это значит, что память для самого объекта выделяется в стеке.

Если в `struct` для полей используются типы, которые сами не являются структурами, то эти поля будут храниться в куче, то есть данные для этого объекта хранятся как в стеке, так и в куче!

К наиболее распространенным структурам относятся следующие:

- *числа* — `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` и `decimal`;
- *разное* — `char` и `bool`;
- *System.Drawing* — `Color`, `Point` и `Rectangle`.

Почти все остальные типы являются классами, включая `string`.

Помимо разницы в том, где в памяти хранятся данные типа, другое важное отличие состоит в том, что вы не можете наследовать от структуры.

Работа со структурами

Рассмотрим работу с типами значений.

1. Создайте файл `DisplacementVector.cs` в проекте `PacktLibrary`.
2. Измените файл, как показано ниже, и обратите внимание на следующие моменты:
   - тип объявляется с помощью ключевого слова `struct` вместо `class`;
   - используются два поля `int`, названные `X` и `Y`;
   - применяется конструктор для установки начальных значений для `X` и `Y`;
   - используется операция для сложения двух экземпляров, которая возвращает новый экземпляр типа с `X`, сложенным с `X`, и `Y`, сложенным с `Y`.

```
namespace Packt.Shared
{
  public struct DisplacementVector
  {
    public int X;
    public int Y;

    public DisplacementVector(int initialX, int initialY)
    {
      X = initialX;
      Y = initialY;
    }
```

```
    public static DisplacementVector operator +(
      DisplacementVector vector1,
      DisplacementVector vector2)
    {
      return new DisplacementVector(
        vector1.X + vector2.X,
        vector1.Y + vector2.Y);
    }
  }
}
```

3. В проекте `PeopleApp` в классе `Program` в методе `Main` добавьте операторы, чтобы создать два новых экземпляра `DisplacementVector`, сложите их и выведите результат:

```
var dv1 = new DisplacementVector(3, 5);
var dv2 = new DisplacementVector(-2, 7);
var dv3 = dv1 + dv2;

WriteLine($"({dv1.X}, {dv1.Y}) + ({dv2.X}, {dv2.Y}) =
({dv3.X}, {dv3.Y})");
```

4. Запустите приложение и проанализируйте результат:

```
(3, 5) + (-2, 7) = (1, 12)
```

Если все поля вашего типа используют не более 16 байт стековой памяти, в нем в качестве полей применяются только структуры и вы не планируете наследоваться от своего типа, то сотрудники корпорации Microsoft рекомендуют задействовать тип struct. Если ваш тип использует более 16 байт стековой памяти, или в нем в качестве полей применяются классы, или если вы планируете наследовать его, то задействуйте тип class.

Освобождение неуправляемых ресурсов

В предыдущей главе вы узнали, что конструкторы могут использоваться для инициализации полей и тип может иметь несколько конструкторов. Представьте, что конструктор выделяет неуправляемый ресурс, то есть нечто неподконтрольное .NET. Неуправляемый ресурс должен освобождаться вручную, поскольку платформа .NET не способна сделать это автоматически.

В рамках этой темы я продемонстрирую несколько примеров кода, которые не нужно создавать в вашем текущем проекте.

Более подробно эту тему можно изучить на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/garbage-collection/

Каждый тип может иметь один *метод завершения (финализатор)*, который будет вызываться общеязыковой исполняющей средой (CLR) при возникновении необходимости освободить ресурсы. Финализатор имеет то же имя, что и конструктор, то есть имя типа, но с префиксом в виде символа тильды (~), как показано в следующем примере:

```
public class Animal
{
  public Animal()
  {
    // выделение неуправляемого ресурса
  }

  ~Animal() // финализатор (метод завершения)
  {
    // освобождение неуправляемого ресурса
  }
}
```

Не путайте *финализатор (метод завершения, также иногда называемый деструктором)* с *деконструктором*. Первый освобождает ресурсы, то есть уничтожает объект. Второй возвращает объект, разбитый на составные части, и использует синтаксис C# для деконструкции, например для работы с кортежами.

Предыдущий пример кода — это минимум, который вы должны делать при работе с неуправляемыми ресурсами. Проблема с реализацией только финализатора такова: сборщику мусора .NET потребуется две сборки мусора, чтобы полностью освободить выделенные ресурсы для данного типа.

Хоть это и не обязательно, рекомендуется также предоставить метод, позволяющий разработчику, использующему ваш тип, явно освобождать ресурсы, чтобы сборщик мусора мог немедленно и детерминированно освободить неуправляемый ресурс, такой как файл, а затем освободить управляемую часть памяти объекта за одну сборку.

Для этого существует стандартный механизм — реализация интерфейса `IDisposable`, как показано в следующем примере:

```
public class Animal : IDisposable
{
  public Animal()
  {
    // выделение неуправляемого ресурса
  }

  ~Animal() // финализатор
  {
```

```
    if (disposed) return;
    Dispose(false);
  }

  bool disposed = false; // освобождены ли ресурсы?

  public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }

  protected virtual void Dispose(bool disposing)
  {
    if (disposed) return;

    // освобождение *неуправляемого* ресурса
    // ...

    if (disposing)
    {
      // освобождение любых других *управляемых* ресурсов
      // ...
    }

    disposed = true;
  }
}
```

Здесь реализовано два метода `Dispose` — `public` и `private`.

- Открытый метод (`public`) должен вызываться разработчиком, использующим ваш тип. При вызове открытого метода `Dispose` необходимо освободить как неуправляемые, так и управляемые ресурсы.
- Защищенный метод `Dispose` (с параметром `bool`) используется для реализации удаления ресурсов. Необходимо проверить параметр `disposing` и флаг `disposed`, поскольку если финализатор уже запущен и он вызвал метод `~Animal`, необходимо освободить только неуправляемые ресурсы.

Кроме того, обратите внимание на вызов метода `GC.SuppressFinalize(this)` — он уведомляет сборщик мусора о том, что больше не нужно запускать финализатор, и устраняет необходимость во второй сборке.

Более подробно информацию, касающуюся финализаторов и освобождения ресурсов, можно изучить на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/garbage-collection/unmanaged.

Обеспечение вызова метода Dispose

Когда используется тип, реализующий интерфейс `IDisposable`, гарантировать вызов открытого метода `Dispose` можно, применив оператор `using`, как показано в коде, приведенном ниже:

```
using(Animal a = new Animal())
{
  // код с экземпляром Animal
}
```

Компилятор преобразует ваш код в нечто подобное показанному ниже, гарантируя, что, даже если будет вызвано исключение, метод `Dispose` все равно будет вызван:

```
Animal a = new Animal();
try
{
  // код с экземпляром Animal
}
finally
{
  if (a != null) a.Dispose();
}
```

Более подробно информацию, касающуюся интерфейса IDisposable, можно изучить на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/garbage-collection/using-objects.

Практические примеры освобождения неуправляемых ресурсов с помощью `IDisposable`, операторов `using` и блоков `try ... finally` вы увидите в главе 9.

Наследование классов

Тип `Person`, созданный нами ранее, неявно произведен (унаследован) от типа `System.Object`. Теперь мы создадим новый класс, наследуемый от класса `Person`.

1. Создайте новый класс `Employee` в проекте `PacktLibrary`.
2. Измените его операторы:

```
using System;

namespace Packt.Shared
{
```

```
    public class Employee : Person
    {
    }
}
```

3. Добавьте следующие операторы в метод `Main`, чтобы создать экземпляр класса `Employee`:

```
Employee john = new Employee
{
  Name = "John Jones",
  DateOfBirth = new DateTime(1990, 7, 28)
};
john.WriteToConsole();
```

4. Запустите консольное приложение и проанализируйте результат:

```
John Jones was born on a Saturday
```

Обратите внимание, что класс `Employee` унаследовал все члены класса `Person`.

Расширение классов

Теперь мы добавим несколько членов, относящихся только к сотрудникам (`Employee`), тем самым расширив класс.

1. В класс `Employee` добавьте код, показанный ниже, чтобы определить два свойства:

```
public string EmployeeCode { get; set; }
public DateTime HireDate { get; set; }
```

2. Вернитесь к методу `Main` и добавьте операторы, чтобы установить код сотрудника Джона и дату найма:

```
john.EmployeeCode = "JJ001";
john.HireDate = new DateTime(2014, 11, 23);
WriteLine($"{john.Name} was hired on
{john.HireDate:dd/MM/yy}");
```

3. Запустите консольное приложение и проанализируйте результат:

```
John Jones was hired on 23/11/14
```

Сокрытие членов класса

До сих пор метод `WriteToConsole` наследовался от класса `Person` и выводил только имя сотрудника и дату его рождения. Вам может понадобиться изменить поведение этого метода в отношении сотрудника.

1. В класс `Employee` добавьте код, показанный ниже и выделенный жирным шрифтом, чтобы переопределить метод `WriteToConsole`:

```
using System;
using static System.Console;

namespace Packt.Shared
{
  public class Employee : Person
  {
    public string EmployeeCode { get; set; }

    public DateTime HireDate { get; set; }

    public void WriteToConsole()
    {
      WriteLine(format:
        "{0} was born on {1:dd/MM/yy} and hired on {2:dd/MM/yy}",
        arg0: Name,
        arg1: DateOfBirth,
        arg2: HireDate));
    }
  }
}
```

2. Запустите приложение и проанализируйте результат:

```
John Jones was born on 28/07/90 and hired on 01/01/01
John Jones was hired on 23/11/14
```

Программа Visual Studio Code предупредит вас о том, что ваш метод скрывает одноименный метод, добавив зеленую волнистую линию под именем вашего метода. Панель `PROBLEMS` (Проблемы) содержит больше подробной информации, а компилятор выдаст предупреждение при сборке и запуске консольного приложения (рис. 6.1).

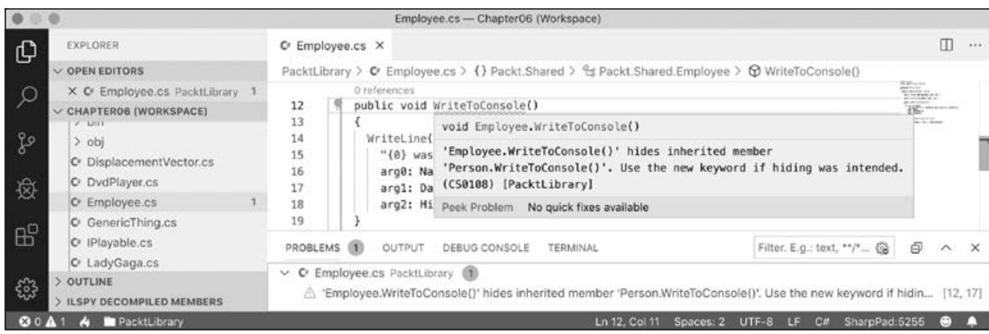


Рис. 6.1. Сообщение о сокрытии метода

Избавиться от предупреждения можно, добавив в метод ключевое слово `new`, указывающее на то, что вы намеренно замещаете старый метод, как показано в коде, приведенном ниже:

```
public new void WriteToConsole()
```

Переопределение членов

Вместо того чтобы скрывать метод, обычно лучше *переопределить* его. Вы можете переопределять члены только в том случае, если базовый класс допускает это, с помощью ключевого слова `virtual`.

1. В метод `Main` добавьте оператор, позволяющий записать значение переменной `john` в консоль в виде строки:

   ```
   WriteLine(john.ToString());
   ```

2. Запустите приложение и обратите внимание, что метод `ToString` наследуется от типа `System.Object`, поэтому реализация выводит пространство имен и имя типа:

   ```
   Packt.Shared.Employee
   ```

3. Переопределите это поведение для класса `Person`, добавив метод `ToString` для вывода имени человека, а также имени типа:

   ```
   // переопределенные методы
   public override string ToString()
   {
     return $"{Name} is a {base.ToString()}";
   }
   ```

   Ключевое слово `base` позволяет подклассу получать доступ к членам своего суперкласса; то есть базового класса, от которого он наследуется.

4. Запустите консольное приложение и проанализируйте результат. Теперь, когда вызывается метод `ToString`, он выводит имя человека, а также реализацию базового класса `ToString`, как показано в выводе ниже:

   ```
   John Jones is a Packt.Shared.Employee
   ```

Многие существующие API, например Microsoft Entity Framework Core, Castle's DynamicProxy и модели контента Episerver, требуют, чтобы свойства, определяемые программистами в своих классах, определялись как виртуальные. Если у вас нет на иное веских оснований, то определяйте члены своих методов и свойств именно так.

Предотвращение наследования и переопределения

Вы можете предотвратить наследование своего класса, указав в его определении ключевое слово sealed (запечатанный). Никто не сможет наследоваться от запечатанного класса ScroogeMcDuck:

```
public sealed class ScroogeMcDuck
{
}
```

Примером запечатанного класса может служить string. Корпорация Microsoft внедрила в данный класс некоторые критические оптимизации, на которые наследование может повлиять негативным образом, и поэтому запечатала его.

Вы можете предотвратить переопределение виртуального метода в своем классе, указав имя метода с ключевым словом sealed.

```
using static System.Console;

namespace Packt.Shared
{
  public class Singer
  {
    // virtual позволяет переопределить метод
    public virtual void Sing()
    {
      WriteLine("Singing...");
    }
  }

  public class LadyGaga : Singer
  {
    // sealed предотвращает переопределение метода в подклассах
    public sealed override void Sing()
    {
      WriteLine("Singing with style...");
    }
  }
}
```

Вы можете запечатать только переопределенный метод.

Полиморфизм

Теперь вы знаете два способа изменения поведения унаследованного метода. Мы можем скрыть его с помощью ключевого слова new *(неполиморфное наследование)* или переопределить *(полиморфное наследование)*.

Оба способа могут вызывать базовый класс с помощью ключевого слова base. Так в чем же разница?

Все зависит от типа переменной, содержащей ссылку на объект. Например, переменная типа Person может содержать ссылку на класс Person или любой тип, производный от класса Person.

1. В классе Employee добавьте операторы для переопределения метода ToString, чтобы он записывал имя и код сотрудника в консоль:

```
public override string ToString()
{
  return $"{Name}'s code is {EmployeeCode}";
}
```

2. В методе Main добавьте операторы для создания сотрудника Alice, сохраните их в переменной типа Person и вызовите методы WriteToConsole и ToString для обеих переменных:

```
Employee aliceInEmployee = new Employee
  { Name = "Alice", EmployeeCode = "AA123" };

Person aliceInPerson = aliceInEmployee;

aliceInEmployee.WriteToConsole();
aliceInPerson.WriteToConsole();

WriteLine(aliceInEmployee.ToString());

WriteLine(aliceInPerson.ToString());
```

3. Запустите консольное приложение и проанализируйте результат:

```
Alice was born on 01/01/01 and hired on 01/01/01
Alice was born on a Monday
Alice's code is AA123
Alice's code is AA123
```

Обратите внимание: когда метод скрывается с помощью ключевого слова new, компилятор «недостаточно умен», чтобы знать, что объект — сотрудник (Employee), поэтому вызывает метод WriteToConsole класса Person.

Если метод переопределяется с помощью ключевых слов virtual и override, то компилятор «понимает», что хоть переменная и объявлена как класс Person, сам объект — это Employee, и потому вызывается реализация Employee метода ToString.

Модификаторы доступа и то, как они влияют на работу кода, приведены в табл. 6.2.

Таблица 6.2. Модификаторы доступа

Тип переменной	Модификатор доступа	Выполненный метод	В классе
Person	—	WriteToConsole	Person
Employee	new	WriteToConsole	Employee
Person	virtual	ToString	Employee
Employee	override	ToString	Employee

Большинству программистов парадигма полиморфизма с практической точки зрения кажется малоперспективной. Если вы освоите данную концепцию — прекрасно; а если нет — не волнуйтесь. Некоторым программистам нравится объявлять себя всезнайками, говоря, что понимание полиморфизма важно, хотя, на мой взгляд, это не так. Вы можете построить блестящую карьеру программиста на C#, будучи неспособным объяснить концепцию полиморфизма точно так же, как успешный автогонщик не знает подробностей работы системы впрыска.

Для изменения реализации унаследованного метода, когда это возможно, следует использовать модификатор virtual или override вместо new.

Приведение в иерархиях наследования

Приведение типов несколько отличается от преобразования типов.

Неявное приведение

В предыдущем примере показано, как экземпляр производного типа может быть сохранен в переменной базового типа (или базового базового типа и т. д.). Этот процесс называется *неявным приведением*.

Явное приведение

Можно пойти другим путем и использовать явное приведение, указав в коде круглые скобки.

1. В методе `Main` добавьте оператор для назначения переменной `aliceInPerson` новой переменной `Employee`:

   ```
   Employee explicitAlice = aliceInPerson;
   ```

2. Программа Visual Studio Code отображает красный знак и ошибку компиляции (рис. 6.2).

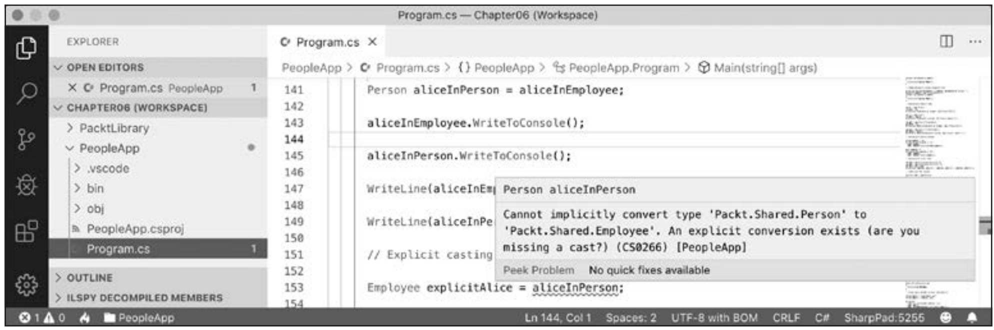


Рис. 6.2. Отсутствует явная ошибка компиляции приведения

3. Измените оператор, добававив перед именем присваиваемой переменной явное приведение к типу `Employee`:

```
Employee explicitAlice = (Employee)aliceInPerson;
```

Обработка исключений приведения

Компилятор теперь не выдает ошибок; *но*, поскольку `aliceInPerson` может быть другим производным типом, например `Student`, а не `Employee`, нужно соблюдать осторожность. В реальном приложении с более сложным кодом текущее значение этой переменной могло быть установлено на экземпляр `Student`, и тогда данный оператор может вызвать ошибку `InvalidCastException`.

Справиться с ней можно, добавив оператор `try`, но существует способ получше: проверить текущий тип объекта с помощью ключевого слова `is`.

1. Оберните оператор явного приведения оператором `if` следующим образом:

```
if (aliceInPerson is Employee)
{
  WriteLine($"{nameof(aliceInPerson)} IS an Employee");
  Employee explicitAlice = (Employee)aliceInPerson;
  // действия с explicitAlice
}
```

2. Запустите консольное приложение и проанализируйте результат:

```
aliceInPerson IS an Employee
```

Кроме того, для приведения вы можете использовать ключевое слово `as`. Вместо вызова исключения ключевое слово `as` возвращает `null`, если тип не может быть приведен.

3. Добавьте следующие операторы в конец метода `Main`:

```
Employee aliceAsEmployee = aliceInPerson as Employee;

if (aliceAsEmployee != null)
{
  WriteLine($"{nameof(aliceInPerson)} AS an Employee");
  // действия с aliceAsEmployee
}
```

Поскольку доступ к переменной со значением `null` может вызвать ошибку `NullReferenceException`, вы должны всегда проверять значение на `null` перед использованием результата.

4. Запустите консольное приложение и проанализируйте результат:

```
aliceInPerson AS an Employee
```

Что, если нужно выполнить блок операторов, а `Alice` не является сотрудником?

Раньше вам приходилось использовать оператор `!`, как показано в следующем коде:

```
if (!(aliceInPerson is Employee))
```

В C# 9 и более поздних версиях вы можете использовать ключевое слово `not`, как показано в следующем коде:

```
if (aliceInPerson is not Employee)
```

Используйте ключевые слова is и as, чтобы избежать вызова исключений при приведении производных типов. Если вы этого не сделаете, то должны написать операторы try ... catch для InvalidCastException.

Наследование и расширение типов .NET

Платформа .NET включает готовые библиотеки классов, содержащие сотни тысяч типов. Вместо того чтобы создавать собственные, совершенно новые типы, зачастую достаточно наследовать один из предустановленных типов корпорации Microsoft.

Наследование исключений

В качестве примера наследования мы выведем новый тип исключения.

1. В проекте `PacktLibrary` создайте класс `PersonException` с тремя конструкторами, как показано ниже:

```
using System;

namespace Packt.Shared
{
  public class PersonException : Exception
  {
    public PersonException() : base() { }

    public PersonException(string message) : base(message) { }

    public PersonException(
      string message, Exception innerException)
      : base(message, innerException) { }
  }
}
```

В отличие от обычных методов, конструкторы не наследуются, вследствие чего мы должны явно объявить и явно вызвать реализации конструктора `base` в `System.Exception`, чтобы сделать их доступными для программистов, которые могут захотеть применить эти конструкторы с нашим пользовательским исключением.

2. В классе `Person` добавьте операторы, чтобы определить метод, который выдает исключение, если параметр даты/времени меньше даты рождения человека, как показано ниже:

```
public void TimeTravel(DateTime when)
{
  if (when <= DateOfBirth)
  {
    throw new PersonException("If you travel back in time to a date earlier
    than your own birth then the universe will explode!");
  }
  else
  {
    WriteLine($"Welcome to {when:yyyy}!");
  }
}
```

3. В метод `Main` добавьте операторы для определения того, что произойдет, если Джон Джонс переместится в машине времени слишком далеко:

```
try
{
```

```
    john.TimeTravel(new DateTime(1999, 12, 31));
    john.TimeTravel(new DateTime(1950, 12, 25));
  }
  catch (PersonException ex)
  {
    WriteLine(ex.Message);
  }
```

4. Запустите консольное приложение и проанализируйте результат:

```
Welcome to 1999!
If you travel back in time to a date earlier than your own birth then
the universe will explode!
```

При определении собственных исключений передавайте им те же три конструктора.

Расширение типов при невозможности наследования

Ранее вы узнали, как можно использовать модификатор `sealed` для предотвращения наследования.

Корпорация Microsoft применила ключевое слово `sealed` к классу `System.String`, чтобы никто не смог наследовать его и потенциально нарушить поведение строк.

Можем ли мы добавить в него новые методы? Да, если воспользуемся *методами расширения*, впервые появившимися в версии C# 3.0.

Применение статических методов для многократного использования

Начиная с первой версии языка C#, мы можем создавать статические методы для многократного использования, например возможность проверки того, содержит ли строка адрес электронной почты. Эта реализация будет применять регулярное выражение, больше о котором вы узнаете в главе 8.

1. В проекте `PacktLibrary` создайте класс `StringExtensions`, как показано ниже, и обратите внимание на следующие моменты:
   - класс импортирует пространство имен для обработки регулярных выражений;
   - статический метод `IsValidEmail` использует тип `Regex` для проверки совпадений с простым шаблоном электронной почты, который ищет допустимые символы до и после символа `@`.

```
using System.Text.RegularExpressions;

namespace Packt.Shared
```

```
{
  public class StringExtensions
  {
    public static bool IsValidEmail(string input)
    {
      // используйте простое регулярное выражение
      // для проверки того, что входная строка — реальный
      // адрес электронной почты
      return Regex.IsMatch(input,
      @"[a-zA-Z0-9\.-_]+@[a-zA-Z0-9\.-_]+");
    }
  }
}
```

2. Добавьте следующие операторы в нижней части метода `Main` для проверки двух примеров адресов электронной почты:

```
string email1 = "pamela@test.com";
string email2 = "ian&test.com";

WriteLine(
  "{0} is a valid e-mail address: {1}",
  arg0: email1,
  arg1: StringExtensions.IsValidEmail(email1));

WriteLine(
  "{0} is a valid e-mail address: {1}",
  arg0: email2,
  arg1: StringExtensions.IsValidEmail(email2));
```

3. Запустите приложение и проанализируйте результат:

```
pamela@test.com is a valid e-mail address: True
ian&test.com is a valid e-mail address: False
```

Способ работает, но методы расширения могут уменьшить объем кода, который требуется набирать, и упростить использование этой функции.

Применение методов расширения для многократного использования

Методы `static` легко превратить в методы расширения для их использования.

1. В классе `StringExtensions` перед его именем добавьте модификатор `static`, а перед типом `string` — модификатор `this`, как показано ниже (выделено полужирным шрифтом):

```
public static class StringExtensions
{
  public static bool IsValidEmail(this string input)
  {
```

Эти две поправки сообщают компилятору, что он должен рассматривать код как метод, расширяющий тип `string`.

2. Вернитесь к классу `Program` и добавьте несколько дополнительных операторов, чтобы использовать метод в качестве метода расширения для значений типа `string`:

```
WriteLine(
  "{0} is a valid e-mail address: {1}",
  arg0: email1,
  arg1: email1.IsValidEmail());

WriteLine(
  "{0} is a valid e-mail address: {1}",
  arg0: email2,
  arg1: email2.IsValidEmail());
```

Обратите внимание на небольшое изменение в синтаксисе. При этом устаревший и длинный синтаксис все еще работает.

3. Метод расширения `IsValidEmail` теперь выглядит как метод экземпляра, аналогично всем действительным методам экземпляра типа `string`, например `IsNormalized` и `Insert` (рис. 6.3).

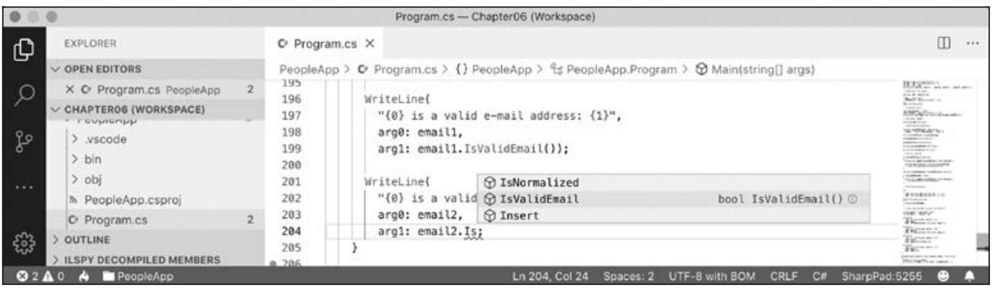


Рис. 6.3. Применение методов расширения

Методы расширения не могут заменить или переопределить используемые методы экземпляра, поэтому вы не можете, к примеру, переопределить метод `Insert` переменной `string`. Метод расширения будет проявляться как перегрузка в IntelliSense, но метод экземпляра будет вызван вместо метода расширения с тем же именем и сигнатурой.

Хотя кажется, будто методы расширения не дают большого преимущества по сравнению со статическими, в главе 12 вы увидите некоторые чрезвычайно эффективные способы использования методов расширения.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 6.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Что такое делегат?
2. Что такое событие?
3. Как связаны базовый и производный классы и как производный класс может получить доступ к базовому классу?
4. В чем разница между ключевыми словами `is` и `as`?
5. Какое ключевое слово используется для предотвращения наследования класса и переопределения метода?
6. Какое ключевое слово применяется для предотвращения создания экземпляра класса с помощью ключевого слова `new`?
7. Какое ключевое слово служит для переопределения члена?
8. Чем деструктор отличается от деконструктора?
9. Как выглядят сигнатуры конструкторов, которые должны иметь все исключения?
10. Что такое метод расширения и как его определить?

Упражнение 6.2. Создание иерархии наследования

Исследуйте иерархии наследования, выполнив следующие действия.

1. Создайте консольное приложение `Exercise02` и добавьте его в вашу рабочую область.
2. Создайте класс `Shape` со свойствами `Height`, `Width` и `Area`.
3. Добавьте три унаследованных класса — `Rectangle`, `Square` и `Circle` — с любыми дополнительными членами, которые, по вашему мнению, подходят и правильно переопределяют и реализуют свойство `Area`.

4. В проекте `Program.cs` в методе `Main` добавьте операторы для создания одного экземпляра каждой фигуры:

```
var r = new Rectangle(3, 4.5);
WriteLine($"Rectangle H: {r.Height}, W: {r.Width}, Area:
{r.Area}");

var s = new Square(5);
WriteLine($"Square H: {s.Height}, W: {s.Width}, Area:
{s.Area}");

var c = new Circle(2.5);
WriteLine($"Circle H: {c.Height}, W: {c.Width}, Area:
{c.Area}");
```

5. Запустите консольное приложение и убедитесь, что получили подобный результат:

```
Rectangle H: 3, W: 4.5, Area: 13.5
Square H: 5, W: 5, Area: 25
Circle H: 5, W: 5, Area: 19.6349540849362
```

Упражнение 6.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- перегрузка операций (справочник по C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/operator-overloading;
- делегаты: https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/features-delegates-and-lambda-expressions;
- ключевое слово `event` (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/event;
- интерфейсы: https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/types-interfaces;
- дженерики (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/generics;
- ссылочные типы (справочник по C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/reference-types;
- типы значений (справочник по C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/value-types;
- наследование (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/inheritance;
- методы завершения (финализаторы) (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/destructors.

Резюме

Вы познакомились с созданием локальных функций и операций, делегатов и событий, реализацией интерфейсов, дженериков и производных типов с помощью наследования и объектно-ориентированного программирования. Вы также узнали о базовых и производных классах, о том, как переопределить член типа, использовать полиморфизм и приведение типов.

Далее вы узнаете, что такое .NET 5, как его установить, какие типы предоставляются для реализации такой общей функциональности, как работа с файлами, доступ к базе данных, шифрование и многозадачность.

7 Описание и упаковка типов .NET

Изучив данную главу, вы узнаете, как ключевые слова C# связаны с типами .NET, а также о взаимосвязи между пространствами имен и сборками. Вдобавок познакомитесь с тем, как упаковать и публиковать кросс-платформенные приложения и библиотеки .NET, использовать существующие библиотеки .NET Framework в библиотеках .NET, а также с тем, как переносить кодовые базы .NET Framework в .NET.

В этой главе:

- введение в .NET Core 5;
- использование компонентов .NET;
- публикация и развертывание ваших приложений;
- декомпиляция сборок;
- упаковка библиотек для распространения NuGet;
- перенос приложений с .NET Framework на .NET 5.

Введение в .NET 5

Данная часть книги посвящена функциональным возможностям *библиотек базовых классов (BCL)* API, предоставляемых .NET Core 5, а также повторному использованию функциональных возможностей на всех различных платформах .NET с помощью .NET Standard.

Поддержка средой .NET Core 2.0 и более поздними версиями стандарта .NET Standard 2.0 важна, поскольку он содержит многие API, отсутствующие в первой версии .NET Core. Среди накопленных за 15 лет библиотек и приложений, доступных разработчикам, использующим .NET Framework, были отобраны актуальные для современной разработки и перенесены на .NET Core, так что теперь они могут работать на нескольких различных платформах в операционных системах Windows, macOS и Linux.

.NET Standard 2.1 добавил еще около 3000 новых API. Некоторые из этих API нуждаются в изменениях в среде выполнения, нарушающих обратную совмести-

мость, вследствие чего .NET Framework 4.8 реализует только .NET Standard 2.0. Платформы .NET Core 3.0, Xamarin, Mono и Unity реализуют .NET Standard 2.1.

С полным списком API в .NET Standard 2.1 и сравнение с .NET Standard 2.0 можно ознакомиться по ссылке github.com/dotnet/standard/blob/master/docs/versions/netstandard2.1.md.

.NET 5 устраняет необходимость в .NET Standard, если все ваши проекты могут использовать .NET 5. Поскольку вам все еще может потребоваться создавать библиотеки классов для устаревших проектов .NET Framework или для мобильных приложений Xamarin, то по-прежнему необходимо создавать библиотеки классов версий .NET Standard 2.0 и 2.1. После выпуска версии .NET 6 в ноябре 2021 года с поддержкой Xamarin Mobile потребность в .NET Standard снизится еще больше.

Дополнительную информацию о .NET Standard вы можете найти на сайте https://devblogs.microsoft.com/dotnet/the-future-of-net-standard/.

Чтобы продемонстрировать прогресс .NET Core за последние три года, я сравнил основные версии .NET Core с эквивалентными версиями .NET Framework и привел их в следующем списке.

- .NET Core 1.0: содержит гораздо меньше API по сравнению с платформой .NET Framework 4.6.1, выпущенной в марте 2016 года.
- .NET Core 2.0: достигнут паритет API с платформой .NET Framework 4.7.1 для современных API, поскольку обе реализуют .NET Standard 2.0.
- .NET Core 3.0: содержит больше API по сравнению с платформой .NET Framework для современных API, поскольку .NET Framework 4.8 не поддерживает .NET Standard 2.1.
- .NET 5: еще более крупный API по сравнению с .NET Framework 4.8 для современных API со значительно улучшенной производительностью.

Найти и проанализировать все .NET API можно на сайте https://docs.microsoft.com/ru-ru/dotnet/api/.

.NET Core 1.0

Платформа .NET Core 1.0 была выпущена в июне 2016 года и ориентирована на реализацию API, подходящего для создания современных кросс-платформенных приложений, включая веб- и облачные приложения, а также сервисы для Linux с использованием ASP.NET Core.

Более подробно о платформе .NET Core 1.0 можно прочитать на сайте https://devblogs.microsoft.com/dotnet/announcing-net-core-1-0/.

.NET Core 1.1

Платформа .NET Core 1.1 была выпущена в ноябре 2016 года и направлена на исправление ошибок, увеличение количества поддерживаемых дистрибутивов Linux, поддержку .NET Standard 1.6 и повышение производительности, особенно в ASP.NET Core для веб-приложений и сервисов.

Подробнее о платформе .NET Core 1.1 можно прочитать на сайте https://devblogs.microsoft.com/dotnet/announcing-net-core-1-1/.

.NET Core 2.0

Платформа .NET Core 2.0 была выпущена в августе 2017 года и сфокусирована на реализации .NET Standard 2.0, возможности использовать библиотеки .NET Framework и другие обновления.

Больше информации о платформе .NET Core 2.0 можно найти на сайте https://devblogs.microsoft.com/dotnet/announcing-net-core-2-0/.

Третье издание этой книги было выпущено в ноябре 2017 года, поэтому охватывало платформы .NET Core 2.0 и .NET Core для приложений универсальной платформы Windows.

.NET Core 2.1

Платформа .NET Core 2.1 была выпущена в мае 2018 года и ориентирована на создание расширяемой системы инструментов, в которую были добавлены новые типы, такие как `Span<T>`, новые API для криптографии и сжатия, метапакет Windows Compatibility Pack с дополнительными 20 000 API для портирования устаревших Windows-приложений, преобразование значений в Entity Framework Core, преобразование `GroupBy` в LINQ, заполнение данными, типы запросов и даже дополнительные обновления производительности, включая темы, перечисленные в табл. 7.1.

Таблица 7.1

Особенности	Глава	Тема
Интервалы, индексы, диапазоны	8	Работа с интервалами, индексами и диапазонами
Brotli (алгоритм сжатия данных с открытым исходным кодом)	9	Сжатие с помощью алгоритма Brotli (алгоритма Бротли)
Криптография	10	Что нового в криптографии
Отложенная загрузка	11	Включение отложенной загрузки
Заполнение данными	11	Заполнение данными

.NET Core 2.2

Платформа .NET Core 2.2 была выпущена в декабре 2018 года и фокусировалась на усовершенствовании диагностики для среды выполнения, опциональной многоуровневой компиляции и добавлении новых функций в ASP.NET Core и Entity Framework Core, таких как поддержка пространственных данных с использованием типов из библиотеки *NetTopologySuite (NTS)*, тегов запросов и коллекций собственных сущностей.

Более подробную информацию о платформе .NET Core 2.2 можно найти на сайте https://devblogs.microsoft.com/dotnet/announcing-net-core-2-2/.

.NET Core 3.0

Платформа .NET Core 3.0 была выпущена в сентябре 2019 года и сосредоточена на добавлении поддержки создания приложений для операционной системы Windows с использованием Windows Forms (2001), Windows Presentation Foundation (WPF; 2006) и Entity Framework 6.3, параллельном и локальном развертывании приложений, быстром чтении JSON-файлов, на решениях для *Интернета вещей* (Internet of Things, IoT) и многоуровневой компиляции по умолчанию, включая темы, перечисленные в табл. 7.2.

Четвертое издание этой книги было опубликовано в октябре 2019 года, поэтому оно охватывало некоторые новые API, добавленные в более поздних версиях.

Больше информации о платформе .NET Core 3.0 можно найти на сайте https://devblogs.microsoft.com/dotnet/announcing-net-core-3-0/.

Таблица 7.2

Особенности	Глава	Тема
Встраивание .NET Core в приложение	7	Публикация и развертывание ваших приложений
Индекс и диапазон	8	Работа с интервалами, индексами и диапазонами
System.Text.Json	9	Высокопроизводительная обработка JSON
Асинхронные потоки	13	Работа с асинхронными потоками

.NET 5.0

Платформа .NET 5.0 была выпущена в ноябре 2020 года и была направлена на унификацию различных платформ .NET, доработку платформы и повышение производительности, включая темы, перечисленные в табл. 7.3.

Таблица 7.3

Особенности	Глава	Тема
Half type	8	Работа с числами
Улучшения производительности регулярных выражений	8	Улучшения производительности регулярных выражений
System.Text.Json	9	Высокопроизводительная обработка JSON
Простой способ получить сгенерированный SQL	11	Получение сгенерированного SQL
Включение фильтрации	11	Фильтрация включенных сущностей
Scaffold-DbContext теперь сингуляризует с помощью Humanizer	11	Модели генерации кода с использованием существующей базы данных

Более подробную информацию о платформе .NET 5 можно найти на сайте https://devblogs.microsoft.com/dotnet/announcing-net-5-0.

Повышение производительности с .NET Core 2.0 до .NET 5

Компания Microsoft за последние несколько лет значительно улучшила производительность.

Подробнее об этом вы можете прочитать на сайте https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/.

Использование компонентов .NET

Платформа .NET Core состоит из нескольких частей.

- *Компиляторы языка* — преобразуют ваш исходный код, написанный на таких языках, как C#, F# и Visual Basic, в код *промежуточного языка* (intermediate language, IL), хранящийся в сборках. В языке C# 6.0 и более поздних версиях корпорация Microsoft переключилась на компилятор с открытым исходным кодом, известный как Roslyn, который также используется в Visual Basic.

О компиляторе Roslyn вы можете узнать на сайте github.com/dotnet/roslyn.

- *Common Language Runtime (общеязыковая исполняющая среда, CoreCLR)* — загружает сборки, компилирует код на IL в нативный код для ЦПУ того компьютера, на котором запущено приложение, и выполняет этот код в среде, которая управляет такими ресурсами, как потоки и память.
- *Библиотеки базовых классов* (Base Class Libraries, BCL) *сборок в пакетах NuGet (CoreFX)* — это предварительно собранные сборки типов, упакованные и распространяемые с помощью NuGet для выполнения типовых задач при разработке приложений. Вы можете использовать компоненты .NET Core для быстрого создания чего угодно, подобно тому, как вы можете комбинировать детали лего. Платформа .NET Core 2.0 реализовала .NET Standard 2.0 — расширенный набор всех предыдущих версий .NET Standard — и достигла паритета с .NET Framework и Xamarin. Платформа .NET Core 3.0 реализует .NET Standard 2.1, что добавляет новые возможности и позволяет повысить производительность по сравнению с функциональностью .NET Framework. .NET 5 реализует унифицированный BCL для всех типов приложений (кроме мобильных). .NET 6 будет реализовывать унифицированный BCL для всех типов приложений, включая мобильные.

Более подробную информацию о .NET Standard 2.1 можно получить на сайте https://devblogs.microsoft.com/dotnet/announcing-net-standard-2-1/.

Сборки, пакеты и пространства имен

Сборка — это место, где тип хранится в файловой системе. Сборки — механизм для развертывания кода. Например, сборка `System.Data.dll` содержит типы для управления данными. Чтобы использовать типы в других сборках, на них должна быть указана ссылка.

Сборки часто распространяются в виде *пакетов NuGet*, которые могут содержать несколько сборок и других ресурсов. Вы также узнаете о *метапакетах* и *платформах* — комбинациях пакетов NuGet.

Пространство имен — это адрес типа. Пространство имен — механизм уникальной идентификации типа, требующий полный адрес, а не просто короткое имя. В реальной жизни *Борис*, проживающий по адресу *улица Ясная, 34*, отличается от *Бориса*, проживающего по адресу *улица Темная, 12*.

В среде .NET интерфейс `IActionFilter` пространства имен `System.Web.Mvc` отличается от интерфейса `IActionFilter` пространства имен `System.Web.Http.Filters`.

Зависимости между сборками

Если сборка компилируется в виде библиотеки классов (предоставляет типы другим сборкам), то ей присваивается расширение `.dll` (dynamic link library, библиотека динамической компоновки) и она не может выполняться автономно.

Если сборка компилируется как приложение, то ей присваивается расширение `.exe` (executable, исполняемый файл) и она может выполняться автономно. До выпуска версии .NET Core 3.0 консольные приложения компилировались в файлы `.dll` и должны были выполняться командой `dotnet run` или хостовым исполняемым файлом.

Любые сборки (как приложения, так и библиотеки классов) могут ссылаться на одну или несколько сборок — библиотек классов, которые в этом случае станут зависимостями ссылающейся сборки. Однако вы не можете использовать циклические ссылки, и потому сборка `B` не может ссылаться на сборку `A`, если та уже ссылается на сборку `B`. Компилятор оповестит вас, если вы попытаетесь добавить зависимость так, что при этом появится циклическая ссылка.

Циклические ссылки зачастую говорят о плохом коде. Если вы уверены, что вам нужна циклическая ссылка, то воспользуйтесь интерфейсом для решения этой проблемы, как описано на Stack Overflow: https://stackoverflow.com/questions/6928387/how-to-solve-circular-reference.

Платформа Microsoft .NET и пакет SDK

По умолчанию консольные приложения содержат ссылку на Microsoft .NET SDK. Эта специальная платформа содержит тысячи типов, таких как `int` и `string`, доступных в виде NuGet-пакетов, которые требуются при разработке практически всех приложений.

Дополнительную информацию о пакетах SDK для проектов .NET вы можете изучить на сайте https://docs.microsoft.com/ru-ru/dotnet/core/project-sdk/overview.

При использовании .NET вы ссылаетесь на сборки-зависимости, пакеты NuGet и платформы, которые нужны вашему приложению, в файле проекта.

Рассмотрим отношения между сборками и пространствами имен.

1. В программе Visual Studio Code создайте подпапку AssembliesAndNamespaces в папке Chapter07 и введите для создания консольного приложения команду `dotnet new console`.
2. Сохраните в папке Chapter07 текущую рабочую область под именем `Chapter07` и добавьте в нее подпапку AssembliesAndNamespaces.
3. Откройте проект `AssembliesAndNamespaces.csproj` и обратите внимание, что это типичный файл проекта для приложения .NET:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Хотя имеется возможность включить сборки, которые использует ваше приложение, в пакет для его распространения, по умолчанию проект будет проверять наличие общих сборок, установленных по известным путям.

В первую очередь проект ищет указанную версию .NET Core в папках текущего пользователя .dotnet/store и .nuget, а затем ищет резервную папку, которая зависит от вашей операционной системы:

- Windows: `C:\Program Files\dotnet\sdk`;
- macOS: `/usr/local/share/dotnet/sdk`.

Наиболее распространенные типы .NET находятся в сборке `System.Runtime.dll`. Вы можете увидеть связь между отдельными сборками и пространствами имен, для которых они предоставляют типы, и заметить, что не всегда существует взаимно однозначное сопоставление между сборками и пространствами имен, как показано в табл. 7.4.

Таблица 7.4

Сборки	**Пример пространства имен**	**Пример типов**
System.Runtime.dll	System, System.Collections, System.Collections.Generic	Int32, String, IEnumerable<T>
System.Console.dll	System	Console
System.Threading.dll	System.Threading	Interlocked, Monitor, Mutex
System.Xml.XDocument.dll	System.Xml.Linq	XDocument, XElement, XNode

Пакеты NuGet

Платформа .NET Core разделена на набор пакетов, распространяемых с помощью поддерживаемой Microsoft технологии управления пакетами под названием NuGet. Каждый из этих пакетов представляет собой одну сборку с таким же именем. Например, пакет `System.Collections` содержит сборку `System.Collections.dll`.

Ниже приведены преимущества пакетов:

- поставка каждого из пакетов может производиться по собственному графику;
- пакеты могут быть протестированы независимо от других пакетов;
- пакеты могут поддерживать разные операционные системы и процессоры, так как могут включать несколько версий одной и той же сборки, созданной для разных операционных систем и процессоров;
- пакеты могут содержать зависимости, характерные только для одной библиотеки;
- приложения меньше, поскольку несвязанные пакеты не входят в состав дистрибутива.

В табл. 7.5 перечислены некоторые из наиболее важных пакетов и их важные типы.

Таблица 7.5

Пакеты	Важные типы
System.Runtime	Object, String, Int32, Array
System.Collections	List<T>, Dictionary<TKey, TValue>
System.Net.Http	HttpClient, HttpResponseMessage
System.IO.FileSystem	File, Directory
System.Reflection	Assembly, TypeInfo, MethodInfo

Фреймворки

Существует двусторонняя связь между фреймворками и пакетами. Пакеты определяют API, а фреймворки группируют пакеты. Фреймворк без каких-либо пакетов не определит никакие API.

Если вы хорошо разбираетесь в интерфейсах и типах, их реализующих, то можете перейти по следующему адресу, по которому можно узнать, как пакеты и их API связаны с такими фреймворками, как различные версии .NET Standard: https://gist.github.com/davidfowl/8939f305567e1755412d6dc0b8baf1b7.

Каждый пакет .NET поддерживает набор фреймворков. Например, пакет `System.IO.FileSystem` версии 4.3.0 поддерживает следующие фреймворки:

- .NET Standard, версия 1.3 или более поздняя;
- .NET Framework, версия 4.6 или более поздняя;
- Платформы Six Mono и Xamarin (например, Xamarin.iOS 1.0).

Более подробную информацию можно найти на сайте https://www.nuget.org/packages/System.IO.FileSystem/.

Импорт пространства имен для использования типа

Рассмотрим, как пространства имен связаны со сборками и типами.

1. В проекте `AssembliesAndNamespaces` в методе `Main` введите следующий код:

   ```
   var doc = new XDocument();
   ```

   Тип `XDocument` не распознается, поскольку мы не сообщили компилятору пространство имен для этого типа. Хотя в данном проекте уже есть ссылка на сборку, содержащую тип, нам также необходимо либо добавить пространство имен как префикс к имени типа, либо импортировать пространство имен.
2. Щелкните кнопкой мыши на имени класса `XDocument`. Программа Visual Studio Code отобразит знак в виде лампочки, тем самым показывая, что программа распознает тип и может автоматически решить проблему.
3. Щелкните на знаке лампочки или нажмите сочетание клавиш Ctrl+. (точка).
4. Выберите в меню пункт `using System.Xml.Linq;`.

Таким образом вы импортировали пространство имен, добавив оператор `using` в начало файла. Как только пространство имен импортируется в начале файла с кодом, все типы в пространстве имен становятся доступными для использования в этом файле путем простого указания имени типа, без необходимости полностью его квалифицировать, задействуя префикс его пространства имен.

Связь ключевых слов языка C# с типами .NET

Один из распространенных вопросов, которые я получаю от неопытных программистов C#: «*Отличаются ли чем-то типы string (строчная буква в начале) и String (прописная буква в начале)?*»

Простой ответ: нет. Полный ответ заключается в том, что все ключевые слова C#, представляющие собой названия типов, — это псевдонимы для типа .NET в сборке библиотеки классов.

Когда вы используете ключевое слово `string`, компилятор преобразует его в тип `System.String`. А тип `int` — в тип `System.Int32`.

1. Объявите в методе `Main` две переменные для хранения значений `string`: с помощью строчных букв и прописных:

```
string s1 = "Hello";
String s2 = "World";

WriteLine($"{s1} {s2}");
```

   На данный момент обе переменные работают одинаково хорошо и имеют одинаковый смысл.

2. В начале файла класса закомментируйте строку `using System;`, поставив перед префиксом оператора символы `//`, и обратите внимание на ошибку компилятора.
3. Чтобы исправить ошибку, удалите косые черты, обозначающие комментарий.

По возможности используйте вместо фактического типа ключевое слово C#, поскольку ключевые слова не нуждаются в импорте пространства имен.

В табл. 7.6 приведены 16 ключевых слов C#, представляющих собой названия типов, и их фактические типы .NET.

Таблица 7.6

Ключевое слово	Тип .NET	Ключевое слово	Тип .NET
string	System.String	char	System.Char
sbyte	System.SByte	byte	System.Byte
short	System.Int16	ushort	System.UInt16
int	System.Int32	uint	System.UInt32
long	System.Int64	ulong	System.UInt64
float	System.Single	double	System.Double
decimal	System.Decimal	bool	System.Boolean
object	System.Object	dynamic	System.Dynamic.DynamicObject

Другие компиляторы языков программирования .NET могут выполнять то же самое. Например, язык Visual Basic .NET содержит тип `Integer` — псевдоним для `System.Int32`.

4. Щелкните правой кнопкой мыши на слове XDocument и выберите пункт Go to Definition (Перейти к определению) в контекстном меню или нажмите клавишу F12.
5. Перейдите к началу файла и обратите внимание, что имя файла сборки — `System.Xml.XDocument.dll`. Однако класс находится в пространстве имен `System.Xml.Linq` (рис. 7.1).

```
#region Assembly System.Xml.XDocument, Version=5.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
// System.Xml.XDocument.dll
#endregion

using System.IO;
using System.Threading;
using System.Threading.Tasks;

namespace System.Xml.Linq
{
  //
  // Summary:
  //     Represents an XML document. For the components and usage of an System.Xml.Linq.XDocument
  //     object, see XDocument Class Overview.
  public class XDocument : XContainer
  {
```

Рис. 7.1. Информация о файле сборки

6. Закройте вкладку [metadata] XDocument.cs.
7. Щелкните правой кнопкой мыши на слове `string` или `String` и выберите пункт Go to Definition (Перейти к определению) в контекстном меню или нажмите клавишу F12.
8. Перейдите к началу файла и обратите внимание, что имя файла сборки — `System.Runtime.dll`. Однако класс находится в пространстве имен `System`.

Создание кросс-платформенных библиотек классов при помощи .NET Standard

До выпуска .NET Standard существовали *переносимые библиотеки классов* (Portable Class Libraries, PCL). С помощью PCL можно было создать библиотеку кода и явно указать, какие платформы должна поддерживать библиотека, например Xamarin, Silverlight и Windows 8. Ваша библиотека могла бы затем использовать пересечение API, поддерживаемых указанными платформами.

Сотрудники корпорации Microsoft поняли, что это нестабильный подход, и потому разработали .NET Standard — единый API, который будет поддерживаться всеми будущими платформами .NET. Существуют более старые версии .NET Standard, но только .NET Standard 2.0 и более поздние версии поддерживаются несколькими платформами .NET. Версия .NET Standard 2.1 была выпущена в конце 2019 года, но только .NET Core 3.0 и версия Xamarin того года поддерживают его новые

функции. Далее я буду использовать термин .NET Standard для обозначения .NET Standard 2.0 или более поздней версии.

Идейно .NET Standard очень похож на HTML5 — это стандарты, которые должна поддерживать платформа. Так же как браузеры Google Chrome и Microsoft Edge реализуют стандарт HTML5, .NET Core, .NET Framework и Xamarin реализуют .NET Standard. Если вы хотите создать библиотеку типов, которая будет работать с различными версиями .NET, то проще всего сделать это с помощью .NET Standard.

Многие добавленные в .NET Standard 2.1 API требуют изменений в среде выполнения, а .NET Framework — устаревшая платформа Microsoft, которая должна оставаться насколько возможно неизменной. Поэтому платформа .NET Framework 4.8 останется на .NET Standard 2.0, а не на NET Standard 2.1. Если вам требуется поддержка заказчиков, работающих с .NET Framework, то следует разрабатывать библиотеки классов в .NET Standard 2.0, даже если это не самая последняя версия и не поддерживает все новые функции BCL.

Выбор версии .NET Standard сводится к балансу между максимальным числом поддерживаемых платформ и доступной функциональностью. Более низкая версия поддерживает больше платформ, но имеет меньший набор API. С более высокой версией ситуация обратная: меньше платформ, но больший набор API. Как правило, вы должны выбирать наименьшую версию, поддерживающую все необходимые вам API.

Версии .NET Standard и поддерживаемые ими платформы перечислены в табл. 7.7.

Таблица 7.7

Платформа	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**2.0**	**2.1**
.NET Core	→	→	→	→	→	1.0, 1.1	2.0	3.0
.NET Framework	4.5	4.5.1	4.6	→	→	→	4.6.1	н/д
Mono	→	→	→	→	→	4.6	5.4	6.2
Xamarin.iOS	→	→	→	→	→	10.0	10.14	12.12
UWP	→	→	→	10	→	→	10.0.16299	н/д

Создание библиотеки классов .NET Standard 2.0

Мы разработаем библиотеку классов с помощью .NET Standard 2.0, чтобы ее можно было использовать на всех важных платформах .NET и кросс-платформенно в операционных системах Windows, macOS и Linux, а также чтобы иметь доступ к широкому набору API .NET.

1. Создайте в папке Code/Chapter07 подпапку SharedLibrary.
2. В программе Visual Studio Code добавьте в рабочую область `Chapter07` папку SharedLibrary.
3. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку SharedLibrary.
4. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new classlib
```

5. Выберите файл `SharedLibrary.csproj` и обратите внимание, что библиотека классов, написанная с помощью интерфейса командной строки (CLI) `dotnet`, по умолчанию предназначена для .NET 5.0:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

6. Измените целевую платформу, как показано в следующем коде:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Если вам нужно создать тип, который использует новые функции .NET 5, создайте отдельные библиотеки классов: одну — для .NET Standard 2.0, а другую — для .NET 5. Эту тему мы будем изучать в главе 11.

Публикация и развертывание ваших приложений

Существует три способа публикации и развертывания приложения .NET:

- платформозависимое развертывание (framework-dependent deployment, FDD);
- зависящие от платформы исполняемые файлы (framework-dependent executables, FDE);
- автономное (self-contained).

Если вы решили развернуть свое приложение и его зависимые пакеты, но не саму платформу .NET, то вы будете полагаться на то, что она уже установлена на целевом компьютере. Это хорошо работает для веб-приложений, развернутых на сервере,

поскольку .NET и множество других веб-приложений, вероятно, уже находятся на сервере.

Иногда вам необходимо передать USB-накопитель с вашим приложением другому пользователю и быть уверенными, что оно станет работать на другом компьютере. Вы хотите выполнить автономное развертывание. Даже если размер файлов развертывания больше, вы будете уверены, что все будет работать.

Разработка консольного приложения для публикации

Рассмотрим, как опубликовать консольное приложение.

1. Создайте проект консольного приложения `DotNetCoreEverywhere` и добавьте его в рабочую область `Chapter07`.
2. Добавьте в файл `Program.cs` следующий оператор для вывода сообщения о том, что консольное приложение может работать везде:

```
using static System.Console;

namespace DotNetCoreEverywhere
{
  class Program
  {
    static void Main(string[] args)
    {
      WriteLine("I can run everywhere!");
    }
  }
}
```

3. Откройте файл `DotNetCoreEverywhere.csproj` и добавьте идентификаторы среды выполнения для трех операционных систем внутри элемента `<PropertyGroup>`, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <RuntimeIdentifiers>
       win10-x64;osx-x64;rhel.7.4-x64
    </RuntimeIdentifiers>
  </PropertyGroup>

</Project>
```

- Идентификатор `win10-x64` означает операционную систему Windows 10 или Windows Server 2016.

- Идентификатор `osx-x64` означает операционную систему macOS Sierra 10.12 или более позднюю версию.
- Идентификатор `rhel.7.4-x64` означает операционную систему *Red Hat Enterprise Linux (RHEL)* 7.4 или более позднюю версию.

Найти текущие поддерживаемые значения идентификатора среды выполнения (Runtime IDentifier, RID) можно на сайте https://docs.microsoft.com/ru-ru/dotnet/articles/core/rid-catalog.

Команды dotnet

Установка .NET SDK включает в себя *интерфейс командной строки* (command-line interface, CLI) `dotnet`.

Создание новых проектов

Интерфейс командной строки `dotnet` содержит команды, работающие с текущей папкой в целях создания нового проекта с помощью шаблонов.

1. В программе Visual Studio Code перейдите к панели TERMINAL (Терминал).
2. Введите команду `dotnet new -l`, чтобы вывести список установленных шаблонов (рис. 7.2).

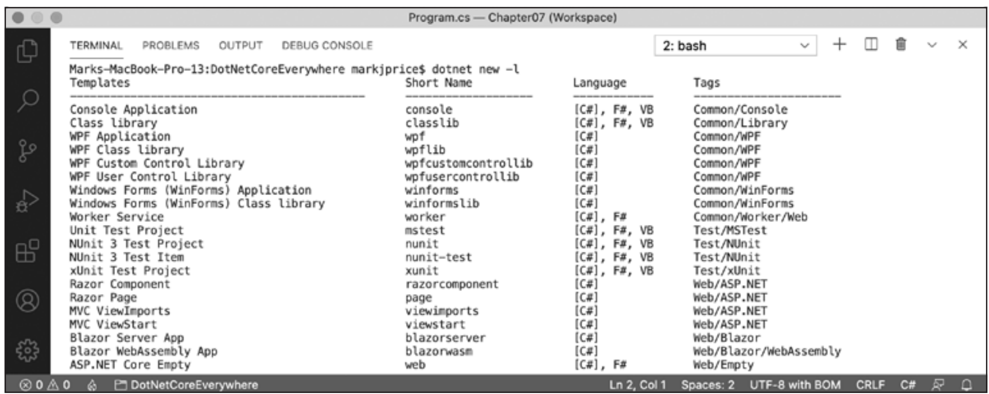


Рис. 7.2. Список установленных шаблонов

Дополнительные шаблоны вы можете установить по следующей ссылке: https://dotnetnew.azurewebsites.net/.

Управление проектами

Интерфейс командной строки `dotnet` содержит команды, управляющие проектом в текущей папке:

- `dotnet restore` — загружает зависимости проекта;
- `dotnet build` — компилирует проект;
- `dotnet test` — запускает модульные тесты проекта;
- `dotnet run` — запускает проект;
- `dotnet pack` — создает пакет NuGet для проекта;
- `dotnet publish` — компилирует, а затем публикует проект с зависимостями или как отдельное приложение;
- `add` — добавляет ссылку на пакет или библиотеку классов в проект;
- `remove` — удаляет ссылку на пакет или библиотеку классов из проекта;
- `list` — перечисляет ссылки на пакет или библиотеки классов для проекта.

Публикация автономного приложения

Теперь мы можем опубликовать наше кросс-платформенное консольное приложение.

1. В программе Visual Studio Code перейдите к панели TERMINAL (Терминал). Чтобы собрать версию выпуска консольного приложения для операционной системы Windows 10, введите следующую команду:

   ```
   dotnet publish -c Release -r win10-x64
   ```

   Затем платформа Microsoft Build Engine скомпилирует и опубликует консольное приложение.
2. На панели TERMINAL (Терминал) введите следующие команды для сборки версий выпуска для операционных систем macOS и RHEL:

   ```
   dotnet publish -c Release -r osx-x64
   dotnet publish -c Release -r rhel.7.4-x64
   ```

3. Откройте окно Finder в macOS или проводник Windows, перейдите к `DotNetCoreEverywhere\bin\Release\net5.0` и обратите внимание на созданные в итоге папки для трех операционных систем и файлов.
4. В папке `osx-x64` выберите папку публикации, обратите внимание на все поддерживающие сборки, а затем выберите исполняемый файл `DotNetCoreEverywhere` и обратите внимание, что размер исполняемого файла составляет около 95 Кбайт (рис. 7.3).

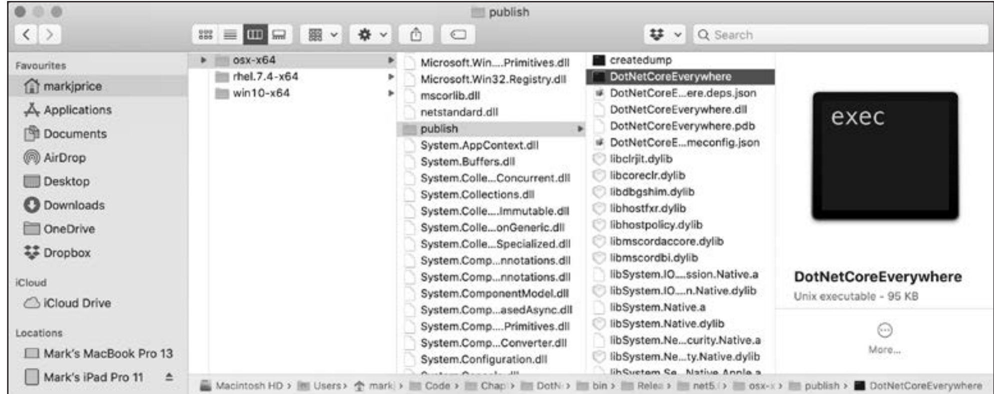


Рис. 7.3. Исполняемый файл DotNetEverywhere для macOS

5. Дважды щелкните на исполняемом файле и обратите внимание на результаты (рис. 7.4).

```
markjprice — DotNetCoreEverywhere — 153×14
Last login: Sun Sep 13 17:38:13 on ttys002

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Marks-MacBook-Pro-13:~ markjprice$ /Users/markjprice/Code/Chapter07/DotNetCoreEverywhere/bin/Release/net5.0/osx-x64/DotNetCoreEverywhere ; exit;
I can run everywhere!
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Process completed]
```

Рис. 7.4. Результаты выполнения DotNetEverywhere

Если вы скопируете любую из этих папок в соответствующую операционную систему, то запустится консольное приложение — это работает, так как мы создали автономное приложение .NET.

Публикация однофайлового приложения

Чтобы опубликовать однофайловое приложение, вы можете указать флаги. Однако в .NET 5.0 однофайловые приложения в первую очередь ориентированы на Linux, поскольку как в Windows, так и в macOS существуют ограничения, которые означают, что настоящая однофайловая публикация технически невозможна.

Если вы можете предположить, что версия .NET 5 уже установлена на компьютере, вы можете использовать следующие флаги:

```
dotnet publish -r win10-x64 -c Release --self-contained=false
/p:PublishSingleFile=true
```

В результате будут созданы два файла: `DotNetCoreEverywhere.exe` и `DotNetCoreEverywhere.pdb`. Файл с расширением `.exe` — исполняемый. Файл с расширением `.pdb` — файл базы данных программы, в котором хранится отладочная информация.

Вы можете прочитать о файлах .pdb на сайте https://www.wintellect.com/pdb-files-what-every-developer-must-know/.

Если вам необходимо, чтобы файл `.pdb` был встроен в исполняемый файл, добавьте элемент в свой файл `.csproj`:

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net5.0</TargetFramework>
  <DebugType>embed</DebugType>
</PropertyGroup>
```

Если вы не можете предположить, что .NET 5 установлена на компьютере, то, хотя Linux также генерирует только два файла, ожидайте следующих дополнительных файлов для Windows: `coreclr.dll`, `clrjit.dll`, `clrcompression.dll` и `mscordaccore.dll`.

Рассмотрим пример для операционной системы macOS.

1. В программе Visual Studio Code перейдите к панели `TERMINAL` (Терминал). Чтобы создать версию выпуска консольного приложения для операционной системы Windows 10, введите следующую команду:

   ```
   dotnet publish -c Release -r osx-x64 /p:PublishSingleFile=true
   ```

2. Перейдите в папку `DotNetCoreEverywhere\bin\Release\net5.0\osx-x64\publish`, выберите исполняемый файл `DotNetEverywhere` и обратите внимание, что размер исполняемого файла теперь составляет около 52 Мбайт, имеется файл `.pdb` и семь файлов `.dylib`.

Более подробно об однофайловом приложении вы можете прочитать на сайте https://github.com/dotnet/runtime/issues/36590.

Уменьшение размера приложений с помощью обрезки приложений

Одна из проблем при развертывании приложения .NET как автономного приложения заключается в том, что библиотеки .NET 5 занимают слишком много места. Одна из самых больших потребностей в уменьшении размера — это компоненты Blazor WebAssembly, так как все библиотеки .NET 5 необходимо загрузить в браузер.

К счастью, вы можете уменьшить этот размер, не упаковывая неиспользуемые сборки. Система обрезки приложений, представленная в .NET Core 3.0, может определять сборки, необходимые для вашего кода, и удалять ненужные.

В .NET 5 обрезка удаляет отдельные типы и даже члены, такие как методы, из сборки, если они не используются. Например, при работе с консольным приложением Hello World сборка `System.Console.dll` сокращена с 61,5 до 31,5 Кбайт. Это экспериментальная функция, поэтому используйте ее с осторожностью.

Проблема в том, насколько хорошо обрезка идентифицирует неиспользуемые сборки, типы и элементы. Если ваш код динамический, возможно, с использованием отражения, он может работать некорректно, поэтому Microsoft также допускает ручное управление.

Существует два способа включить обрезку на уровне сборки. Сначала добавьте элемент в файл проекта:

```
<PublishTrimmed>true</PublishTrimmed>
```

Затем добавьте при публикации флаг:

```
-p:PublishTrimmed=True
```

Существует два способа включить обрезку на уровне элементов.

Сначала добавьте элемент в файл проекта:

```
<PublishTrimmed>true</PublishTrimmed>
<TrimMode>Link</TrimMode>
```

Затем добавьте при публикации флаг:

```
-p:PublishTrimmed=True -p:TrimMode=Link
```

Более подробно об обрезке приложений вы можете прочитать на сайте https://devblogs.microsoft.com/dotnet/app-trimming-in-net-5/.

Декомпиляция сборок

Один из лучших способов научиться программировать для .NET — наблюдать, как это делают профессионалы.

В целях обучения вы можете декомпилировать любую сборку .NET с помощью такого инструмента, как *ILSpy*. Если вы еще не установили расширение *ILSpy .NET Decompiler* для Visual Studio Code, найдите его и установите.

Вы можете декомпилировать сборки других разработчиков и в целях, не связанных с обучением, но всегда помните, что пользуетесь чьей-то интеллектуальной собственностью, и относитесь к этому с уважением.

1. В программе Visual Studio Code выберите View ▸ Command Palette (Вид ▸ Панель команд) или нажмите сочетание клавиш Cmd+Shift+P.
2. Введите `ilspy`, а затем выберите ILSpy: Decompile IL Assembly (выбрать файл).
3. Перейдите в папку Code/Chapter07/DotNetCodeEverywhere/bin/Release/net5.0/osx-x64.
4. Выберите сборку System.IO.FileSystem.dll и щелкните на пункте Select assembly (Выбрать сборку).
5. В окне EXPLORER (Проводник) разверните Ilspy Decompiled Members, выберите сборку и обратите внимание на два окна редактирования, в которых отображаются атрибуты текущей сборки, для чего используется язык C#, и ссылки на внешние DLL и сборки, для чего используется IL (рис. 7.5).

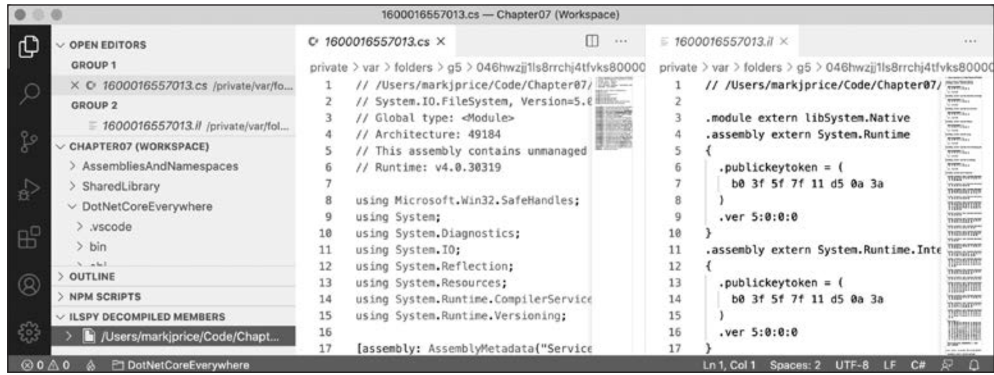


Рис. 7.5. Расширение Ilspy Decompiler

6. В коде IL обратите внимание на ссылку на сборку `System.Runtime`, включая номер версии:

```
.assembly extern System.Runtime
{
  .publickeytoken = (
    b0 3f 5f 7f 11 d5 0a 3a
  )
  .ver 5:0:0:0
}
```

Элемент `.module extern kernel32.dll` означает, что эта сборка выполняет вызовы функций Win32 API, как и следовало ожидать от кода, взаимодействующего с файловой системой.

7. На панели EXPLORER (Проводник) разверните пространства имен, выберите из них System.IO, пункт меню Directory (Каталог) и обратите внимание на два открывающихся окна редактирования, в которых отображается декомпилированный класс `Directory` как на языке C#, так и на IL (рис. 7.6).

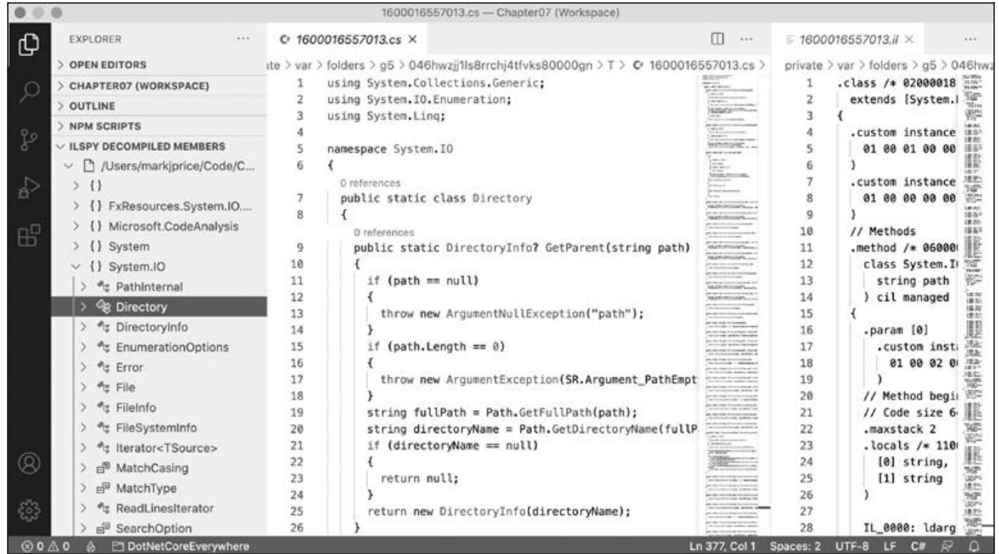


Рис. 7.6. Декомпилированный класс Directory в коде C# и IL

8. Сравните исходный код C# метода `GetParent`:

```
public static DirectoryInfo GetParent(string path)
{
  if (path == null)
  {
    throw new ArgumentNullException("path");
  }
  if (path.Length == 0)
  {
    throw new ArgumentException(SR.Argument_PathEmpty, "path");
  }
  string fullPath = Path.GetFullPath(path);
  string directoryName = Path.GetDirectoryName(fullPath);
  if (directoryName == null)
  {
    return null;
  }
  return new DirectoryInfo(directoryName);
}
```

с эквивалентным исходным кодом метода GetParent на IL:

```
.method /* 06000067 */ public hidebysig static
  class System.IO.DirectoryInfo GetParent (
    string path
  ) cil managed
{
  .param [0]
    .custom instance void System.Runtime.CompilerServices.
    NullableAttribute::.ctor(uint8) = (
    01 00 02 00 00
    )
  // Метод начинается с RVA 0x62d4
  // Размер 64 (0x40)
  .maxstack 2
  .locals /* 1100000E */ (
    [0] string,
    [1] string
  )

  IL_0000: ldarg.0
  IL_0001: brtrue.s IL_000e

  IL_0003: ldstr "path" /* 700005CB */
  IL_0008: newobj instance void
    [System.Runtime]System.ArgumentNullException::.ctor(string)
    /* 0A000035 */
  IL_000d: throw

  IL_000e: ldarg.0
  IL_000f: callvirt instance int32 [System.Runtime]System.String::get_
  Length() /* 0A000022 */
  IL_0014: brtrue.s IL_0026
  IL_0016: call string System.SR::get_Argument_PathEmpty() /* 0600004C */
  IL_001b: ldstr "path" /* 700005CB */
  IL_0020: newobj instance void [System.Runtime]System.ArgumentException::
  .ctor(string, string) /* 0A000036 */
  IL_0025: throw
  IL_0026: ldarg.0
  IL_0027: call string [System.Runtime.Extensions]System.
  IO.Path::GetFullPath(string) /* 0A000037 */
  IL_002c: stloc.0
  IL_002d: ldloc.0
  IL_002e: call string [System.Runtime.Extensions]System.IO.Path::
  GetDirectoryName(string) /* 0A000038 */
  IL_0033: stloc.1
  IL_0034: ldloc.1
  IL_0035: brtrue.s IL_0039
  IL_0037: ldnull
  IL_0038: ret
```

```
    IL_0039: ldloc.1
    IL_003a: newobj instance void System.IO.DirectoryInfo::.ctor(string)
    /* 06000097 */
    IL_003f: ret
  } // конец метода Directory::GetParent
```

Окна редактирования кода IL не будут вам особенно полезны, пока вы не разберетесь в разработке приложений на C# и .NET достаточно хорошо — настолько, что уже станет действительно важно знать, как компилятор C# преобразует ваш исходный код в код IL. Более полезные окна редактирования содержат эквивалентный исходный код на C#, написанный экспертами корпорации Microsoft. Вы можете узнать много полезной и важной информации, увидев, как профессионалы реализуют типы.

9. Закройте окна редактирования без сохранения изменений.
10. На панели EXPLORER (Проводник) в ILSPY DECOMPILED MEMBERS щелкните правой кнопкой мыши на сборке и выберите пункт меню Unload Assembly (Выгрузить сборку).

Упаковка библиотек для распространения через NuGet

Прежде чем мы научимся создавать и упаковывать наши собственные библиотеки, рассмотрим, как проект может использовать существующий пакет.

Ссылка на пакет NuGet

Предположим, вы хотите добавить пакет, созданный сторонним разработчиком, например `Newtonsoft.Json` — популярный пакет для работы с форматом сериализации *JavaScript Object Notation (JSON)*.

1. В программе Visual Studio Code откройте проект `AssembliesAndNamespaces`.
2. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet add package newtonsoft.json
```

3. Откройте проект `AssembliesAndNamespaces.csproj`, и вы увидите, что ссылка на пакет добавлена, как показано ниже (выделено полужирным шрифтом):

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

```
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="newtonsoft.json"
      Version="12.0.3" />
  </ItemGroup>
</Project>
```

У вас может быть установлена более новая версия пакета `newtonsoft.json`, поскольку с момента публикации этой книги наверняка было выпущено обновление.

Фиксация зависимостей

Чтобы восстанавливать пакеты целостно и писать надежный код, важно *фиксировать зависимости*. Фиксирование зависимостей обеспечит использование одного и то же семейства пакетов, выпущенных для конкретной версии .NET, например 5.

Чтобы зафиксировать зависимости, каждый пакет должен содержать одну версию без дополнительных спецификаторов. Дополнительные спецификаторы включают в себя бета-версии (`beta1`), релиз-кандидаты (`rc4`) и подстановочные символы (*). Подстановочные символы позволяют автоматически ссылаться и использовать будущие версии, поскольку символы всегда представляют самый последний выпуск. Но подстановочные символы особенно опасны, поскольку могут привести к применению несовместимых пакетов, что повлечет неработоспособность приложения.

Если вы используете команду `dotnet add package`, то всегда будет выбрана самая последняя версия пакета. Но если вы копируете и вставляете конфигурацию из статьи блога или вручную добавляете ссылку, то можете включить подстановочные знаки.

Следующие зависимости не зафиксированы, чего следует избегать:

```
<PackageReference Include="System.Net.Http"
                  Version="4.1.0-*" />
<PackageReference Include="Newtonsoft.Json"
                  Version="12.0.3-beta1" />
```

Сотрудники корпорации Microsoft гарантируют совместную работу пакетов, если вы зафиксировали свою зависимость на версии, поставляемой с определенной версией .NET, например 5. Всегда фиксируйте свои зависимости.

Упаковка библиотеки для NuGet

Теперь упакуем ранее созданный проект `SharedLibrary`.

1. В проекте `SharedLibrary` переименуйте файл `Class1.cs` в `StringExtensions.cs` и измените его содержимое:

```
using System.Text.RegularExpressions;

namespace Packt.Shared
{
  public static class StringExtensions
  {
    public static bool IsValidXmlTag(this string input)
    {
      return Regex.IsMatch(input,
        @"^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$");
    }

    public static bool IsValidPassword(this string input)
    {
      // минимум восемь допустимых символов
      return Regex.IsMatch(input, "^[a-zA-Z0-9_-]{8,}$");
    }

    public static bool IsValidHex(this string input)
    {
      // три или шесть допустимых символов шестнадцатеричного числа
      return Regex.IsMatch(input,
        "^#?([a-fA-F0-9]{3}|[a-fA-F0-9]{6})$");
    }
  }
}
```

   Данные методы расширения для проверки значения `string` используют регулярные выражения. Как писать такие выражения, вы узнаете в главе 8.

2. Отредактируйте проект `SharedLibrary.csproj` и измените его содержимое, как показано ниже, а также обратите внимание на следующие моменты:

   - идентификатор `PackageId` — уникальный, поэтому вы должны использовать другое значение, если хотите опубликовать данный пакет NuGet в общедоступном веб-канале www.nuget.org;
   - `PackageLicenseExpression` — значение из spdx.org/licenses/, или вы можете указать собственную лицензию;
   - все остальные элементы не требуют объяснения.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
```

```
    <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
    <PackageId>Packt.CSdotnet.SharedLibrary</PackageId>
    <PackageVersion>5.0.0.0</PackageVersion>
    <Title>C# 9 and .NET 5 Shared Library</Title>
    <Authors>Mark J Price</Authors>
    <PackageLicenseExpression> MS-PL
    </PackageLicenseExpression>
    <PackageProjectUrl> http://github.com/markjprice/cs9dotnet5
    </PackageProjectUrl>
    <PackageIcon>packt-csdotnet-sharedlibrary.png</PackageIcon>
    <PackageRequireLicenseAcceptance>true
      </PackageRequireLicenseAcceptance>
    <PackageReleaseNotes>
      Example shared library packaged for NuGet.
    </PackageReleaseNotes>
    <Description>
      Three extension methods to validate a string value.
    </Description>
    <Copyright>
      Copyright © 2020 Packt Publishing Limited
    </Copyright>
    <PackageTags>string extensions packt csharp net5</PackageTags>
  </PropertyGroup>

  <ItemGroup>
    <None Include="packt-csdotnet-sharedlibrary.png">
      <Pack>True</Pack>
      <PackagePath></PackagePath>
    </None>
  </ItemGroup>
</Project>
```

Свойства конфигурации, которые допускают значения `true` или `false`, не могут содержать пробелов, поэтому запись `<PackageRequireLicenseAcceptance>` не допускает возврата каретки и отступа, которые присутствуют в предыдущем коде ввиду ограничений пространства на книжном листе.

3. Скачайте файл, содержащий значок, и сохраните его в папке SharedLibrary по следующей ссылке: https://github.com/markjprice/cs9dotnet5/tree/master/Chapter07/SharedLibrary/packt-csdotnet-sharedlibrary.png.
4. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал), а затем папку SharedLibrary.
5. На панели TERMINAL (Терминал) введите команды для выпуска сборки, а затем сгенерируйте пакет NuGet:

```
dotnet build -c Release
dotnet pack -c Release
```

6. Запустите ваш любимый браузер и перейдите по следующей ссылке: https://www.nuget.org/packages/manage/upload.

7. Если вы хотите загрузить пакет NuGet, чтобы другие разработчики могли ссылаться на него как на пакет зависимостей, то вам необходимо войти в систему через учетную запись Microsoft по адресу https://www.nuget.org/.
8. Нажмите Browse (Обзор) и выберите файл `.nupkg`, созданный командой `pack`. Путь к папке выглядит следующим образом: `Code\Chapter07\SharedLibrary\bin\Release`, а файл должен называться `Packt.CSdotnet.SharedLibrary.5.0.0.nupkg`.
9. Убедитесь, что информация, введенная вами в файл `SharedLibrary.csproj`, правильная, а затем нажмите кнопку Submit (Отправить).
10. Через несколько секунд отобразится сообщение об успешном завершении загрузки вашего пакета (рис. 7.7).

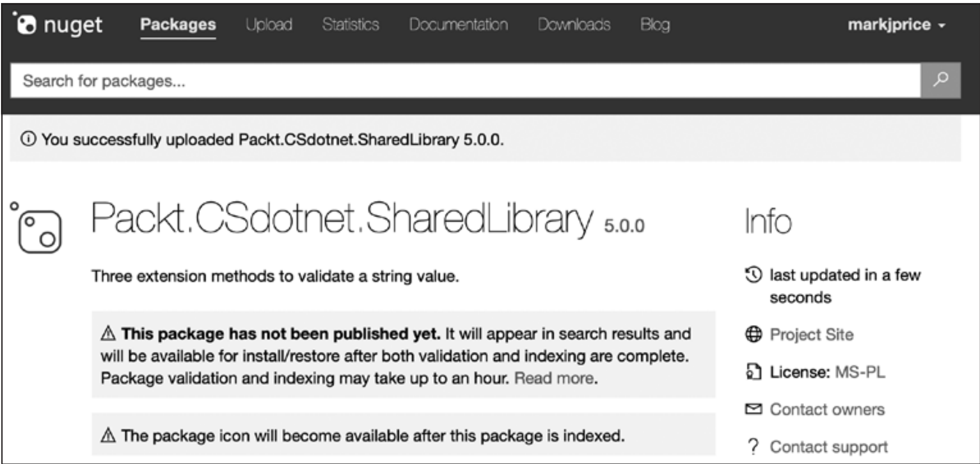


Рис. 7.7. Сообщение о загрузке пакета NuGet

При возникновении ошибки просмотрите файл проекта на наличие ошибок или прочитайте дополнительную информацию о формате PackageReference на сайте https://docs.microsoft.com/ru-ru/nuget/reference/msbuild-targets.

Тестирование пакета

Теперь загруженный пакет необходимо протестировать, сославшись на него в проекте `AssembliesAndNamespaces`.

1. Откройте файл `AssembliesAndNamespaces.csproj`.
2. Измените код файла так, чтобы добавить ссылку на ваш пакет (или можете использовать команду `dotnet add package`), как показано ниже (выделено полужирным шрифтом).

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="newtonsoft.json"
      Version="12.0.3" />
    <PackageReference Include="packt.csdotnet.sharedlibrary"
      Version="5.0.0" />
  </ItemGroup>
</Project>
```

В предыдущем коде я использовал ссылку на свой пакет. При успешной загрузке вы должны применить ссылку на собственный пакет.

3. Для восстановления пакетов и компиляции консольного приложения введите следующую команду: `dotnet build`.
4. Отредактируйте файл `Program.cs`, чтобы импортировать пространство имен `Packt.Shared`.
5. В методе `Main` предложите пользователю ввести некоторые строковые значения, а затем проверьте их с помощью методов расширения в пакете, как показано ниже:

```
Write("Enter a color value in hex: ");
string hex = ReadLine();
WriteLine("Is {0} a valid color value? {1}",
  arg0: hex, arg1: hex.IsValidHex());

Write("Enter a XML element: ");
string xmlTag = ReadLine();
WriteLine("Is {0} a valid XML element? {1}",
  arg0: xmlTag, arg1: xmlTag.IsValidXmlTag());

Write("Enter a password: ");
string password = ReadLine();
WriteLine("Is {0} a valid password? {1}",
  arg0: password, arg1: password.IsValidPassword());
```

6. Запустите приложение и проанализируйте результат:

```
Enter a color value in hex: 00ffc8
Is 00ffc8 a valid color value? True
Enter an XML element: <h1 class="<" />
Is <h1 class="<" /> a valid XML element? False
Enter a password: secretsauce
Is secretsauce a valid password? True
```

Перенос приложений с .NET Framework на .NET 5

Если вы ранее разрабатывали на .NET Framework, то можете иметь существующие приложения, и вам интересно, стоит ли переносить их на .NET 5. Подумайте, является ли перенос правильным решением для вашего кода, потому что иногда правильнее всего ничего не переносить.

Что означает перенос

Платформа .NET Core прекрасно поддерживает следующие типы приложений в операционных системах Windows, macOS и Linux:

- веб-приложения *ASP.NET Core MVC*;
- веб-сервисы *ASP.NET Core Web API* (REST/HTTP);
- *консольные* приложения.

Платформа .NET 5 прекрасно поддерживает следующие типы приложений в операционной системе Windows:

- приложения *Windows Forms*;
- приложения *Windows Presentation Foundation (WPF)*;
- приложения *Universal Windows Platform (UWP)*.

Приложения UWP могут быть разработаны на языках C++, JavaScript, C# и Visual Basic с помощью специальной версии .NET Core. Более подробную информацию можно найти на сайте https://docs.microsoft.com/ru-ru/windows/uwp/get-started/universal-application-platform-guide.

Платформа .NET 5 не поддерживает следующие типы устаревших приложений Microsoft, а также многие другие:

- веб-приложения *ASP.NET Web Forms*;
- сервисы *Windows Communication Foundation*;
- приложения *Silverlight*.

Приложения Silverlight и ASP.NET Web Forms никогда нельзя будет перенести на .NET Core. Однако существующие приложения Windows Forms и WPF-приложения можно перенести на .NET 5 в операционной системе Windows в целях

использования новых API и повышения производительности. Существующие веб-приложения ASP.NET MVC и веб-сервисы Web API в ASP.NET могут быть перенесены на .NET 5 в операционных системах Windows, Linux или macOS.

Стоит ли переносить

Так *следует* ли выполнять перенос, если это *технически возможно*? Какие преимущества вы при этом получаете? Ниже перечислены некоторые из них.

- *Развертывание в Linux или Docker для веб-приложений и веб-сервисов*. Эти операционные системы просты в использовании и экономичны в качестве платформ веб-приложений и веб-сервисов, особенно по сравнению с операционной системой Windows Server.
- *Удаление зависимости от IIS и System.Web.dll*. Даже если вы продолжите развертывание на Windows Server, платформа ASP.NET Core может быть размещена на простых и высокопроизводительных веб-серверах Kestrel (или других).
- *Инструменты командной строки*. Инструменты, которые разработчики и администраторы используют для автоматизации своих задач, часто создаются как консольные приложения. Возможность запуска единого инструмента кросс-платформенно очень полезна.

Сравнение .NET Framework и .NET 5

В табл. 7.8 приведены три важных различия этих платформ.

Таблица 7.8

.NET 5	.NET Framework
Распространяется в виде NuGet-пакетов, поэтому каждое приложение может быть развернуто с помощью отдельной локальной версии платформы .NET, которая необходима для запуска	Распространяется в виде общесистемного набора сборок (если буквально, то глобального кэша сборок (Global Assembly Cache, GAC))
Разделена на небольшие разноуровневые компоненты, поэтому необходимо выполнить минимальное развертывание	Требует полноценного монолитного развертывания
Удалены неиспользуемые компоненты. Помимо удаления старых технологий, таких как ASP.NET Web Forms, отсутствуют функции, привязанные к той или иной платформе, например AppDomains, .NET Remoting и бинарная сериализация	Помимо технологий в .NET 5, здесь сохраняются некоторые более старые технологии, такие как ASP.NET Web Forms

Анализатор переносимости .NET

Специалисты корпорации Microsoft создали полезный инструмент, который можно использовать для оценки возможности переноса существующих приложений и подготовки соответствующего отчета, — .NET Portability Analyzer. Демонстрация процесса работы с ним доступна по адресу https://channel9.msdn.com/Blogs/Seth-Juarez/A-Brief-Look-at-the-NET-Portability-Analyzer.

Использование библиотек, не скомпилированных для .NET Standard

Большинство существующих пакетов NuGet можно использовать с .NET 5, даже если они не скомпилированы для .NET Standard. Если вы нашли пакет, который официально не поддерживает .NET Standard, как показано на его веб-странице nuget.org, то не стоит беспокоиться. Просто проверьте, работает ли он.

Полезные пакеты NuGet можно найти по адресу https://www.nuget.org/packages.

Например, существует пакет пользовательских коллекций для обработки матриц, созданный компанией Dialect Software LLC, документацию которого можно найти по ссылке https://www.nuget.org/packages/DialectSoftware.Collections.Matrix/.

Последний раз пакет обновлялся в 2013 году, задолго до появления .NET 5, поэтому данный пакет был создан для .NET Framework. Поскольку такой сборочный пакет задействует только API, доступные в .NET Standard, он может применяться в проекте .NET 5.

Давайте попробуем его использовать.

1. Откройте проект `AssembliesAndNamespaces.csproj`.
2. Добавьте элемент `<PackageReference>` для пакета программного обеспечения Dialect:

```
<PackageReference
  Include="dialectsoftware.collections.matrix"
  Version="1.0.0" />
```

3. На панели TERMINAL (Терминал) восстановите зависимые пакеты с помощью следующей команды:

```
dotnet restore
```

4. Откройте файл `Program.cs` и добавьте операторы для импорта пространств имен `DialectSoftware.Collections` и `DialectSoftware.Collections.Generics`.
5. Добавьте операторы для создания экземпляров `Axis` и `Matrix<T>`, заполните их значениями и выведите:

```
var x = new Axis("x", 0, 10, 1);
var y = new Axis("y", 0, 4, 1);

var matrix = new Matrix<long>(new[] { x, y });
for (int i = 0; i < matrix.Axes[0].Points.Length; i++)
{
  matrix.Axes[0].Points[i].Label = "x" + i.ToString();
}

for (int i = 0; i < matrix.Axes[1].Points.Length; i++)
{
  matrix.Axes[1].Points[i].Label = "y" + i.ToString();
}

foreach (long[] c in matrix)
{
  matrix[c] = c[0] + c[1];
}

foreach (long[] c in matrix)
{
  WriteLine("{0},{1} ({2},{3}) = {4}",
    matrix.Axes[0].Points[c[0]].Label,
    matrix.Axes[1].Points[c[1]].Label,
    c[0], c[1], matrix[c]);
}
```

6. Запустите консольное приложение, проанализируйте результат и обратите внимание на предупреждающее сообщение:

```
warning NU1701: Package 'DialectSoftware.Collections.Matrix
1.0.0' was restored using '.NETFramework,Version=v4.6.1,
.NETFramework,Version=v4.6.2, .NETFramework,Version=v4.7,
.NETFramework,Version=v4.7.1, .NETFramework,Version=v4.7.2,
.NETFramework,Version=v4.8' instead of the project target framework
'net5.0'. This package may not be fully compatible with your project.
x0,y0 (0,0) = 0
x0,y1 (0,1) = 1
x0,y2 (0,2) = 2
x0,y3 (0,3) = 3
...and so on.
```

Этот пакет был создан до того, как появилась .NET 5, и компилятор и среда выполнения не могут знать, сработает ли он, и потому отображают предупреждения; однако, поскольку пакет вызывает только API, совместимые с NET Standard, он успешно работает.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 7.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. В чем разница между пространством имен и сборкой?
2. Как вы ссылаетесь на другой проект в файле `.csproj`?
3. В чем преимущество такого инструмента, как ILSpy?
4. Какой тип .NET представлен псевдонимом `float` в C#?
5. Какой инструмент следует использовать перед переносом приложения с .NET Framework на .NET 5?
6. В чем разница между платформозависимым и автономным развертыванием приложений .NET?
7. Что такое RID?
8. В чем разница между командами `dotnet pack` и `dotnet publish`?
9. Какие типы приложений, написанных для .NET Framework, можно перенести на .NET 5?
10. Можете ли вы использовать пакеты, написанные для .NET Framework, с .NET 5?

Упражнение 7.2. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- перенос кода в .NET Core из .NET Framework: https://docs.microsoft.com/ru-ru/dotnet/core/porting/;
- обзор публикации приложений .NET Core: https://docs.microsoft.com/ru-ru/dotnet/core/deploying/;
- .NET Blog: https://devblogs.microsoft.com/dotnet/;
- учебное пособие: создание шаблона элемента: https://docs.microsoft.com/ru-ru/dotnet/core/tutorials/cli-templates-create-item-template;
- что нужно знать разработчикам .NET: https://www.hanselman.com/blog/WhatNETDevelopersOughtToKnowToStartIn2017.aspx;
- CoreFX README.md: https://github.com/dotnet/corefx/blob/master/Documentation/README.md.

Резюме

Мы обсудили взаимосвязь сборок и пространств имен, а также возможности переноса существующих кодовых баз, написанных на .NET Framework, публикации ваших приложений и библиотек и развертывания вашего кода независимо от платформы.

Далее вы узнаете о некоторых распространенных типах библиотек базовых классов, включенных в .NET 5.

8 Работа с распространенными типами .NET

Данная глава посвящена распространенным типам .NET. В их число входят типы для работы с числами, текстом, коллекциями, доступом к сети, отражением, атрибутами, для улучшения работы с интервалами, индексами и диапазонами, а также интернационализацией.

В этой главе:

- работа с числами;
- работа с текстом;
- сопоставление шаблонов с использованием регулярных выражений;
- хранение данных с помощью коллекций;
- работа с интервалами, индексами и диапазонами;
- работа с сетевыми ресурсами;
- работа с типами и атрибутами;
- работа с изображениями;
- интернационализация кода.

Работа с числами

Числа — один из наиболее распространенных типов данных. В табл. 8.1 перечислены наиболее распространенные типы в .NET для работы с числами.

Более подробную информацию можно найти на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/numerics.

В .NET входили 32-разрядные типы с плавающей запятой и 64-разрядные типы `double`, начиная с .NET Framework 1.0. Спецификация IEEE 754 также определяет 16-битный стандарт с плавающей запятой. Машинное обучение и другие

алгоритмы выиграют от этого меньшего числового типа с меньшей точностью, поэтому Microsoft в .NET 5 добавила тип `System.Half`.

Таблица 8.1

Пространство имен	Тип	Описание
System	SByte, Int16, Int32, Int64	Целочисленные типы: ноль, положительные и отрицательные целые числа
System	Byte, UInt16, UInt32, UInt64	Кардинальные типы: ноль и положительные целые числа
System	Single, Double	Вещественный типы: числа с плавающей запятой
System	Decimal	Точный вещественный тип: используется для научных, инженерных или финансовых сценариев использования
System.Numerics	BigInteger, Complex, Quaternion	Целые числа произвольной величины, комплексные числа и кватернионы

В настоящее время язык C# не определяет псевдоним `Half`, поэтому вам необходимо использовать имя типа .NET `Half`. Однако это в дальнейшем может быть изменено.

Более подробную информацию о типе Half можно прочитать на сайте https://devblogs.microsoft.com/dotnet/introducing-the-half-type/.

Большие целые числа

Наибольшее целое число, которое может храниться в типах .NET и у которого есть псевдоним в C#, составляет около 18,5 квинтиллиона. Для его хранения используется тип `unsigned long` (беззнаковое длинное число). Но что делать, если вам понадобится хранить бо́льшие числа?

1. Создайте в папке Chapter08 консольное приложение `WorkingWithNumbers`.
2. Сохраните рабочую область под именем `Chapter08` и добавьте в нее приложение `WorkingWithNumbers`.
3. В файле `Program.cs` добавьте следующий оператор для импорта простанства имен `System.Numerics`:

```
using System.Numerics;
```

4. В метод `Main` добавьте операторы для вывода наибольшего значения `ulong` и числа из 30 знаков, используя `BigInteger`:

```
var largest = ulong.MaxValue;
WriteLine($"{largest,40:N0}");
```

```
var atomsInTheUniverse =
    BigInteger.Parse("123456789012345678901234567890");
WriteLine($"{atomsInTheUniverse,40:N0}");
```

Число `40` в коде формата означает выравнивание 40 символов по правому краю, поэтому оба числа выровнены по правому краю. `N0` означает разделитель тысяч и нулевых десятичных знаков.

5. Запустите консольное приложение и проанализируйте результат:

```
              18,446,744,073,709,551,615
123,456,789,012,345,678,901,234,567,890
```

Комплексные числа

Комплексное число может быть выражено следующим образом: $a + bi$, где a и b — действительные числа, а i — мнимая единица, где $i^2 = -1$. Если действительная часть равна нулю, то это мнимое число. Если мнимая часть равна нулю, то это действительное число.

Комплексные числа используются во многих областях *STEM* (science, technology, engineering, mathematics — «наука, технология, инженерия, математика»). Кроме того, обратите внимание, что сложение для них производится путем независимого сложения действительных и мнимых частей слагаемых:

```
(a + bi) + (c + di) = (a + c) + (b + d)i
```

Рассмотрим комплексные числа.

1. В метод `Main` введите операторы для добавления двух комплексных чисел, как показано ниже:

```
var c1 = new Complex(4, 2);
var c2 = new Complex(3, 7);
var c3 = c1 + c2;
WriteLine($"{c1} added to {c2} is {c3}");
```

2. Запустите консольное приложение и проанализируйте результат:

```
(4, 2) added to (3, 7) is (7, 9)
```

Кватернионы — система чисел, расширяющая систему комплексных чисел. Они образуют четырехмерную ассоциативную нормированную алгебру с делением над полем действительных чисел.

Я тоже не очень хорошо в этом разбираюсь. Не беспокойтесь! Мы не будем использовать их при написании кода. Достаточно того, что кватернионы применяются для записи пространственных вращений, так что их используют в движках видеоигр, симуляторах и системах управления полетами.

Работа с текстом

Другой весьма распространенный тип данных для переменных — это текст. Самые распространенные типы .NET Standard для работы с текстом перечислены в табл. 8.2.

Таблица 8.2

Пространство имен	**Тип**	**Описание**
System	Char	Хранит один текстовый символ
System	String	Хранит несколько текстовых символов
System.Text	StringBuilder	Эффективно управляет строками
System.Text.RegularExpressions	Regex	Эффективно сопоставляет строки с шаблонами

Извлечение длины строки

Рассмотрим некоторые задачи при работе с текстом, например, вам необходимо определить длину фрагмента текста, хранящегося в строковой переменной.

1. Создайте в папке Chapter08 проект консольного приложения `WorkingWithText` и добавьте его в рабочую область `Chapter08`.
2. Выберите View ▸ Command Palette (Вид ▸ Палитра команд), введите и выберите OmniSharp: Select Project, а затем выберите WorkingWithText.
3. В проекте `WorkingWithText`, в файле `Program.cs`, в методе `Main` добавьте операторы, чтобы определить переменную для хранения названия города Лондона, а затем запишите его имя и длину в консоль:

```
string city = "London";
WriteLine($"{city} is {city.Length} characters long.");
```

4. Запустите консольное приложение и проанализируйте результат:

```
London is 6 characters long.
```

Извлечение символов строки

Класс `string` для хранения текста использует внутренний массив `char`. Для этого класса определен индексатор, поэтому мы можем задействовать синтаксис массива для чтения хранящихся символов.

1. Добавьте оператор для записи символов в первой и третьей позиции строковой переменной:

```
WriteLine($"First char is {city[0]} and third is {city[2]}.");
```

2. Запустите консольное приложение и проанализируйте результат:

```
First char is L and third is n.
```

Нумерация элементов массива начинается с нуля, поэтому третий символ находится по индексу 2.

Разделение строк

Иногда требуется разделить текст в местах, где используется определенный символ, например запятая.

1. Добавьте операторы, определяющие одну строку с названиями городов, разделенными запятыми, а затем использующие метод `Split` с указанием запятой в качестве символа-разделителя и в итоге перечисляющие возвращенный массив значений `string`:

```
string cities = "Paris,Berlin,Madrid,New York";

string[] citiesArray = cities.Split(',');

foreach (string item in citiesArray)
{
  WriteLine(item);
}
```

2. Запустите консольное приложение и проанализируйте результат:

```
Paris
Berlin
Madrid
New York
```

Далее в этой главе вы научитесь обрабатывать более сложные сценарии. Например, для случая, когда строковая переменная содержит такие названия фильмов: `""Monsters, Inc.","I, Tonya","Lock, Stock and Two Smoking Barrels""`.

Извлечение фрагмента строки

В некоторых случаях может потребоваться извлечь часть текста. У метода `IndexOf` есть девять перегрузок, возвращающих позицию индекса указанного символа или строки. Метод `Substring` содержит две перегрузки:

- `Substring(startIndex, length)` — возвращает подстроку, которая начинается с `startIndex` и содержит последующие `length` символов;

- `Substring(startIndex)` — возвращает подстроку, которая начинается с `startIndex` и содержит все символы до конца строки.

Рассмотрим пример.

1. Добавьте операторы, чтобы сохранить полное имя человека в строковой переменной с символом пробела между именем и фамилией, обнаружить позицию пробела и извлечь по отдельности имя и фамилию, например, так:

```
string fullName = "Alan Jones";
int indexOfTheSpace = fullName.IndexOf(' ');

string firstName = fullName.Substring(
  startIndex: 0, length: indexOfTheSpace);

string lastName = fullName.Substring(
  startIndex: indexOfTheSpace + 1);

WriteLine($"{lastName}, {firstName}");
```

2. Запустите консольное приложение и проанализируйте результат:

```
Jones, Alan
```

Если формат исходного полного имени был иным, к примеру `"Lastname, Firstname"`, то код будет несколько отличаться. В качестве дополнительного упражнения попробуйте написать несколько операторов, которые изменят входные данные `"Jones, Alan"` на `"Alan Jones"`.

Поиск содержимого в строках

Иногда необходимо проверить наличие определенных символов в начале, в составе или в конце строки. Это возможно с помощью методов `StartsWith`, `EndsWith` и `Contains`.

1. Добавьте операторы для хранения строкового значения, а затем проверьте, начинается ли оно с некоторых строковых значений или содержит их, как показано ниже:

```
string company = "Microsoft";
bool startsWithM = company.StartsWith("M");
bool containsN = company.Contains("N");
WriteLine($"Starts with M: {startsWithM}, contains an N:
{containsN}");
```

2. Запустите консольное приложение и проанализируйте результат:

```
Starts with M: True, contains an N : False
```

Конкатенация строк, форматирование и прочие члены типа string

В табл. 8.3 перечислены некоторые другие члены типа `string`.

Таблица 8.3

Член	Описание
Trim, TrimStart и TrimEnd	Удаляют пробельные символы, такие как пробел, табуляция и возврат каретки в начале и/или конце переменной string
ToUpper и ToLower	Преобразуют символы в переменной string в прописные или строчные
Insert и Remove	Добавляют или удаляют указанный текст в переменной string
Replace	Замещает указанный текст
string.Concat	Конкатенирует две переменные string. Оператор + вызывает этот метод, если используется между переменными string
string.Join	Конкатенирует одну или несколько переменных string с указанным символом между ними
string.IsNullOrEmpty	Проверяет, является ли переменная string пустой ("") или null
string.IsNullOrWhitespace	Проверяет, является ли строковая переменная null или пробелом; то есть сочетание любого количества горизонтальных и вертикальных интервальных символов, например табуляция, пробел, возврат каретки, перевод строки и т. д.
string.Empty	Можно задействовать вместо выделения памяти каждый раз, когда вы используете литеральное значение string, применяя пару двойных кавычек без содержимого ("")
string.Format	Более старый альтернативный метод интерполяции строк для вывода форматированных строковых переменных, в котором вместо именованных параметров используется позиционирование

Некоторые из предыдущих методов — статические. Это значит, что метод может быть вызван только из типа, а не из экземпляра переменной. В табл. 8.3 я указал статические методы, поставив перед ними префикс `string.`, например `string.Format`.

Далее рассмотрим некоторые из перечисленных методов.

1. Добавьте операторы, чтобы взять массив строковых значений и заново объединить их в одну строку с разделителями, используя метод `Join`, как показано ниже:

```
string recombined = string.Join(" => ", citiesArray);
WriteLine(recombined);
```

2. Запустите консольное приложение и проанализируйте результат:

```
Paris => Berlin => Madrid => New York
```

3. Добавьте операторы для использования позиционированных параметров и синтаксиса интерполяции строк, используя одни и те же переменные, как показано ниже:

```
string fruit = "Apples";
decimal price = 0.39M;
DateTime when = DateTime.Today;

WriteLine($"{fruit} cost {price:C} on {when:dddd}.");
WriteLine(string.Format("{0} cost {1:C} on {2:dddd}.",
  fruit, price, when));
```

4. Запустите консольное приложение и проанализируйте результат:

```
Apples cost J0.39 on Thursdays.
Apples cost J0.39 on Thursdays.
```

Эффективное создание строк

Вы можете конкатенировать (сцепить) две строки, чтобы создать новую переменную типа `string`, используя метод `String.Concat` или с помощью оператора `+`. Но этот способ не рекомендуется, поскольку .NET необходимо будет создать совершенно новую переменную типа `string` в памяти.

Последствий может и не быть, если вы объединяете только две переменные типа `string`, но если конкатенацию проводить в цикле, то операция может весьма негативно повлиять на производительность и использование памяти.

В главе 13 вы научитесь эффективно конкатенировать строковые переменные с помощью типа `StringBuilder`.

Сопоставление шаблонов с использованием регулярных выражений

Регулярные выражения полезны для проверки на допустимость ввода пользователя. Они очень эффективны и могут становиться крайне сложными. Почти все языки программирования поддерживают регулярные выражения и применяют универсальный набор специальных символов для их определения.

1. Создайте проект консольного приложения `WorkingWithRegularExpressions`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.

2. В начале файла импортируйте следующее пространство имен:

```
using System.Text.RegularExpressions;
```

Проверка цифр, введенных как текст

Рассмотрим пример проверки ввода цифр.

1. В метод `Main` добавьте операторы, чтобы пользователь ввел свой возраст, а затем с помощью регулярного выражения убедитесь, что он правильный, как показано ниже:

```
Write("Enter your age: ");
string input = ReadLine();

var ageChecker = new Regex(@"\d");

if (ageChecker.IsMatch(input))
{
  WriteLine("Thank you!");
}
else
{
  WriteLine($"This is not a valid age: {input}");
}
```

   Символ `@` перед строкой отключает возможность использования escape-последовательностей в строке. Escape-последовательности предваряются префиксом в виде обратного слеша (`\`). К примеру, escape-последовательность `\t` обозначает горизонтальный отступ (табуляцию), а `\n` — новую строку.

   При использовании регулярных выражений нам нужно отключить эту функцию.

   После того как escape-последовательности отключены с помощью символа `@`, движок регулярных выражений может их интерпретировать. Например, escape-последовательность `\d` обозначает цифру. Далее вы узнаете еще больше регулярных выражений с префиксом обратного слеша.

2. Запустите консольное приложение, введите целое число, например, для возраста — `34` и проанализируйте результат:

```
Enter your age: 34
Thank you!
```

3. Снова запустите консольное приложение, введите `carrots` и проанализируйте результат:

```
Enter your age: carrots
This is not a valid age: carrots
```

4. Снова запустите консольное приложение, введите `bob30smith` и проанализируйте результат:

```
Enter your age: bob30smith
Thank you!
```

Регулярное выражение, которое мы использовали, — `\d` — обозначает одну цифру. Однако оно не ограничивает ввод значения *до* и *после* цифры. Это регулярное выражение на русском языке можно объяснить так: «Введите любые пришедшие вам в голову символы, используя хотя бы одну цифру».

5. Измените регулярное выражение на `^\d$`, например, так:

```
var ageChecker = new Regex(@"^\d$");
```

6. Перезапустите приложение. Теперь не допускаются любые значения, кроме одной цифры.

Мы же хотим, чтобы можно было указать одну цифру или больше. В этом случае нужно добавить символ `+` (плюс) после выражения `\d`.

7. Измените регулярное выражение следующим образом:

```
var ageChecker = new Regex(@"^\d+$");
```

8. Запустите приложение и посмотрите, что регулярное выражение теперь допускает ввод только нуля или положительных целых чисел любой длины.

Синтаксис регулярных выражений

В табл. 8.4 приведены несколько универсальных комбинаций *символов*, которые вы можете использовать в регулярных выражениях.

Таблица 8.4

Символ	**Значение**	**Символ**	**Значение**
^	Начало ввода	$	Конец ввода
\d	Цифра	\D	Не цифра (любой символ, не являющийся цифрой)
\w	Пробел	\W	Не пробел (любой символ, не являющийся пробелом)
[A-Za-z0-9]	Диапазон символов	\^	Символ ^ (каретки)
[aeiou]	Набор символов	[^aeiou]	Любой символ, кроме входящего в набор
.	Любой одиночный символ	\.	Символ . (точка)

Чтобы указать символ Unicode, используйте символ \u, за которым следуют четыре символа, указывающие его номер. Например, \u00c0 — это символ À. Более подробную информацию можно найти на сайте https://www.regular-expressions.info/unicode.html.

В табл. 8.5 приведены некоторые *квантификаторы* регулярных выражений, влияющие на предыдущие символы.

Таблица 8.5

Символ	Значение	Символ	Значение
+	Один или более	?	Один или ноль
{3}	Ровно три	{3,5}	От трех до пяти
{3,}	Минимум три	{,3}	Более трех

Примеры регулярных выражений

В табл. 8.6 я привел примеры некоторых регулярных выражений.

Таблица 8.6

Выражение	Значение
\d	Одна цифра где-либо во вводе
a	Символ a где-либо во вводе
Bob	Слово Bob где-либо во вводе
^Bob	Слово Bob в начале ввода
Bob$	Слово Bob в конце ввода
^\d{2}$	Строго две цифры
^[0-9]{2}$	Строго две цифры
^[A-Z]{4,}$	Не менее четырех прописных букв
^[A-Za-z]{4,}$	Не менее четырех прописных или строчных букв
^[A-Z]{2}\d{3}$	Строго две прописные буквы и три цифры
^[A-Za-z\u00c0-\u017e]+$	Как минимум одна прописная или строчная английская буква в наборе символов ASCII или европейская буква из набора символов Unicode: ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ × ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö ÷ øùúûüýþÿıŒœŠšŸŽž
^d.g$	Буква d, затем любой символ, а затем буква g, то есть совпадет со словами типа dig, dog и другими с любым символом между буквами d и g
^d\.g$	Буква d, затем точка (.), а затем буква g, поэтому совпадет только с последовательностью d.g

Применяйте регулярные выражения для проверки пользовательского ввода. Такие же регулярные выражения можно задействовать и в других языках, например в JavaScript и Python.

Разбивка сложных строк, разделенных запятыми

Ранее в этой главе вы узнали, как разделить простую строковую переменную с помощью запятой. Но как быть с таким случаем?

```
"Monsters, Inc.","I, Tonya","Lock, Stock and Two Smoking Barrels"
```

В начале и в конце каждого заголовка фильма используются двойные кавычки. Мы можем использовать их, чтобы понять, разделять запятыми в конкретном месте или нет. Метод `Split` так не умеет, поэтому мы можем задействовать вместо него регулярное выражение.

Чтобы включить двойные кавычки в строковое значение, мы ставим перед ними обратную косую черту (обратный слеш).

1. Добавьте операторы для хранения сложной переменной `string`, разделенной запятыми, а затем разделите ее с помощью «глупого» метода `Split`, как показано ниже:

```
string films = "\"Monsters, Inc.\",\"I, Tonya\",\"Lock, Stock and
Two Smoking Barrels\"";

string[] filmsDumb = films.Split(',');

WriteLine("Dumb attempt at splitting:");
foreach (string film in filmsDumb)
{
  WriteLine(film);
}
```

2. Добавьте следующие операторы для определения регулярного выражения, чтобы разделить заголовки фильма по-умному и вывести их на экран:

```
var csv = new Regex(
  "(?:^|,)(?=[^\"]|(\")?)\"?((?(1)[^\"]*|[^,\"]*))\"?(?=,|$)");

MatchCollection filmsSmart = csv.Matches(films);

WriteLine("Smart attempt at splitting:");
foreach (Match film in filmsSmart)
{
  WriteLine(film.Groups[2].Value);
}
```

3. Запустите консольное приложение и проанализируйте результат:

```
Dumb attempt at splitting:
"Monsters
 Inc."
"I
 Tonya"
"Lock
 Stock and Two Smoking Barrels"
Smart attempt at splitting:
Monsters, Inc.
I, Tonya
Lock, Stock and Two Smoking Barrels
```

Тема регулярных выражений достаточно объемная. О них было написано множество книг. Дополнительную информацию можно найти на сайте https://www.regular-expressions.info.

Улучшение производительности регулярных выражений

Для работы с регулярными выражениями типы .NET используются на всей платформе .NET и во многих приложениях, созданных с ней. Они оказывают значительное влияние на производительность, но до сих пор Microsoft не уделяла особого внимания их оптимизации.

В .NET 5 пространство имен `System.Text.RegularExpressions` было переписано внутри, чтобы добиться максимальной производительности. Стандартные тесты для регулярных выражений, использующие такие методы, как IsMatch, теперь в пять раз быстрее. И что самое важное, вам не нужно менять код, чтобы получить преимущества!

Дополнительную информацию об улучшении производительности вы можете найти по адресу https://devblogs.microsoft.com/dotnet/regex-performance-improvements-in-net-5/.

Хранение данных с помощью коллекций

Коллекции — еще один распространенный тип данных. Если в переменной требуется сохранить несколько значений, то вы можете использовать коллекции.

Коллекция — это структура данных в памяти, позволяющая управлять несколькими элементами различными способами, хотя все коллекции имеют общие функции.

В табл. 8.7 перечислены наиболее распространенные типы .NET Standard для работы с коллекциями.

Таблица 8.7

Пространство имен	**Тип**	**Описание**
System.Collections	IEnumerable, IEnumerable<T>	Интерфейсы и базовые классы, используемые коллекциями
System.Collections. Generic	List<T>, Dictionary<T>, Queue<T>, Stack<T>	Типы из этой сборки и пространства имен стали применяться в версии C# 2.0 с .NET Framework 2.0, поскольку позволяют указать тип, который вы хотите сохранить, с помощью параметра типа-дженерика (что более безопасно, быстро и эффективно)
System.Collections. Concurrent	BlockingCollection, ConcurrentDictionary, ConcurrentQueue	Типы из этой сборки и пространства имен безопасны для использования в многопоточных приложениях
System.Collections. Immutable	ImmutableArray, ImmutableDictionary, ImmutableList, ImmutableQueue	Типы из этой сборки и пространства имен предназначены для сценариев, в которых содержимое коллекции никогда не должно изменяться, хотя они могут создавать измененные коллекции как новый экземпляр

Более подробную информацию о коллекциях можно найти на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/collections.

Общие свойства коллекций

Коллекции реализуют интерфейс `ICollection`. Это значит, все коллекции содержат свойство `Count`, возвращающее количество объектов, содержащихся в коллекции.

Например, если бы у нас была коллекция `passengers`, то мы могли бы сделать следующее:

```
int howMany = passengers.Count;
```

Коллекции реализуют интерфейс `IEnumerable`, то есть все коллекции имеют метод `GetEnumerator`, возвращающий объект `IEnumerator`. Это значит, что возвращаемый объект содержит метод `MoveNext` и свойство `Current`, чтобы их можно было перебирать с помощью оператора `foreach`.

Например, чтобы выполнить действие для каждого объекта в коллекции `passengers`, используется следующий код:

```
foreach (var passenger in passengers)
{
  // операции с каждым пассажиром
}
```

Чтобы понимать коллекции, полезно изучить наиболее распространенные интерфейсы, ими реализуемые (рис. 8.1).

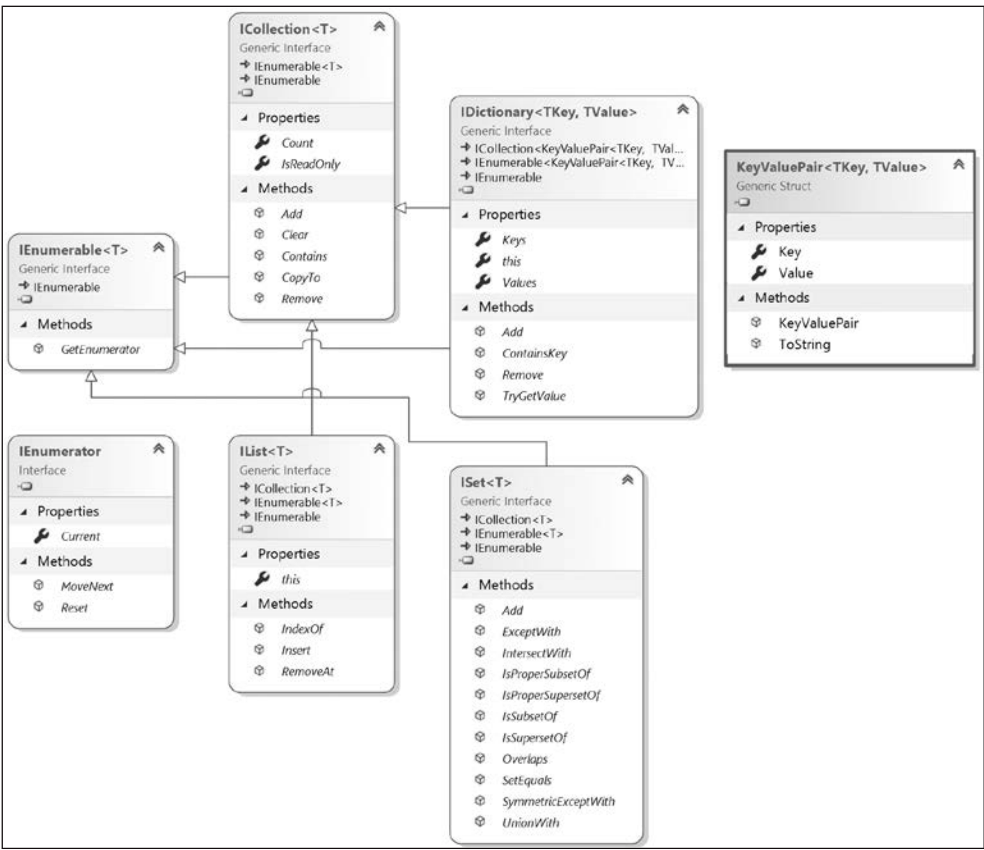


Рис. 8.1. Общие интерфейсы, которые реализуют коллекции

Списки, то есть тип, реализующий интерфейс `IList`, являются *упорядоченными коллекциями*. Как следует из предыдущей схемы, интерфейс `IList<T>` включает интерфейс `ICollection<T>`, поэтому они содержат свойство `Count` и метод `Add` для добавления элемента в конец коллекции, а также метод `Insert` для вставки элемента в указанную позицию и `RemoveAt` для удаления элемента в указанной позиции.

Выбор коллекции

Существует несколько различных категорий коллекции, которые следует выбирать для разных ситуаций: списки, словари, стеки, очереди, множества и многие другие узкоспециализированные коллекции.

Списки

Списки весьма уместны, если вы хотите вручную управлять порядком элементов в коллекции. Каждому элементу в списке автоматически присваивается уникальный индекс (позиция). Элементы могут быть любого типа, определяемого типом `T`, и дублироваться. Индексы имеют тип `int` и начинаются с 0, поэтому первый элемент в списке, как и в массиве, имеет индекс 0 (табл. 8.8).

Таблица 8.8

Индекс	**Элемент**
0	London
1	Paris
2	London
3	Sydney

Если новый элемент (к примеру, *Santiago*) добавить между элементами *London* и *Sydney*, то индекс элемента *Sydney* автоматически увеличится. Исходя из этого, следует учитывать, что индекс объекта может измениться после добавления или удаления элементов (табл. 8.9).

Таблица 8.9

Индекс	**Элемент**
0	London
1	Paris
2	London
3	Santiago
4	Sydney

Словари

Словари будут удобны в случае, если каждое *значение* (или элемент) имеет уникальное подзначение (или искусственно созданное значение), которое в дальнейшем

можно использовать в качестве *ключа* для быстрого поиска значения в коллекции. Ключ должен быть уникальным. К примеру, при сохранении списка персон вы можете применить в качестве ключа номера паспортов.

Представьте, что ключ — это своего рода указатель в словаре в реальном мире. Так вы можете быстро найти определение слова, поскольку слова (к примеру, ключи) сортируются, и, если мы ищем определение слова *manatee*, то открыли бы словарь в середине, так как буква *M* находится в середине алфавита.

Словари в программировании ведут себя аналогичным образом при выполнении поиска. Они реализуют интерфейс `IDictionary<TKey, TValue>`.

И ключ, и значение могут быть любого типа, определенного `TKey` и `TValue`. В примере `Dictionary<string, Person>` в качестве ключа используется `string`, а в качестве значения — экземпляр `Person`. `Dictionary<string, string>` использует значения `string` и для ключа, и для значения (табл. 8.10).

Таблица 8.10

Ключ	**Значение**
BSA	Bob Smith
MW	Max Williams
BSB	Bob Smith
AM	Amir Mohammed

Стеки

Стеки удобно использовать в тех случаях, если вы хотите реализовать поведение *«последним пришел — первым вышел»* (last-in, first-out, LIFO). С помощью стека вы можете получить доступ только к одному элементу в его начале, хотя можете перечислить весь стек элементов. Вы не можете получить доступ, например, ко второму элементу в стеке.

Так, текстовые редакторы используют стек для хранения последовательности действий, которые вы недавно выполнили, а затем, когда вы нажимаете сочетание клавиш Ctrl+Z, программа отменяет последнее действие в стеке, затем предпоследнее и т. д.

Очереди

Очереди удобны, если вы хотите реализовать поведение *«первым пришел — первым вышел»* (first-in, first-out, FIFO). С помощью очереди вы можете получить доступ только к одному элементу в ее начале, хотя можете перечислить всю

цепочку элементов. Вы не можете получить доступ, например, ко второму элементу в очереди.

Так, фоновые процессы используют очередь для обработки заданий в том порядке, в котором те поступают, — точно так же, как получают услугу люди, стоящие в очереди в почтовом отделении.

Множества

Множества — прекрасный выбор, если вы хотите выполнять операции над множествами между двумя коллекциями. Например, у вас могут быть две коллекции с названиями городов и вы хотите выяснить, какие имена используются в обоих множествах (так называемое *пересечение* между множествами). Элементы множества должны быть уникальными.

Работа со списками

Работу со списками рассмотрим на следующем примере.

1. Создайте консольное приложение `WorkingWithLists`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.
2. В начале файла импортируйте следующие пространства имен:

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

3. В методе `Main` введите приведенный ниже код. Он демонстрирует некоторые из распространенных способов работы со списками:

   ```
   var cities = new List<string>();
   cities.Add("London");
   cities.Add("Paris");
   cities.Add("Milan");

   WriteLine("Initial list");
   foreach (string city in cities)
   {
     WriteLine($" {city}");
   }

   WriteLine($"The first city is {cities[0]}.");
   WriteLine($"The last city is {cities[cities.Count - 1]}.");

   cities.Insert(0, "Sydney");

   WriteLine("After inserting Sydney at index 0");
   foreach (string city in cities)
   ```

```
{
  WriteLine($" {city}");
}

cities.RemoveAt(1);
cities.Remove("Milan");

WriteLine("After removing two cities");
foreach (string city in cities)
{
  WriteLine($" {city}");
}
```

4. Запустите консольное приложение и проанализируйте результат:

```
Initial list
  London
  Paris
  Milan
The first city is London.
The last city is Milan.
After inserting Sydney at index 0
  Sydney
  London
  Paris
  Milan
After removing two cities
  Sydney
  Paris
```

Работа со словарями

Работу со словарями рассмотрим на следующем примере.

1. Создайте консольное приложение `WorkingWithDictionaries`, добавьте его в рабочую область и выберите его в качестве активного проекта для OmniSharp.
2. Импортируйте пространства имен `System.Collections.Generic`.
3. В методе `Main` введите приведенный ниже код. Он демонстрирует некоторые из распространенных способов работы со словарями:

```
var keywords = new Dictionary<string, string>();
keywords.Add("int", "32-bit integer data type");
keywords.Add("long", "64-bit integer data type");
keywords.Add("float", "Single precision floating point number");

WriteLine("Keywords and their definitions");
foreach (KeyValuePair<string, string> item in keywords)
```

```
{
  WriteLine($" {item.Key}: {item.Value}");
}
WriteLine($"The definition of long is {keywords["long"]}");
```

4. Запустите приложение и проанализируйте результат:

```
Keywords and their definitions
  int: 32-bit integer data type
  long: 64-bit integer data type
  float: Single precision floating point number
The definition of long is 64-bit integer data type
```

Сортировка коллекций

Класс `List<T>` можно отсортировать, вызвав вручную его метод `Sort` (но помните, что позиции индексов всех элементов будут изменены). Ручную сортировку списка значений `string` или других встроенных типов также довольно просто выполнить, но если вы создаете коллекцию элементов собственного типа, то он должен реализовать интерфейс `IComparable`. Как это сделать, вы узнали в главе 6.

Классы `Dictionary<T>`, `Stack<T>` и `Queue<T>` не могут быть отсортированы, поскольку обычно это не требуется. Вы же никогда не будете сортировать очередь посетителей гостиницы. Но в некоторых случаях вам может понадобиться отсортировать словарь или множество.

Иногда бывает полезно иметь автоматически сортируемую коллекцию. Такая коллекция по мере добавления или удаления элементов сама будет поддерживать их упорядоченность. Существует несколько самосортирующихся коллекций. Различия между ними часто незначительны, но могут влиять на загруженность памяти и производительность вашего приложения, и потому рекомендуется выбирать коллекции, наиболее подходящие под ваши требования.

В табл. 8.11 приведены некоторые наиболее часто используемые самосортирующиеся коллекции.

Таблица 8.11

Коллекция	Описание
SortedDictionary<TKey, TValue>	Представляет собой коллекцию пар «ключ — значение», которые сортируются по ключу
SortedList<TKey, TValue>	Представляет собой коллекцию пар «ключ — значение», которые сортируются по ключу на основе связанной реализации интерфейса IComparer<T>
SortedSet<T>	Представляет собой коллекцию объектов, которые хранятся в отсортированном порядке

Использование специализированных коллекций

Существуют коллекции для особых случаев (табл. 8.12).

Таблица 8.12

Коллекция	Описание
System.Collections.BitArray	Управляет компактным массивом двоичных значений, представленных логическими значениями, где true соответствует включенному биту (1), а false — отключенному биту (0)
System.Collections.Generics.LinkedList<T>	Представляет собой двусвязный список, в котором каждый элемент имеет ссылку на свой предыдущий и следующий элементы. Они обеспечивают лучшую производительность по сравнению с List<T> для сценариев, в которых вы будете часто вставлять и удалять элементы из середины списка, поскольку в LinkedList<T> элементы не нужно переставлять в памяти

Использование неизменяемых коллекций

Иногда необходимо создать коллекцию *неизменяемой*. Это значит, что ее члены не могут изменяться, то есть вы не можете добавлять или удалять их.

Если вы импортируете пространство имен `System.Collections.Immutable`, то у любой коллекции, реализующей интерфейс `IEnumerable<T>`, появится шесть методов расширения для преобразования ее в неизменяемый список, словарь, хеш-множество и т. д.

1. В проекте `WorkingWithLists` в файле `Program.cs` импортируйте пространство имен `System.Collections.Immutable`, а затем добавьте операторы в конец метода `Main`:

```
var immutableCities = cities.ToImmutableList();

var newList = immutableCities.Add("Rio");

Write("Immutable list of cities:");
foreach (string city in immutableCities)
{
  Write($" {city}");
}
WriteLine();

Write("New list of cities:");
foreach (string city in newList)
```

```
{
  Write($" {city}");
}
WriteLine();
```

2. Запустите консольное приложение, проанализируйте результат вывода и обратите внимание, что неизменяемый список городов не изменяется при вызове метода `Add`, он возвращает новый список с вновь добавленным городом, как показано ниже:

```
Immutable cities: Sydney Paris
New cities: Sydney Paris Rio
```

Для повышения производительности многие приложения хранят копию часто используемых объектов в центральном кэше. Чтобы обезопасить работу нескольких потоков с этими объектами, необходимо сделать их неизменяемыми или использовать тип параллельной коллекции. Дополнительную информацию вы можете найти по следующей ссылке: https://docs.microsoft.com/en- нас / dotnet / api / system.collections.concurrent.

Работа с интервалами, индексами и диапазонами

Одной из целей корпорации Microsoft относительно .NET Core 2.1 было повышение производительности и улучшение использования ресурсов. Ключевой функцией .NET, обеспечивающей это, является тип `Span<T>`.

Официальную документацию, касающуюся типа Span<T> можно прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/api/system.span-1.

Управление памятью с помощью интервалов

Чтобы передавать часть коллекции, вы часто будете создавать новые коллекции из существующих объектов. Это неэффективно, поскольку в памяти создаются дубликаты объектов.

Если вам нужно работать с подмножествами коллекции, то вместо того, чтобы реплицировать подмножество в новую коллекцию, можно использовать интервал — это словно «окно» в подмножество исходной коллекции и более эффективное решение с точки зрения использования памяти и повышения производительности.

Прежде чем мы рассмотрим интервалы более подробно, нам необходимо освоить некоторые новые объекты: индексы и диапазоны.

Идентификация позиций с типом Index

В языке C# 8.0 представлены две новые функции для идентификации индекса элемента в коллекции и диапазона элементов с использованием двух индексов.

В предыдущем подразделе вы узнали, что к объектам в списке можно получить доступ, передав целое число в их индекс:

```
int index = 3;
Person p = people[index];  // четвертый человек в списке или массиве
char letter = name[index]; // четвертая буква в названии
```

Использование типа значения `Index` — более формальный способ идентификации позиции, к тому же поддерживает отсчет с конца:

```
// два способа определить один и тот же индекс, 3 в начале
var i1 = new Index(value: 3); // считает с самого начала
Index i2 = 3; // использует неявную операцию преобразования целого числа

// два способа определить один и тот же индекс, 5 в конце
var i3 = new Index(value: 5, fromEnd: true);
var i4 = ^5; // операция каретки (используется в C# 8.0)
```

Мы должны были явно определить тип `Index` во втором операторе, поскольку в противном случае компилятор будет рассматривать его как `int`. В четвертом операторе достаточно, чтобы компилятор понимал смысл символа каретки `^`.

Идентификация диапазонов с помощью типа Range

Тип значения `Range` использует значения `Index` для указания начала и конца диапазона через конструктор, синтаксис языка C# или статические методы, как показано ниже:

```
Range r1 = new Range(start: new Index(3), end: new Index(7));
Range r2 = new Range(start: 3, end: 7); // неявное преобразование целого числа
Range r3 = 3..7; // использование синтаксиса языка C# 8.0
Range r4 = Range.StartAt(3); // от индекса 3 до последнего индекса
Range r5 = 3..; // от индекса 3 до последнего индекса
Range r6 = Range.EndAt(3); // от индекса 0 до 3
Range r7 = ..3; // от индекса 0 до 3
```

Для упрощения работы с диапазонами к классу `string`, массиву `int` и структуре `Span<T>` были добавлены методы расширения. Эти методы расширения принимают

диапазон в качестве параметра и возвращают `Span<T>`, что делает их очень эффективными с точки зрения управления памятью.

Использование индексов и диапазонов

Рассмотрим применение индексов и диапазонов для возврата интервалов.

1. Создайте консольное приложение `WorkingWithRanges`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.
2. В метод `Main` добавьте операторы для сравнения использования метода `Substring` типа `string` и диапазонов для извлечения фрагментов чьего-либо имени, как показано ниже:

```
string name = "Samantha Jones";

int lengthOfFirst = name.IndexOf(, ,);
int lengthOfLast = name.Length - lengthOfFirst - 1;

string firstName = name.Substring(
  startIndex: 0,
  length: lengthOfFirst);

string lastName = name.Substring(
  startIndex: name.Length - lengthOfLast,
  length: lengthOfLast);

WriteLine($"First name: {firstName}, Last name: {lastName}");

ReadOnlySpan<char> nameAsSpan = name.AsSpan();

var firstNameSpan = nameAsSpan[0..lengthOfFirst];

var lastNameSpan = nameAsSpan[^lengthOfLast..^0];

WriteLine("First name: {0}, Last name: {1}",
  arg0: firstNameSpan.ToString(),
  arg1: lastNameSpan.ToString());
```

3. Запустите консольное приложение и проанализируйте результат:

```
First name: Samantha, Last name: Jones
First name: Samantha, Last name: Jones
```

Более подробную информацию о том, как реализованы интервалы, можно найти на сайте https://docs.microsoft.com/ru-ru/archive/msdn-magazine/2018/january/csharp-all-about-span-exploring -a-new-net-mainstay.

Работа с сетевыми ресурсами

Иногда возникает необходимость работать с сетевыми ресурсами. В табл. 8.13 описаны наиболее распространенные типы в .NET, которые предназначены для такой работы.

Таблица 8.13

Пространство имен	Тип	Описание
System.Net	Dns, Uri, Cookie, WebClient, IPAddress	Предназначены для работы с DNS-серверами, URI, IP-адресами и т. д.
System.Net	FtpStatusCode, FtpWebRequest, FtpWebResponse	Предназначены для работы с FTP-серверами
System.Net	HttpStatusCode, HttpWebRequest, HttpWebResponse	Предназначены для работы с HTTP-серверами, то есть сайтами и сервисами. Типы из System.Net.Http проще в использовании
System.Net.Http	HttpClient, HttpMethod, HttpRequestMessage, HttpResponseMessage	Предназначены для работы с HTTP-серверами, то есть сайтами и сервисами. Как их использовать, вы узнаете в главе 18
System.Net.Mail	Attachment, MailAddress, MailMessage, SmtpClient	Предназначены для работы с SMTP-серверами (отправка сообщений электронной почты)
System.Net.NetworkInformation	IPStatus, NetworkChange, Ping, TcpStatistics	Предназначены для работы с низкоуровневыми сетевыми протоколами

Работа с URI, DNS и IP-адресами

Рассмотрим некоторые распространенные типы для работы с сетевыми ресурсами.

1. Создайте консольное приложение `WorkingWithNetworkResources`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.
2. В начале файла импортируйте следующее пространство имен:

```
using System.Net;
```

3. Добавьте в метод `Main` операторы, чтобы предложить пользователю ввести адрес сайта, а затем с помощью типа `Uri` разделите его на части, включая схему (HTTP, FTP и т. д.), номер порта и имя хоста:

```
Write("Enter a valid web address: ");
string url = ReadLine();
```

```
if (string.IsNullOrWhiteSpace(url))
{
  url = "https://world.episerver.com/cms/?q=pagetype";
}

var uri = new Uri(url);

WriteLine($"URL: {url}");
WriteLine($"Scheme: {uri.Scheme}");
WriteLine($"Port: {uri.Port}");
WriteLine($"Host: {uri.Host}");
WriteLine($"Path: {uri.AbsolutePath}");
WriteLine($"Query: {uri.Query}");
```

Для удобства я также разрешил пользователю просто нажать клавишу Enter, чтобы использовать URL, выбранный мной в качестве примера.

4. Запустите консольное приложение, введите правильный адрес сайта или нажмите клавишу Enter и проанализируйте результат:

```
Enter a valid web address:
URL: https://world.episerver.com/cms/?q=pagetype
Scheme: https
Port: 443
Host: world.episerver.com
Path: /cms/
Query: ?q=pagetype
```

5. Чтобы получить IP-адрес для введенного сайта, добавьте операторы в метод `Main`:

```
IPHostEntry entry = Dns.GetHostEntry(uri.Host);
WriteLine($"{entry.HostName} has the following IP addresses:");
foreach (IPAddress address in entry.AddressList)
{
  WriteLine($" {address}");
}
```

6. Запустите консольное приложение, введите правильный адрес сайта или нажмите клавишу Enter и проанализируйте результат:

```
world.episerver.com.cdn.cloudflare.net has the following IP addresses:
  104.18.23.198
  104.18.22.198
```

Проверка соединения с сервером

Теперь необходимо добавить программный код для проверки соединения с веб-сервером.

1. В файл `Program.cs` добавьте следующий оператор для импорта пространства имен `System.Net.NetworkInformation`:

```
using System.Net.NetworkInformation;
```

2. В метод `Main` добавьте следующие операторы, проверяющие соединение с сервером:

```
try
{
  var ping = new Ping();
  WriteLine("Pinging server. Please wait...");
  PingReply reply = ping.Send(uri.Host);

  WriteLine($"{uri.Host} was pinged and replied: {reply.Status}.");
  if (reply.Status == IPStatus.Success)
  {
    WriteLine("Reply from {0} took {1:N0}ms",
      reply.Address, reply.RoundtripTime);
  }
}
catch (Exception ex)
{
  WriteLine($"{ex.GetType().ToString()} says {ex.Message}");
}
```

3. Запустите консольное приложение, нажмите клавишу Enter, проанализируйте результат вывода, как показано в следующих выходных данных для macOS:

```
Pinging server. Please wait...
world.episerver.com was pinged and replied: Success.
Reply from 104.18.23.198 took 4ms
```

4. Снова запустите консольное приложение и введите адрес http://google.com, как показано ниже:

```
Enter a valid web address: http://google.com
URL: http://google.com
Scheme: http
Port: 80
Host: google.com
Path: /
Query:
google.com has the following IP addresses:
172.217.18.78
google.com was pinged and replied: Success.
Reply from 172.217.18.78 took 19ms
```

Работа с типами и атрибутами

Отражение (или *рефлексия*) означает процесс, в ходе которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения. Сборка состоит из таких четырех частей, как:

- *метаданные и манифест сборки* — имя, сборка и версия файла, ссылки на сборки и т. д.;
- *метаданные типов* — информация о типах, их членах и т. д.;
- *IL-код* — реализация методов, свойств, конструкторов и т. д.;
- *встроенные ресурсы (опционально)* — изображения, строки, JavaScript и т. д.

Метаданные содержат информацию о вашем коде. Метаданные автоматически генерируются из вашего кода (например, информация о типах и членах) или применяются к вашему коду с помощью атрибутов.

Атрибуты могут применяться на нескольких уровнях: к сборкам, типам и их элементам:

```
// атрибут уровня сборки
[assembly: AssemblyTitle("Working with Reflection")]

// атрибут уровня типа
[Serializable]
public class Person
{
  // атрибут уровня члена
  [Obsolete("Deprecated: use Run instead.")]
  public void Walk()
  {
     ...
```

Версии сборок

Номера версий в .NET представляют собой комбинацию из трех чисел с двумя необязательными дополнениями.

Следуйте принципам семантического версионирования:

- *старшая версия* — изменяется при добавлении новой функциональности, которая может не быть обратно совместимой;
- *младшая версия* — содержит обратно совместимые изменения, включая новые функции и исправления ошибок;
- *патч-версия* — содержит обратно совместимые исправления ошибок.

При обновлении пакета NuGet вы должны указать дополнительный флаг, чтобы обновляться только до самой высокой младшей версии с целью избежать несовместимых изменений или до самой высокой патч-версии, если вы очень осторожны и хотите получать только исправления ошибок, как показано ниже:

```
Update-Package Newtonsoft.Json -ToHighestMinor or Update-Package
Newtonsoft.Json -ToHighestPatch
```

При необходимости версия может включать в себя:

- *предварительный выпуск* — неподдерживаемые предварительные версии;
- *номер сборки* — ночные сборки.

Следуйте правилам семантического версионирования, описанным на сайте http://semver.org.

Чтение метаданных сборки

Рассмотрим работу с атрибутами на следующем примере.

1. Создайте консольное приложение `WorkingWithReflection`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.
2. В начале файла импортируйте следующее пространство имен:

```
using System.Reflection;
```

3. В методе `Main` введите операторы, чтобы получить сведения о сборке консольного приложения, его имени и местоположении, а также получить все атрибуты уровня сборки и вывести их типы:

```
WriteLine("Assembly metadata:");
Assembly assembly = Assembly.GetEntryAssembly();

WriteLine($" Full name: {assembly.FullName}");
WriteLine($" Location: {assembly.Location}");

var attributes = assembly.GetCustomAttributes();

WriteLine($" Attributes:");
foreach (Attribute a in attributes)
{
  WriteLine($" {a.GetType()}");
}
```

4. Запустите консольное приложение и проанализируйте результат:

```
Assembly metadata:
  Full name: WorkingWithReflection, Version=1.0.0.0, Culture=neutral,
  PublicKeyToken=null
  Location: /Users/markjprice/Code/Chapter08/WorkingWithReflection/bin/
  Debug/net5.0/WorkingWithReflection.dll
Attributes:
  System.Runtime.CompilerServices.CompilationRelaxationsAttribute
  System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
  System.Diagnostics.DebuggableAttribute
  System.Runtime.Versioning.TargetFrameworkAttribute
  System.Reflection.AssemblyCompanyAttribute
  System.Reflection.AssemblyConfigurationAttribute
  System.Reflection.AssemblyFileVersionAttribute
  System.Reflection.AssemblyInformationalVersionAttribute
  System.Reflection.AssemblyProductAttribute
  System.Reflection.AssemblyTitleAttribute
```

Теперь, зная некоторые атрибуты, определенные для сборки, мы можем специально их запросить.

5. Чтобы получить классы `AssemblyInformationalVersionAttribute` и `AssemblyCompanyAttribute`, добавьте операторы в конец метода `Main`, как показано ниже:

```
var version = assembly.GetCustomAttribute
  <AssemblyInformationalVersionAttribute>();

WriteLine($" Version: {version.InformationalVersion}");

var company = assembly.GetCustomAttribute
  <AssemblyCompanyAttribute>();

WriteLine($" Company: {company.Company}");
```

6. Запустите консольное приложение и проанализируйте результат:

```
  Version: 1.0.0
  Company: WorkingWithReflection
```

Давайте явно установим данную информацию. Прежний способ установки этих значений в .NET Framework заключался в добавлении атрибутов в файл исходного кода C#:

```
[assembly: AssemblyCompany("Packt Publishing")]
[assembly: AssemblyInformationalVersion("1.3.0")]
```

Компилятор Roslyn, используемый в .NET Core, устанавливает эти атрибуты автоматически, ввиду чего мы не можем применить прежний способ. Вместо этого они могут быть установлены в файле проекта.

7. Отредактируйте код файла `WorkingWithReflection.csproj`:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>

    <Version>1.3.0</Version>
    <Company>Packt Publishing</Company>
  </PropertyGroup>
</Project>
```

8. Запустите консольное приложение и проанализируйте результат:

```
  Version: 1.3.0
  Company: Packt Publishing
```

Создание пользовательских атрибутов

Вы можете определить собственные атрибуты, наследуя от класса `Attribute`.

1. Добавьте файл класса в проект `CoderAttribute.cs`.
2. Определите класс атрибута, который может использоваться либо с классами, либо с методами. Добавьте в этот класс два свойства для хранения имени кодировщика и даты последнего изменения:

```
using System;

namespace Packt.Shared
{
  [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
    AllowMultiple = true)]
  public class CoderAttribute : Attribute
  {
    public string Coder { get; set; }
    public DateTime LastModified { get; set; }

    public CoderAttribute(string coder, string lastModified)
    {
      Coder = coder;
      LastModified = DateTime.Parse(lastModified);
    }
  }
}
```

3. В файле `Program.cs` импортируйте метод `System.Linq`:

```
using System.Linq; // использование OrderByDescending
using System.Runtime.CompilerServices; // использование
CompilerGeneratedAttribute
using Packt.Shared; // CoderAttribute
```

Более подробную информацию о LINQ вы получите в главе 12. Здесь мы его импортируем, чтобы использовать метод `OrderByDescending`.

4. В классе `Program` добавьте метод `DoStuff` и дополните его атрибутом `Coder` с данными о двух кодировщиках:

```
[Coder("Mark Price", "22 August 2019")]
[Coder("Johnni Rasmussen", "13 September 2019")]
public static void DoStuff()
{
}
```

5. В метод `Main` добавьте код для получения типов, перечислите их элементы, прочитайте все атрибуты `Coder` этих элементов и запишите информацию в консоль, как показано ниже:

```
WriteLine();
WriteLine($"* Types:");
Type[] types = assembly.GetTypes();

foreach (Type type in types)
{
  WriteLine();
  WriteLine($"Type: {type.FullName}");
  MemberInfo[] members = type.GetMembers();

  foreach (MemberInfo member in members)
  {
    WriteLine("{0}: {1} ({2})",
      arg0: member.MemberType,
      arg1: member.Name,
      arg2: member.DeclaringType.Name);

    var coders = member.GetCustomAttributes<CoderAttribute>()
      .OrderByDescending(c => c.LastModified);

    foreach (CoderAttribute coder in coders)
    {
      WriteLine("-> Modified by {0} on {1}",
        coder.Coder, coder.LastModified.ToShortDateString());
    }
  }
}
```

6. Запустите консольное приложение и проанализируйте результат:

```
* Types:

Type: CoderAttribute
Method: get_Coder (CoderAttribute)
Method: set_Coder (CoderAttribute)
Method: get_LastModified (CoderAttribute)
```

```
Method: set_LastModified (CoderAttribute) Method: Equals (Attribute)
Method: GetHashCode (Attribute) Method: get_TypeId (Attribute)
Method: Match (Attribute)
Method: IsDefaultAttribute (Attribute)
Method: ToString (Object)
Method: GetType (Object)
Constructor: .ctor (CoderAttribute)
Property: Coder (CoderAttribute)
Property: LastModified (CoderAttribute)
Property: TypeId (Attribute)

Type: WorkingWithReflection.Program
Method: DoStuff (Program)
-> Modified by Johnni Rasmussen on 13/09/2019
-> Modified by Mark Price on 22/08/2019
Method: ToString (Object)
Method: Equals (Object)
Method: GetHashCode (Object)
Method: GetType (Object)
Constructor: .ctor (Program)

Type: WorkingWithReflection.Program+<>c
Method: ToString (Object)
Method: Equals (Object)
Method: GetHashCode (Object)
Method: GetType (Object)
Constructor: .ctor (<>c)
Field: <>9 (<>c)
Field: <>9__0_0 (<>c)
```

Что означает тип WorkingWithReflection.Program+<>c? Это сгенерированный компилятором класс отображения. Оператор <> указывает на то, что он сгенерирован компилятором, а с — на то, что это класс отображения. Более подробную информацию можно найти на сайте http://stackoverflow.com/a/2509524/55847.

В качестве дополнительной задачи добавьте операторы в консольное приложение для фильтрации сгенерированных компилятором типов, пропустив типы, дополненные атрибутом `CompilerGeneratedAttribute`.

Еще немного об отражении

Рассмотрим, что может быть достигнуто с помощью отражения. Мы использовали отражение только для чтения метаданных из нашего кода. Кроме этого, оно может динамически:

- загружать сборки, на которые в данный момент нет ссылок: https://docs.microsoft.com/ru-ru/dotnet/standard/assembly/unloadability-howto;

- выполнять код: https://docs.microsoft.com/ru-ru/dotnet/api/system.reflection.methodbase.invoke;
- генерировать новый код и сборки: https://docs.microsoft.com/ru-ru/dotnet/api/system.reflection.emit.assemblybuilder.

Работа с изображениями

ImageSharp — это сторонняя кросс-платформенная библиотека 2D-графики. Когда версия .NET Core 1.0 находилась в разработке, были отрицательные отзывы об отсутствии пространства имен `System.Drawing` для работы с 2D-изображениями. Проект `ImageSharp` был начат, чтобы восполнить этот пробел для современных приложений .NET.

В официальной документации по `System.Drawing` компания Microsoft сообщает: «Пространство имен `System.Drawing` не рекомендуется для новой разработки, так как оно не поддерживается в службах Windows или ASP.NET и не является кросс-платформенным. `ImageSharp` и `SkiaSharp` рекомендуются в качестве альтернативы».

Рассмотрим пример, чего можно достичь, используя ImageSharp.

1. Создайте новый проект консольного приложения с именем `WorkingWithImages`, добавьте его в рабочую область и выберите его в качестве активного проекта для OmniSharp.
2. Создайте папку изображений и загрузите девять изображений по следующей ссылке: https://github.com/markjprice/cs9dotnet5/tree/master/Assets/Categories.
3. Откройте файл `WorkingWithImages.csproj` и добавьте ссылку на пакет для `SixLabors.ImageSharp`:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="SixLabors.ImageSharp" Version="1.0.0" />
  </ItemGroup>
</Project>
```

4. В начале файла `Program.cs` импортируйте следующие пространства имен:

```
using System.Collections.Generic;
using System.IO;
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Processing;
```

5. В методе `Main` введите операторы для преобразования всех файлов в папке изображений в миниатюры, представленных в оттенках серого с размером одна десятая:

```
string imagesFolder = Path.Combine(
  Environment.CurrentDirectory, "images");

IEnumerable<string> images =
  Directory.EnumerateFiles(imagesFolder);

foreach (string imagePath in images)
{
  string thumbnailPath = Path.Combine(
    Environment.CurrentDirectory, "images",
    Path.GetFileNameWithoutExtension(imagePath)
    + "-thumbnail" + Path.GetExtension(imagePath)
  );

  using (Image image = Image.Load(imagePath))
  {
    image.Mutate(x => x.Resize(image.Width / 10, image.Height / 10));
    image.Mutate(x => x.Grayscale());
    image.Save(thumbnailPath);
  }
}
```

6. Запустите консольное приложение.
7. В файловой системе откройте папку изображений и обратите внимание на гораздо меньшие по размеру в байтах миниатюры, представленные в оттенках серого (рис. 8.2).

Рис. 8.2. Изображения после обработки

Дополнительную информацию об ImageSharp вы можете получить на сайте https://github.com/SixLabors/ImageSharp.

Интернационализация кода

Интернационализация — это процесс, позволяющий вашему приложению выполняться правильно во всем мире. Он состоит из двух частей: *глобализации* и *локализации*.

Глобализация заключается в написании кода для поддержки разных языковых и региональных параметров. При разработке приложений важно учитывать язык и регион, поскольку форматы даты и валюты отличаются, к примеру, в Квебеке и Париже, несмотря на то что в обоих городах говорят по-французски.

Существуют коды *Международной организации по стандартизации* (International Standards Organization, ISO) для всех языковых и региональных параметров. Например, в коде `da-DK` символы `da` указывают на датский язык, а символы `DK` определяют страну — Данию; в коде `fr-CA` символы `fr` указывают на французский язык, а `CA` определяют страну — Канаду.

ISO не аббревиатура. Это отсылка к греческому слову *isos*, которое в переводе означает «равный».

Локализация — это настройка пользовательского интерфейса для реализации поддержки языка, например изменение названия кнопки Закрыть на Close (`en`) или Fermer (`fr`). Поскольку локализация больше связана с языком, ей не всегда нужно учитывать регион.

Ирония, однако, в том, что даже слово «стандартизация» в `en-US` (standardization) и `en-GB` (standardisation) намекает, что знать о регионе все же полезно.

Обнаружение и изменение региональных настроек

Интернационализация — огромная тема, по которой написаны целые книги. В этом разделе вы изучите самые основы, используя тип `CultureInfo` в пространстве имен `System.Globalization`.

1. Создайте проект консольного приложения `Internationalization`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.

2. В начале файла импортируйте следующее пространство имен:

```
using System.Globalization;
```

3. В методе `Main` введите операторы для получения текущих языковых и региональных настроек и запишите часть информации о них в консоль, а затем предложите пользователю ввести новые настройки, чтобы показать, насколько это влияет на форматирование значений, таких как дата и валюта:

```
CultureInfo globalization = CultureInfo.CurrentCulture;
CultureInfo localization = CultureInfo.CurrentUICulture;

WriteLine("The current globalization culture is {0}: {1}",
  globalization.Name, globalization.DisplayName);
WriteLine("The current localization culture is {0}: {1}",
  localization.Name, localization.DisplayName);
WriteLine();

WriteLine("ru-ru: English (United States)");
WriteLine("da-DK: Danish (Denmark)");
WriteLine("fr-CA: French (Canada)");
Write("Enter an ISO culture code: ");
string newCulture = ReadLine();

if (!string.IsNullOrEmpty(newCulture))
{
  var ci = new CultureInfo(newCulture);
  CultureInfo.CurrentCulture = ci;
  CultureInfo.CurrentUICulture = ci;
}
WriteLine();

Write("Enter your name: ");
string name = ReadLine();

Write("Enter your date of birth: ");
string dob = ReadLine();

Write("Enter your salary: ");
string salary = ReadLine();

DateTime date = DateTime.Parse(dob);
int minutes = (int)DateTime.Today.Subtract(date).TotalMinutes;
decimal earns = decimal.Parse(salary);

WriteLine(
  "{0} was born on a {1:dddd}, is {2:N0} minutes old, and earns {3:C}",
  name, date, minutes, earns);
```

При запуске приложения поток автоматически устанавливается на использование языковых и региональных настроек в соответствии с операционной системой. Я запускаю свой код в Лондоне, поэтому поток уже установлен на английский язык (Великобритания).

Код уведомит пользователя, что нужно указать другой ISO-код. Благодаря этому ваши приложения смогут менять языковые и региональные настройки по умолчанию во время выполнения.

Затем приложение использует стандартные коды формата для вывода дня недели (`dddd`), количества минут с разделителями тысяч (`N0`) и денежных единиц с символом валюты (`C`). Настройка происходит автоматически на основе языковых и региональных настроек потока.

4. Запустите консольное приложение, введите значение `en-GB` в качестве ISO-кода, а затем какие-нибудь данные для примера, включая дату в формате, допустимом для британского варианта английского языка:

```
Enter an ISO culture code: en-GB
Enter your name: Alice
Enter your date of birth: 30/3/1967
Enter your salary: 23500
Alice was born on a Thursday, is 25,469,280 minutes old, and earns
£23,500.00
```

5. Перезапустите приложение и используйте другие языковые и региональные настройки, например датский язык в Дании (`da-DK`):

```
Enter an ISO culture code: da-DK
Enter your name: Mikkel
Enter your date of birth: 12/3/1980
Enter your salary: 340000
Mikkel was born on a onsdag, is 18.656.640 minutes old, and earns
340.000,00 kr.
```

Подумайте, нужна ли интернационализация вашему приложению, и если да, то запланируйте ее прежде, чем начинать кодирование! Запишите весь применяемый в пользовательском интерфейсе текст, который необходимо будет локализовать. Подумайте обо всех данных, которые подлежат глобализации (форматы даты и чисел, сортировка текста).

Обработка часовых поясов

Одна из самых сложных областей интернационализации — это работа с часовыми поясами. Это слишком сложная тема, чтобы описать ее в этой книге.

Дополнительную информацию о часовых поясах вы можете найти на сайте https://devblogs.microsoft.com/dotnet/cross-platform-time-zones-with-net-core/.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 8.1. Проверочные вопросы

Пользуясь ресурсами в Интернете, ответьте на следующие вопросы.

1. Какое максимальное количество символов может быть сохранено в переменной `string`?
2. В каких случаях и почему нужно использовать тип `SecureString`?
3. В каких ситуациях целесообразно применить тип `StringBuilder`?
4. В каких случаях следует задействовать `LinkedList<T>`?
5. Когда класс `SortedDictionary<T>` нужно использовать вместо класса `SortedList<T>`?
6. Каков ISO-код языковых и региональных параметров ISO для валлийского языка?
7. В чем разница между локализацией, глобализацией и интернационализацией?
8. Что означает символ `$` в регулярных выражениях?
9. Как в регулярных выражениях представить цифры?
10. Почему *нельзя* применять официальный стандарт для адресов электронной почты при создании регулярного выражения, призванного проверять адрес электронной почты пользователя?

Упражнение 8.2. Регулярные выражения

Создайте проект консольного приложения `Exercise02`, которое предлагает пользователю ввести сначала регулярное выражение, а затем еще некоторые данные и выполняет их сравнение на соответствие выражению. Процесс повторяется, пока пользователь не нажмет клавишу Esc.

```
The default regular expression checks for at least one digit.
Enter a regular expression (or press ENTER to use the default): ^[a-z]+$
```

```
Enter some input: apples
apples matches ^[a-z]+$? True
Press ESC to end or any key to try again.
Enter a regular expression (or press ENTER to use the default): ^[a-z]+$
Enter some input: abc123xyz
abc123xyz matches ^[a-z]+$? False
Press ESC to end or any key to try again.
```

Упражнение 8.3. Методы расширения

Создайте библиотеку классов `Exercise03`, которая определяет методы, расширяющие числовые типы, такие как `BigInteger` и `int`, с помощью метода `ToWords`, который возвращает строку, описывающую число. Например, 18 000 000 — восемнадцать миллионов, а 18 456 002 032 011 000 007 — восемнадцать квинтиллионов четыреста пятьдесят шесть квадриллионов два триллиона тридцать два миллиарда одиннадцать миллионов семь.

Более подробно об именах для больших чисел можно прочитать на сайте https://en.wikipedia.org/wiki/Names_of_large_numbers.

Упражнение 8.4. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- справочник .NET API: https://docs.microsoft.com/ru-ru/dotnet/api/;
- класс `String`: https://docs.microsoft.com/ru-ru/dotnet/api/system.string;
- класс `Regex`: //docs.microsoft.com/ru-ru/dotnet/api/system.text.regularexpressions.regex;
- регулярные выражения в .NET: https://docs.microsoft.com/ru-ru/dotnet/articles/standard/base-types/regular-expressions;
- язык регулярных выражений — быстрое руководство: https://docs.microsoft.com/ru-ru/dotnet/standard/base-types/regular-expression-language-quick-reference;
- коллекции (C# и Visual Basic): https://docs.microsoft.com/ru-ru/dotnet/api/system.collections;
- расширение метаданных с помощью атрибутов: https://docs.microsoft.com/ru-ru/dotnet/standard/attributes/;
- глобализация и локализация приложений .NET: https://docs.microsoft.com/ru-ru/dotnet/standard/globalization-localization/.

Резюме

Вы узнали об эффективных способах использования типов для хранения чисел и текста и управления ими, включая регулярные выражения; о том, какие коллекции задействовать для хранения групп элементов; изучили индексы, диапазоны и интервалы, научились использовать некоторые сетевые ресурсы, применять рефлексию для кода и атрибутов, а также интернационализировать код.

Далее вы узнаете, как управлять файлами и потоками, кодировать и декодировать текст и выполнять сериализацию.

9 Работа с файлами, потоками и сериализация

Данная глава посвящена чтению и записи в файлы и потоки, кодированию текста и сериализации.

В этой главе:

- управление файловой системой;
- чтение и запись с помощью потоков;
- кодирование и декодирование текста;
- сериализация графов объектов.

Управление файловой системой

В приложениях часто возникает необходимость выполнять операции ввода и вывода с файлами и каталогами. Пространства имен `System` и `System.IO` содержат классы для выполнения этих задач.

Работа с различными платформами и файловыми системами

Рассмотрим, как работать с различными платформами, учитывая различия между Windows, macOS или Linux.

1. Создайте в папке Chapter09 проект консольного приложения `WorkingWithFileSystems`.
2. Сохраните рабочую область как `Chapter09` и добавьте в нее приложение `WorkingWithFileSystems`.
3. Импортируйте пространство имен `System.IO` и статически импортируйте типы `System.Console`, `System.IO.Directory`, `System.Environment` и `System.IO.Path`, как показано ниже:

```
using System.IO; // типы для управления файловой системой
using static System.Console;
using static System.IO.Directory;
using static System.IO.Path;
using static System.Environment;
```

Начнем с изучения того, как платформа .NET работает с различными операционными системами, такими как Windows, macOS и Linux.

4. Создайте статический метод `OutputFileSystemInfo` и добавьте операторы для выполнения следующих действий:
   - вывод символов-разделителей для пути и каталогов;
   - вывод пути к текущему каталогу;
   - вывод нескольких специальных путей для системных и временных файлов и документов.

```
static void OutputFileSystemInfo()
{
  WriteLine("{0,-33} {1}", "Path.PathSeparator", PathSeparator);
  WriteLine("{0,-33} {1}", "Path.DirectorySeparatorChar",
    DirectorySeparatorChar);
  WriteLine("{0,-33} {1}", "Directory.GetCurrentDirectory()",
    GetCurrentDirectory());
  WriteLine("{0,-33} {1}", "Environment.CurrentDirectory",
    CurrentDirectory);
  WriteLine("{0,-33} {1}", "Environment.SystemDirectory",
    SystemDirectory);
  WriteLine("{0,-33} {1}", "Path.GetTempPath()", GetTempPath());

  WriteLine("GetFolderPath(SpecialFolder");
  WriteLine("{0,-33} {1}", " .System)",
    GetFolderPath(SpecialFolder.System));
  WriteLine("{0,-33} {1}", " .ApplicationData)",
    GetFolderPath(SpecialFolder.ApplicationData));
  WriteLine("{0,-33} {1}", " .MyDocuments)",
    GetFolderPath(SpecialFolder.MyDocuments));
  WriteLine("{0,-33} {1}", " .Personal)",
    GetFolderPath(SpecialFolder.Personal));
}
```

Тип `Environment` содержит множество других полезных членов, включая метод `GetEnvironmentVariables` и свойства `OSVersion` и `ProcessorCount`.

5. В методе `Main` вызовите метод `OutputFileSystemInfo`:

```
static void Main(string[] args)
{
  OutputFileSystemIn fo();
}
```

6. Запустите консольное приложение и проанализируйте результат, возникающий при запуске в операционной системе Windows (рис. 9.1).

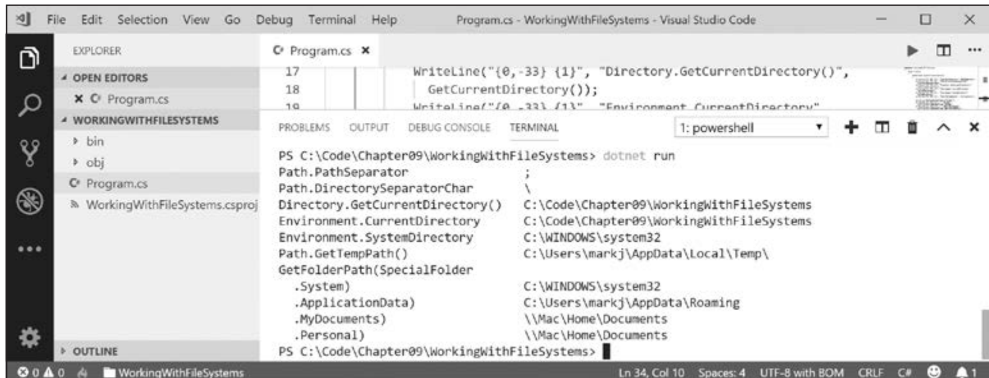


Рис. 9.1. Запуск вашего приложения для отображения информации о файловой системе

В операционной системе Windows в качестве разделителя каталогов используется символ обратной косой черты, или обратный слеш. В операционных же системах macOS и Linux — символ косой черты (слеш).

Управление дисками

Для управления дисками используйте класс `DriveInfo`, который содержит статический метод, возвращающий в результате сведения обо всех дисках, подключенных к вашему компьютеру. Каждый диск имеет тип.

1. Создайте метод `WorkWithDrives` и добавьте следующие операторы, чтобы получить все диски и вывести их имя, тип, размер, доступное свободное пространство и формат, но только если диск находится в состоянии готовности:

```
static void WorkWithDrives()
{
  WriteLine("{0,-30} | {1,-10} | {2,-7} | {3,18} | {4,18}",
    "NAME", "TYPE", "FORMAT", "SIZE (BYTES)", "FREE SPACE");

  foreach (DriveInfo drive in DriveInfo.GetDrives())
  {
    if (drive.IsReady)
    {
      WriteLine(
        "{0,-30} | {1,-10} | {2,-7} | {3,18:N0} | {4,18:N0}",
        drive.Name, drive.DriveType, drive.DriveFormat,
```

```
        drive.TotalSize, drive.AvailableFreeSpace);
    }
    else
    {
      WriteLine("{0,-30} | {1,-10}", drive.Name, drive.DriveType);
    }
  }
}
```

Убедитесь, что диск находится в состоянии готовности, прежде чем использовать свойства, такие как TotalSize. В противном случае возможна ошибка, обычно возникающая при работе со съемными носителями.

2. В методе `Main` закомментируйте предыдущий вызов метода и добавьте вызов метода `WorkWithDrives`:

```
static void Main(string[] args)
{
  // OutputFileSystemInfo();
  WorkWithDrives();
}
```

3. Запустите консольное приложение и проанализируйте результат (рис. 9.2).

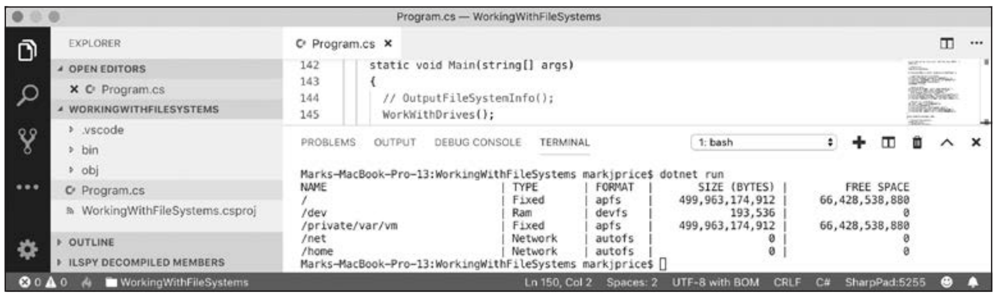


Рис. 9.2. Отображение информации о диске

Управление каталогами

Для управления каталогами используйте статические классы `Directory`, `Path` и `Environment`.

Эти типы содержат множество свойств и методов для работы с файловой системой, как показано на рис. 9.3.

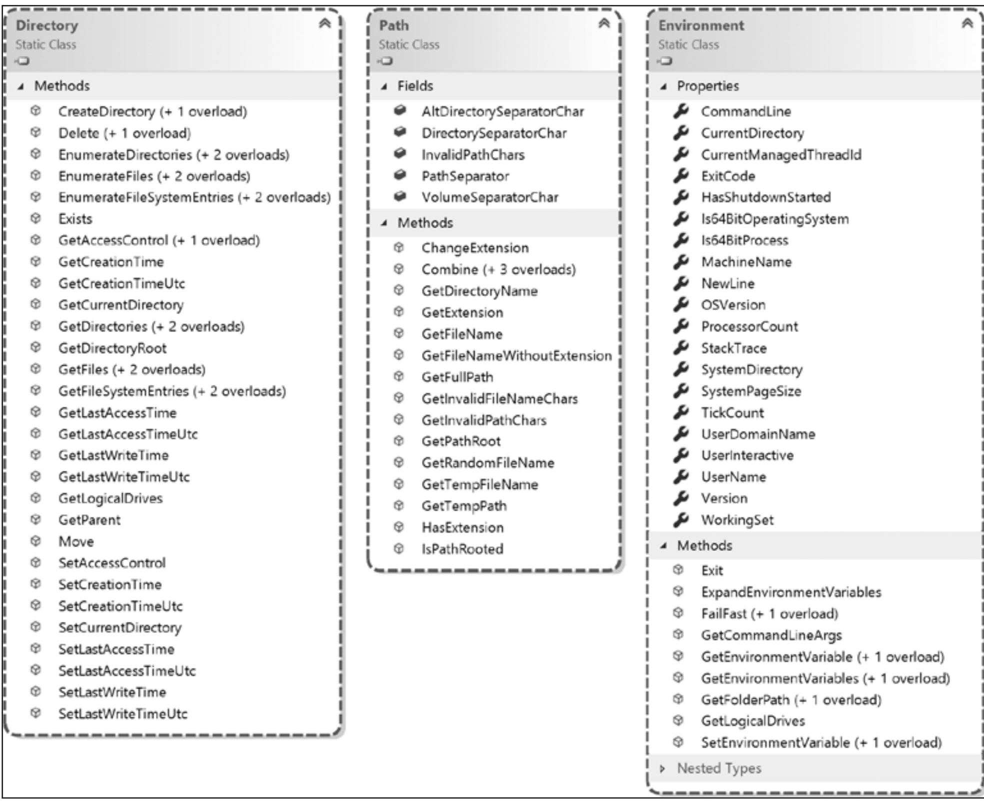


Рис. 9.3. Статические типы и их члены для работы с файловыми системами

При создании собственных путей в коде вы должны быть очень внимательны и не допускать необоснованных предположений о том, например, какой символ применить в качестве разделителя каталогов.

1. Создайте метод `WorkWithDirectories` и добавьте операторы для выполнения следующих действий:

   - определите собственный путь в корневом каталоге пользователя, создав массив строк для имен каталогов, а затем корректно скомбинировав их с помощью статического метода `Combine` типа `Path`;
   - проверьте наличие пользовательского пути к каталогу, применив статический метод `Exists` класса `Directory`;
   - с помощью статических методов `CreateDirectory` и `Delete` класса `Directory` создайте каталог, а затем удалите его, включая файлы и подкаталоги в нем.

```
static void WorkWithDirectories()
{
```

```
    // определение пути к каталогу для новой папки,
    // начиная с папки пользователя
    var newFolder = Combine(
      GetFolderPath(SpecialFolder.Personal),
      "Code", "Chapter09", "NewFolder");

    WriteLine($"Working with: {newFolder}");

    // проверка существования каталога
    WriteLine($"Does it exist? {Exists(newFolder)}");

    // создание каталога
    WriteLine("Creating it...");
    CreateDirectory(newFolder);
    WriteLine($"Does it exist? {Exists(newFolder)}");
    Write("Confirm the directory exists, and then press ENTER: ");
    ReadLine();

    // удаление каталога
    WriteLine("Deleting it...");
    Delete(newFolder, recursive: true);
    WriteLine($"Does it exist? {Exists(newFolder)}");
  }
```

2. В методе `Main` закомментируйте предыдущий вызов метода и добавьте вызов метода `WorkWithDirectories`:

```
static void Main(string[] args)
{
  // OutputFileSystemInfo();
  // WorkWithDrives();
  WorkWithDirectories();
}
```

3. Запустите консольное приложение, проанализируйте результат и используйте ваш любимый инструмент управления файлами, чтобы подтвердить создание каталога, прежде чем нажать клавишу Enter для его удаления:

```
Working with: /Users/markjprice/Code/Chapter09/NewFolder
Does it exist? False
Creating it...
Does it exist? True
Confirm the directory exists, and then press ENTER:
Deleting it...
Does it exist? False
```

Управление файлами

Обратите внимание: на сей раз мы не выполняем статический импорт типа `File`, поскольку он содержит некоторые из тех же методов, что и тип `Directory`, а это может привести к конфликту. В нашем случае тип `File` имеет достаточно короткое имя.

1. Создайте метод `WorkWithFiles` и добавьте операторы для выполнения таких действий, как:
   - проверка существования файла;
   - создание текстового файла;
   - запись текстовой строки в файл;
   - закрытие файла для освобождения системных ресурсов и блокировок файла (обычно это делается внутри блока операторов `try-finally`, чтобы обеспечить закрытие файла, даже если при записи в него возникает исключение);
   - резервное копирование файла;
   - удаление оригинального файла;
   - чтение содержимого файла из резервной копии с последующим закрытием резервного файла.

```
static void WorkWithFiles()
{
  // определение пути к каталогу для выходных файлов,
  // начиная с папки пользователя
  var dir = Combine(
    GetFolderPath(SpecialFolder.Personal),
    "Code", "Chapter09", "OutputFiles");

  CreateDirectory(dir);

  // определение путей к файлам
  string textFile = Combine(dir, "Dummy.txt");
  string backupFile = Combine(dir, "Dummy.bak");
  WriteLine($"Working with: {textFile}");

  // проверка существования файла
  WriteLine($"Does it exist? {File.Exists(textFile)}");

  // создание нового текстового файла и запись текстовой строки
  StreamWriter textWriter = File.CreateText(textFile);
  textWriter.WriteLine("Hello, C#!");
  textWriter.Close(); // закрытие файла и освобождение ресурсов
  WriteLine($"Does it exist? {File.Exists(textFile)}");

  // копирование файла с перезаписью (если существует)
  File.Copy(sourceFileName: textFile,
    destFileName: backupFile, overwrite: true);
  WriteLine(
    $"Does {backupFile} exist? {File.Exists(backupFile)}");
  Write("Confirm the files exist, and then press ENTER: ");
  ReadLine();

  // удаление файла
  File.Delete(textFile);
  WriteLine($"Does it exist? {File.Exists(textFile)}");
```

```
    // чтение содержимого текстового файла
    WriteLine($"Reading contents of {backupFile}:");
    StreamReader textReader = File.OpenText(backupFile);
    WriteLine(textReader.ReadToEnd());
    textReader.Close();
  }
```

2. В методе `Main` закомментируйте предыдущий вызов и добавьте вызов метода `WorkWithFiles`.
3. Запустите консольное приложение и проанализируйте результат:

```
Working with: /Users/markjprice/Code/Chapter09/OutputFiles/Dummy.txt
Does it exist? False
Does it exist? True
Does /Users/markjprice/Code/Chapter09/OutputFiles/Dummy.bak exist?
True
Confirm the files exist, and then press ENTER:
Does it exist? False
Reading contents of /Users/markjprice/Code/Chapter09/OutputFiles/
Dummy.bak:
Hello, C#!
```

Управление путями

В одних случаях вам может потребоваться работа с путями, например, если нужно извлечь только имя папки, файла или расширение. В других понадобится создавать временные папки и имена файлов. Все это выполняется с помощью класса `Path`.

1. Добавьте следующие операторы в конец метода `WorkWithFiles`:

```
// Управление путями
WriteLine($"Folder Name: {GetDirectoryName(textFile)}");
WriteLine($"File Name: {GetFileName(textFile)}");
WriteLine("File Name without Extension: {0}",
  GetFileNameWithoutExtension(textFile));
WriteLine($"File Extension: {GetExtension(textFile)}");
WriteLine($"Random File Name: {GetRandomFileName()}");
WriteLine($"Temporary File Name: {GetTempFileName()}");
```

2. Запустите консольное приложение и проанализируйте результат:

```
Folder Name: /Users/markjprice/Code/Chapter09/OutputFiles
File Name: Dummy.txt
File Name without Extension: Dummy
File Extension: .txt
Random File Name: u45w1zki.co3
Temporary File Name:
/var/folders/tz/xx0y_wld5sx0nv0fjtq4tnpc0000gn/T/tmpyqrepP.tmp
```

Метод `GetTempFileName` создает файл нулевого размера и возвращает его имя, готовое к использованию. А метод `GetRandomFileName` просто возвращает имя файла, не создавая сам файл.

Извлечение информации о файле

Для получения дополнительной информации о файле или каталоге вы можете создать экземпляр класса `FileInfo` или `DirectoryInfo` соответственно.

Классы `FileInfo` и `DirectoryInfo` наследуются от класса `FileSystemInfo`, поэтому оба имеют такие члены, как `LastAccessTime` и `Delete`, как показано на следующей схеме (рис. 9.4).

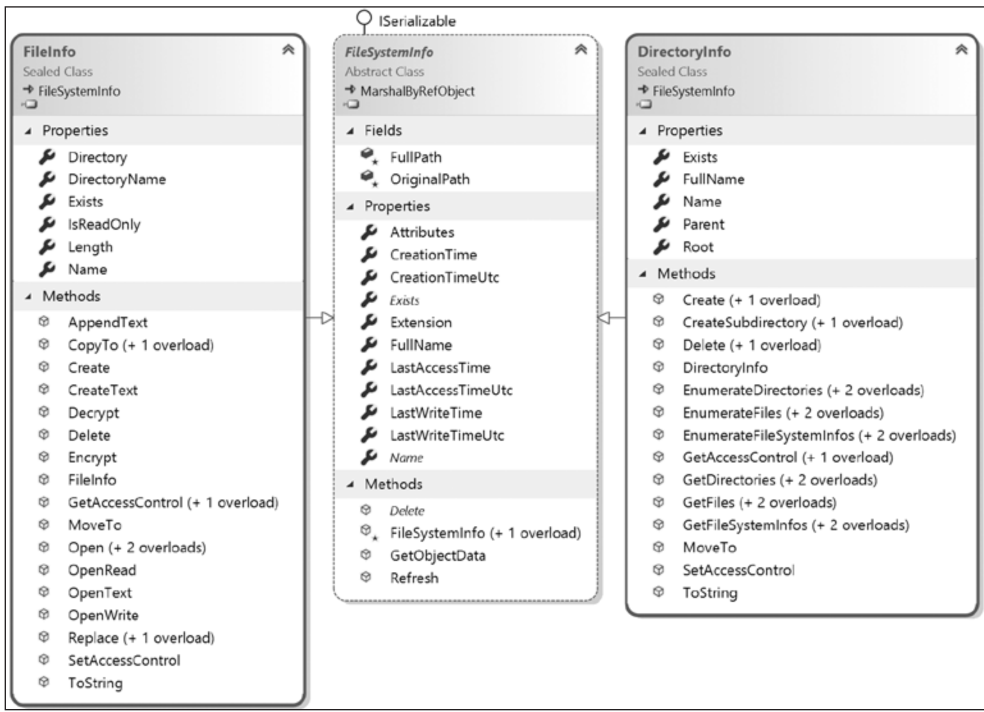


Рис. 9.4. Список свойств и методов для файлов и каталогов

Напишем код, использующий экземпляр `FileInfo` для эффективного выполнения нескольких действий с файлом.

1. В целях эффективного выполнения нескольких действий с файлом напишем код, использующий экземпляр класса `FileInfo`. Добавьте операторы в конец метода `WorkWithFiles`, чтобы создать экземпляр класса `FileInfo` для резервного файла и записать информацию о нем в консоль:

```
var info = new FileInfo(backupFile);
WriteLine($"{backupFile}:");
WriteLine($"Contains {info.Length} bytes");
```

```
WriteLine($"Last accessed {info.LastAccessTime}");
WriteLine($"Has readonly set to {info.IsReadOnly}");
```

2. Запустите консольное приложение и проанализируйте результат:

```
/Users/markjprice/Code/Chapter09/OutputFiles/Dummy.bak:
Contains 11 bytes
Last accessed 26/11/2018 09:08:26
Has readonly set to False
```

Количество байтов может отличаться в вашей операционной системе, поскольку операционные системы могут использовать разные окончания строки.

Контроль работы с файлами

При работе с файлами возникает необходимость контролировать, как именно они открываются. Метод `File.Open` содержит перегрузки для указания дополнительных параметров с помощью значений `enum`. Ниже приведены используемые типы `enum`:

- `FileMode` — определяет, что вы хотите сделать с файлом, например `CreateNew` (Создать новый), `OpenOrCreate` (Открыть файл, если он существует, или создать новый) или `Truncate` (Открыть с удалением текущего содержимого);
- `FileAccess` — определяет, какой уровень доступа вам нужен, например `ReadWrite` (Чтение и запись);
- `FileShare` — управляет блокировками файла, чтобы разрешить другим процессам указанный уровень доступа, например `Read`.

Возможно, вы захотите открыть файл и прочитать его, а также разрешить его считывать другим процессам:

```
FileStream file = File.Open(pathToFile,
  FileMode.Open, FileAccess.Read, FileShare.Read);
```

Существует также тип `enum` для атрибутов файла:

- `FileAttributes` — используется для проверки значения свойства `Attributes` типов, производных от `FileSystemInfo`. Например, выставлены ли для файла флаги `Archive` и `Encrypted`.

Вы можете проверить атрибуты файла или каталога:

```
var info = new FileInfo(backupFile);
WriteLine("Is the backup file compressed? {0}",
  info.Attributes.HasFlag(FileAttributes.Compressed));
```

Чтение и запись с помощью потоков

Поток (stream)[1] представляет собой последовательность байтов, которую можно считать или в которую можно записать некие данные. Хотя файлы могут обрабатываться во многом подобно массивам, с произвольным доступом по известной позиции байта в файле, считается полезным обрабатывать файлы как поток, в котором байты могут быть доступны в последовательном порядке.

Кроме того, потоки могут использоваться для обработки входных и выходных данных терминала и сетевых ресурсов, таких как сокеты и порты, которые не обеспечивают произвольный доступ и не могут искать расположение. Вы можете написать код для обработки произвольных байтов, не зная и не заботясь о том, откуда они берутся. Ваш код просто считывает или записывает в поток, а другой фрагмент кода определяет, где байты хранятся фактически.

Существует абстрактный класс `Stream`, представляющий собой поток. Есть множество классов, которые наследуются от этого базового, включая `FileStream`, `MemoryStream`, `BufferedStream`, `GZipStream` и `SslStream`, и потому все они работают одинаково.

Все потоки реализуют интерфейс `IDisposable`, поэтому имеют метод `Dispose` для освобождения неуправляемых ресурсов.

В табл. 9.1 приведены некоторые из универсальных членов класса `Stream`.

Таблица 9.1

Член	Описание
CanRead, CanWrite	Определяет, поддерживает ли текущий поток возможность чтения и записи соответственно
Length, Position	Определяет длину потока в байтах и текущую позицию в потоке соответственно. Эти свойства могут вызвать исключение для некоторых типов потоков
Dispose()	Закрывает поток и освобождает его ресурсы
Flush()	Если поток имеет буфер, то байты в нем записываются в поток и буфер очищается
Read(), ReadAsync()	Считывает определенное количество байтов из потока в байтовый массив и перемещает позицию синхронно и асинхронно соответственно

[1] В русскоязычной терминологии слово «поток» используется для обозначения англоязычных понятий stream (дословно: «поток», «струя», «течение») и thread (дословно: «нить»).

Член	Описание
ReadByte()	Считывает байт из потока и перемещает позицию
Seek()	Задает позицию в текущем потоке (если значение CanSeek истинно)
Write(), WriteAsync()	Записывает последовательность байтов в текущий поток синхронно и асинхронно соответственно
WriteByte()	Записывает байт в поток

В табл. 9.2 приведены некоторые *запоминающие потоки*, предоставляющие место хранения байтов.

Таблица 9.2

Пространство имен	Класс	Описание
System.IO	FileStream	Создает поток, хранилищем которого является файловая система
System.IO	Memory Stream	Создает поток, хранилищем которого является память
System.Net.Sockets	NetworkStream	Создает поток, хранилищем которого является сеть

В табл. 9.3 приведены некоторые *функциональные потоки*, которые не могут существовать сами по себе. Их можно «подключить» к другим потокам, чтобы расширить их функциональность.

Таблица 9.3

Пространство имен	Класс	Описание
System.Security.Cryptography	CryptoStream	Шифрует и дешифрует поток
System.IO.Compression	GZipStream, DeflateStream	Сжимает и распаковывает поток
System.Net.Security	AuthenticatedStream	Передает учетные данные через поток

Временами бывает необходимо работать с потоками на низком уровне. Чаще всего, однако, удается упростить задачу, добавив в цепочку специальные вспомогательные классы.

Все вспомогательные типы для потоков реализуют интерфейс `IDisposable`, поэтому имеют метод `Dispose` для освобождения неуправляемых ресурсов.

В табл. 9.4 представлены некоторые вспомогательные классы для обработки распространенных сценариев.

Таблица 9.4

Пространство имен	Класс	Описание
System.IO	StreamReader	Считывает данные из потока в текстовом формате
	StreamWriter	Записывает данные в поток в текстовом формате
	BinaryReader	Считывает данные из потока в виде типов .NET. Например, метод ReadDecimal считывает следующие 16 байт из базового потока в виде десятичного значения, а метод ReadInt32 считывает следующие четыре байта как значение int
	BinaryWriter	Записывает данные в поток в виде типов .NET. Например, метод Write с параметром типа decimal записывает 16 байт в базовый поток, а метод Write с параметром типа int записывает четыре байта
System.Xml	XmlReader	Считывает данные из потока в XML-формате
	XmlWriter	Записывает данные в поток в XML-формате

Запись в текстовые потоки

Напишем код для записи текста в поток.

1. Создайте проект консольного приложения `WorkingWithStreams`, добавьте его в рабочую область `Chapter09` и выберите проект как активный для OmniSharp.
2. Импортируйте пространства имен `System.IO` и `System.Xml`, статически импортируйте типы `System.Console`, `System.Environment` и `System.IO.Path`.
3. Определите массив строковых значений позывных пилота вертолета Viper и создайте метод `WorkWithText`, перечисляющий позывные, записывая каждый из них в текстовый файл:

```
// определение массива позывных пилота Viper
static string[] callsigns = new string[] {
  "Husker", "Starbuck", "Apollo", "Boomer",
  "Bulldog", "Athena", "Helo", "Racetrack" };

staticvoidWorkWithText()
{
  // определение файла для записи
  string textFile = Combine(CurrentDirectory, "streams.txt");

  // создание текстового файла
  // и возвращение вспомогательного объекта для записи
  StreamWriter text = File.CreateText(textFile);
```

```
    // перечисление строк с записью каждой из них
    // в поток в отдельной строке
    foreach (string item in callsigns)
    {
      text.WriteLine(item);
    }
    text.Close(); // освобождение ресурсов

    // вывод содержимого файла в консоль
    WriteLine("{0} contains {1:N0} bytes.",
      arg0: textFile,
      arg1: new FileInfo(textFile).Length);

    WriteLine(File.ReadAllText(textFile));
  }
```

4. В методе `Main` вызовите метод `WorkWithText`.
5. Запустите приложение и проанализируйте результат:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.txt
contains 60 bytes.
Husker
Starbuck
Apollo
Boomer
Bulldog
Athena
Helo
Racetrack
```

6. Откройте созданный файл и убедитесь, что он содержит список позывных.

Запись в XML-потоки

Существует два способа написания XML элемента:

- `WriteStartElement` и `WriteEndElement` — используется, когда элемент содержит дочерние элементы;
- `WriteElementString` — используется, когда у элемента дочерние элементы отсутствуют.

Теперь сохраним тот же массив строковых значений в XML-файле.

1. Создайте метод `WorkWithXml`, перечисляющий позывные пилота вертолета Viper, как показано ниже:

```
Static void WorkWithXml()
{
```

```
    // определение файла для записи
    string xmlFile = Combine(CurrentDirectory, "streams.xml");

    // создание файлового потока
    FileStream xmlFileStream = File.Create(xmlFile);

    // оборачивание файлового потока во вспомогательный объект
    // для записи XML и автоматическое добавление
    // отступов для вложенных элементов
    XmlWriter xml = XmlWriter.Create(xmlFileStream,
      new XmlWriterSettings { Indent = true });

    // запись объявления XML
    xml.WriteStartDocument();

    // запись корневого элемента
    xml.WriteStartElement("callsigns");

    // перечисление строк с записью каждой из них в поток
    foreach (string item in callsigns)
    {
      xml.WriteElementString("callsign", item);
    }

    // запись закрывающего корневого элемента
    xml.WriteEndElement();

    // закрытие вспомогательного объекта и потока
    xml.Close();
    xmlFileStream.Close();

    // вывод содержимого файла
    WriteLine("{0} contains {1:N0} bytes.",
      arg0: xmlFile,
      arg1: new FileInfo(xmlFile).Length);

    WriteLine(File.ReadAllText(xmlFile));
  }
```

2. В методе `Main` закомментируйте предыдущий вызов метода и добавьте вызов в метод `WorkWithXml`.
3. Запустите консольное приложение и проанализируйте результат:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.xml
contains 310 bytes.
<?xml version="1.0" encoding="utf-8"?>
<callsigns>
  <callsign>Husker</callsign>
  <callsign>Starbuck</callsign>
  <callsign>Apollo</callsign>
  <callsign>Boomer</callsign>
  <callsign>Bulldog</callsign>
```

```
  <callsign>Athena</callsign>
  <callsign>Helo</callsign>
  <callsign>Racetrack</callsign>
</callsigns>
```

Освобождение файловых ресурсов

При открытии файла для чтения или записи вы используете ресурсы вне .NET. Эти ресурсы называются неуправляемыми и должны быть освобождены по окончании работы с ними. Чтобы гарантировать их освобождение, можно вызывать метод `Dispose` внутри блока `finally`.

Чтобы правильно распоряжаться неуправляемыми ресурсами, отредактируем наш предыдущий код.

1. Измените метод `WorkWithXml`, как показано ниже (выделено полужирным шрифтом):

```
static void WorkWithXml()
{
  FileStream xmlFileStream = null;
  XmlWriterxml = null;

  Try
  {
    // определение файла для записи
    string xmlFile = Combine(CurrentDirectory, "streams.xml");

    // создание файлового потока
    xmlFileStream = File.Create(xmlFile);

    // оборачивание файлового потока во вспомогательный объект для записи
    // XML и автоматическое добавление отступов для вложенных элементов
    xml = XmlWriter.Create(xmlFileStream,
      new XmlWriterSettings { Indent = true });

    // запись объявления XML
    xml.WriteStartDocument();

    // запись корневого элемента
    xml.WriteStartElement("callsigns");

    // перечисление строк с записью каждой из них в поток
    foreach (string item in callsigns)
    {
      xml.WriteElementString("callsign", item);
    }

    // запись закрывающего корневого элемента
    xml.WriteEndElement();
```

```
      // закрытие вспомогательного объекта и потока
      xml.Close();
      xmlFileStream.Close();

      // вывод содержимого файла
      WriteLine($"{0} contains {1:N0} bytes.",
        arg0: xmlFile,
        arg1: new FileInfo(xmlFile).Length);

      WriteLine(File.ReadAllText(xmlFile));
    }
    catch(Exception ex)
    {
      // если путь не существует, то выбрасывается исключение
      WriteLine($"{ex.GetType()} says {ex.Message}");
    }
    finally
    {
      if (xml != null)
      {
        xml.Dispose();
        WriteLine("The XML writer's unmanaged resources have been disposed.");
      }
      if (xmlFileStream != null)
      {
        xmlFileStream.Dispose();
        WriteLine("The file stream's unmanaged resources have been disposed.");
      }
    }
  }
```

Вы также можете вернуться и изменить другие ранее созданные методы, но я оставлю это для вас в качестве дополнительного упражнения.

2. Запустите консольное приложение и проанализируйте результат:

```
The XML writer's unmanaged resources have been disposed.
The file stream's unmanaged resources have been disposed.
```

Перед вызовом метода Dispose убедитесь, что объект не равен null.

Вы можете упростить код, проверяющий объект на `null`, а затем вызывающий его метод `Dispose` с помощью оператора `using`.

Может немного сбить с толку то, что существует два варианта использования ключевого слова `using`: импорт пространства имен и генерация оператора `finally`, который вызывает метод `Dispose` для объекта, реализующего интерфейс `IDisposable`.

Компилятор изменяет блок оператора `using` на оператор `try-finally` без оператора `catch`. Кроме того, вы можете использовать вложенные операторы `try`, так что при желании вы все еще можете перехватить какие-либо исключения. Рассмотрим это на примере следующего кода:

```
using (FileStream file2 = File.OpenWrite(
  Path.Combine(path, "file2.txt")))
{
  using (StreamWriter writer2 = new StreamWriter(file2))
  {
    try
    {
      writer2.WriteLine("Welcome, .NET Core!");
    }
    catch(Exception ex)
    {
      WriteLine($"{ex.GetType()} says {ex.Message}");
    }
  } // автоматический вызов метода Dispose, если объект не равен null
} // автоматический вызов метода Dispose, если объект не равен null
```

Сжатие потоков

Формат XML довольно объемный, поэтому занимает больше памяти в байтах, чем обычный текст. Мы можем сжать XML-данные, воспользовавшись всем знакомым алгоритмом сжатия, известным под названием *GZIP*.

1. Импортируйте следующее пространство имен:

   ```
   using System.IO.Compression;
   ```

2. Добавьте метод `WorkWithCompression`, использующий экземпляры `GZipSteam` для создания сжатого файла, содержащего те же элементы XML, что и раньше, а затем распаковывающий его при чтении и выводе в консоль:

   ```
   static void WorkWithCompression()
   {
     // сжатие XML-вывода
     string gzipFilePath = Combine(
       CurrentDirectory, "streams.gzip");

     FileStream gzipFile = File.Create(gzipFilePath);

     using (GZipStream compressor = new GZipStream(
       gzipFile, CompressionMode.Compress))
     {
       using (XmlWriter xmlGzip = XmlWriter.Create(compressor))
       {
   ```

```
        xmlGzip.WriteStartDocument();
        xmlGzip.WriteStartElement("callsigns");

        foreach (string item in callsigns)
        {
          xmlGzip.WriteElementString("callsign", item);
        }
        // вызов метода WriteEndElement необязателен,
        // поскольку, освобождаясь, XmlWriter
        // автоматически закрывает любые элементы
      }
    } // закрытие основного потока

    // выводит все содержимое сжатого файла в консоль
    WriteLine("{0} contains {1:N0} bytes.",
      gzipFilePath, new FileInfo(gzipFilePath).Length);

    WriteLine($"The compressed contents:");
    WriteLine(File.ReadAllText(gzipFilePath));

    // чтение сжатого файла
    WriteLine("Reading the compressed XML file:");
    gzipFile = File.Open(gzipFilePath, FileMode.Open);

    using (GZipStream decompressor = new GZipStream(
      gzipFile, CompressionMode.Decompress))
    {
      using (XmlReader reader = XmlReader.Create(decompressor))
      {
        while (reader.Read()) // чтение сжатого файла
        {
          // проверить, находимся ли мы на элементе с именем callsign
          if ((reader.NodeType == XmlNodeType.Element)
            && (reader.Name == "callsign"))
          {
            reader.Read(); // переход к тексту внутри элемента
            WriteLine($"{reader.Value}"); // чтение его значения
          }
        }
      }
    }
  }
```

3. В методе `Main` оставьте вызов метода `WorkWithXml` и добавьте вызов метода `WorkWithCompression`:

```
static void Main(string[] args)
{
  // WorkWithText();
  WorkWithXml();
  WorkWithCompression();
}
```

4. Запустите консольное приложение и сравните размеры XML-файла и сжатого XML-файла. Обратите внимание: сжатые XML-данные занимают вполовину меньше объема памяти по сравнению с таким же количеством несжатых.

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.xml
contains 310 bytes.
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.gzip
c ontains 150 bytes.
```

Сжатие с помощью алгоритма Бротли

Выпуская платформу .NET Core 2.1, корпорация Microsoft представила реализацию алгоритма сжатия Бротли. По производительности он похож на алгоритм, используемый в DEFLATE и GZIP, но результат, как правило, сжат на 20 % больше.

1. Измените метод `WorkWithCompression`, в котором необязательный параметр указывает, следует ли использовать алгоритм Бротли, и по умолчанию указывает, что алгоритм Бротли следует использовать, как показано ниже (выделено полужирным шрифтом):

```
static void WorkWithCompression(bool useBrotli = true)
{
  string fileExt = useBrotli ? "brotli" : "gzip";

  // сжатие XML-вывода
  string filePath = Combine(
    CurrentDirectory, $"streams.{fileExt}");

  FileStream file = File.Create(filePath);

  Stream compressor;
  if (useBrotli)
  {
    compressor = new BrotliStream(file, CompressionMode.Compress);
  }
  else
  {
    compressor = new GZipStream(file, CompressionMode.Compress);
  }

  using (compressor)
  {
    using (XmlWriter xml = XmlWriter.Create(compressor))
    {
      xml.WriteStartDocument();
      xml.WriteStartElement("callsigns");
      foreach (string item in callsigns)
      {
```

```
        xml.WriteElementString("callsign", item);
      }
    }
  } // закрытие основного потока

  // выводит все содержимое сжатого файла в консоль
  WriteLine("{0} contains {1:N0} bytes.",
    filePath, new FileInfo(filePath).Length);
  WriteLine(File.ReadAllText(filePath));

  // чтение сжатого файла
  WriteLine("Reading the compressed XML file:");
  file = File.Open(filePath, FileMode.Open);

  Stream decompressor;
  if (useBrotli)
  {
    decompressor = new BrotliStream(
      file, CompressionMode.Decompress);
  }
  else
  {
    decompressor = new GZipStream(
      file, CompressionMode.Decompress);
  }

  using (decompressor)
  {
    using (XmlReader reader = XmlReader.Create(decompressor))
    {
      while (reader.Read())
      {
        // проверить, находимся ли мы на элементе с именем callsign
        if ((reader.NodeType == XmlNodeType.Element)
          && (reader.Name == "callsign"))
        {
          reader.Read(); // переход к тексту внутри элемента
          WriteLine($"{reader.Value}"); // чтение его значения
        }
      }
    }
  }
}
```

2. Измените метод `Main`, чтобы он дважды вызывал метод `WorkWithCompression`, один раз по умолчанию с помощью алгоритма Бротли и один раз с использованием утилиты GZIP:

```
WorkWithCompression();
WorkWithCompression(useBrotli: false);
```

3. Запустите консольное приложение и сравните размеры двух сжатых файлов XML. Благодаря использованию алгоритма Бротли сжатые XML-данные занимают почти на 21 % меньше места, как показано в следующем отредактированном выводе:

```
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.brotli
contains 118 bytes.
/Users/markjprice/Code/Chapter09/WorkingWithStreams/streams.gzip
contains 150 bytes.
```

Высокопроизводительные потоки с использованием конвейеров

Выпуская платформу .NET Core 2.1, корпорация Microsoft представила *конвейеры*. Для правильной обработки данных из потока требуется написать сложный стандартный код, который трудно сопровождать. Тестирование на вашем локальном ноутбуке часто выполняется успешно с небольшими тестовыми файлами, но в действительности код может не работать из-за неправильных предположений. Разобраться с этой проблемой помогут конвейеры.

Более подробно о конвейерах можно прочитать на сайте https://devblogs.microsoft.com/dotnet/system-io-pipelines-high-performance-io-in-net/.

Асинхронные потоки

В .NET Core 3.0 корпорация Microsoft ввела асинхронную обработку потоков, о которой вы узнаете в главе 13.

Учебное пособие по асинхронным потокам можно получить по следующей ссылке: https://docs.microsoft.com/ru-ru/dotnet/csharp/tutorials/generate-consume-asynchronous-stream.

Кодирование и декодирование текста

Текстовые символы могут быть представлены разными способами. Например, алфавит можно закодировать с помощью азбуки Морзе в серии точек и тире для передачи по телеграфной линии.

Аналогичным образом текст в памяти компьютера сохраняется в виде битов (единиц и нулей), представляющих кодовую точку в кодовом пространстве. Большинство кодовых точек представляет один текстовый символ, но некоторые могут иметь и другое значение. Например, форматирование.

Например, таблица ASCII имеет кодовое пространство со 128 кодовыми точками. Платформа .NET использует стандарт *Unicode* для внутреннего кодирования текста. Данный стандарт содержит более миллиона кодовых точек.

Иногда возникает необходимость переместить текст за пределы .NET для использования системами, не применяющими Unicode или другую вариацию стандарта Unicode, поэтому важно научиться преобразовывать кодировки.

В табл. 9.5 перечислены некоторые альтернативные кодировки текста, обычно используемые на компьютерах.

Таблица 9.5

Кодировка	Описание
ASCII	Кодирует ограниченный диапазон символов, используя семь младших битов байта
UTF-8	Представляет каждую кодовую точку Unicode в виде последовательности от одного до четырех байтов
UTF-7	Спроектирована как более эффективная, чем UTF-8, при работе с 7-битовыми каналами, но имеет целый набор проблем с безопасностью и надежностью, так что UTF-8 является более предпочтительной
UTF-16	Представляет каждую кодовую точку Unicode в виде последовательности из одного или двух 16-битных целых чисел
UTF-32	Представляет каждую кодовую точку Unicode в виде 32-битного целого числа и, следовательно, является кодировкой фиксированной длины в отличие от других кодировок Unicode с переменной длиной
Кодировки ANSI и ISO	Предоставляет поддержку ряда кодовых страниц для поддержки конкретного языка или группы языков

В большинстве случаев в настоящее время кодировка UTF-8 считается хорошим вариантом по умолчанию, и поэтому она и выбрана по умолчанию в .NET. То есть именно она представлена в `Encoding.Default`.

Преобразование строк в последовательности байтов

Рассмотрим примеры кодировки текста.

1. Создайте проект консольного приложения `WorkingWithEncodings`, добавьте его в рабочую область `Chapter09` и определите проект как активный для OmniSharp.
2. Импортируйте пространство имен `System.Text` и статически импортируйте класс `Console`.

3. Добавьте операторы в метод `Main` для кодирования строки, применяя выбранную пользователем кодировку. Переберите каждый байт, а затем декодируйте его обратно в строку и выведите результат:

```
WriteLine("Encodings");
WriteLine("[1] ASCII");
WriteLine("[2] UTF-7");
WriteLine("[3] UTF-8");
WriteLine("[4] UTF-16 (Unicode)");
WriteLine("[5] UTF-32");
WriteLine("[any other key] Default");

// выбор кодировки
Write("Press a number to choose an encoding: ");
ConsoleKey number = ReadKey(intercept: false).Key;
WriteLine();
WriteLine();

Encoding encoder = number switch
{
  ConsoleKey.D1 => Encoding.ASCII,
  ConsoleKey.D2 => Encoding.UTF7,
  ConsoleKey.D3 => Encoding.UTF8,
  ConsoleKey.D4 => Encoding.Unicode,
  ConsoleKey.D5 => Encoding.UTF32,
  _             => Encoding.Default
};

// определение строки для кодирования
string message = "A pint of milk is £1.99";

// кодирование строки в последовательность байтов
byte[] encoded = encoder.GetBytes(message);

// проверка количества байтов, необходимого для кодирования
WriteLine("{0} uses {1:N0} bytes.",
  encoder.GetType().Name, encoded.Length);

// перечисление каждого байта
WriteLine($"BYTE HEX CHAR");
foreach (byte b in encoded)
{
WriteLine($"{b,4} {b.ToString("X"),4} {(char)b,5}");
}

// декодирование последовательности байтов обратно
// в строку и ее вывод
string decoded = encoder.GetString(encoded);
WriteLine(decoded);
```

4. Запустите приложение и обратите внимание на предупреждение, чтобы не использовать UTF7, поскольку это небезопасно. Конечно, если вам нужно

сгенерировать текст, используя данную кодировку для совместимости с другой системой, она должна оставаться опцией в .NET.

5. Запустите приложение и нажмите клавишу `1` для выбора кодировки ASCII. Обратите внимание: при выводе байтов символ фунта стерлинга (`£`) не может быть представлен в кодировке ASCII, поэтому вместо него используется знак вопроса (`?`).

```
ASCIIEncodingSealed uses 23 bytes.
BYTE HEX CHAR
  65  41    A
  32  20
 112  70    p
 105  69    i
 110  6E    n
 116  74    t
  32  20
 111  6F    o
 102  66    f
  32  20
 109  6D    m
 105  69    i
 108  6C    l
 107  6B    k
  32  20
 105  69    i
 115  73    s
  32  20
  63  3F    ?
  49  31    1
  46  2E    .
  57  39    9
  57  39    9
A pint of milk is ?1.99
```

6. Перезапустите приложение и нажмите клавишу `3` для выбора кодировки UTF-8. Обратите внимание: кодировке UTF-8 для хранения данных требуется дополнительный байт (24 байта вместо 23), но она корректно сохраняет символ `£`.
7. Перезапустите приложение и нажмите клавишу `4` для выбора кодировки UTF-16. Обратите внимание: кодировке UTF-16 для хранения каждого символа требуется два байта (всего 46 байт), но она корректно сохраняет символ `£`. Данная кодировка используется внутри .NET для хранения значений `char` и `string`.

Кодирование и декодирование текста в файлах

При использовании вспомогательных классов потоков, таких как `StreamReader` и `StreamWriter`, вы можете указать предпочтительную кодировку. При осуществлении записи во вспомогательный поток строки будут кодироваться автоматически, а при чтении из вспомогательного потока — автоматически декодироваться.

Чтобы указать кодировку, передайте ее в качестве второго параметра конструктору вспомогательного типа:

```
var reader = new StreamReader(stream, Encoding.UTF7);
var writer = new StreamWriter(stream, Encoding.UTF7);
```

Зачастую вы не будете иметь возможности выбирать кодировку, так как будете генерировать файл для использования в другой системе. Но если возможность есть, то выбирайте такую кодировку, которая занимает наименьший объем памяти (количество байтов), но поддерживает все необходимые символы.

Сериализация графов объектов

Сериализация — процесс преобразования объекта в поток байтов в выбранном формате. Обратный процесс называется *десериализацией*.

Существуют десятки доступных для выбора форматов, среди которых два наиболее распространенных — это *расширяемый язык разметки* (eXtensible Markup Language, XML) и *объектная нотация JavaScript* (JavaScript Object Notation, JSON).

Формат JSON компактнее и лучше подходит для веб- и мобильных приложений. Формат XML более объемный, но лучше поддерживается устаревшими системами. Используйте формат JSON для минимизации размера сериализованных графов объектов. Он также отлично подойдет при отправке графов объектов в веб- и мобильные приложения, поскольку JSON — собственный формат сериализации языка JavaScript, а мобильные приложения часто выполняют вызовы с ограниченной пропускной способностью, поэтому количество байтов важно.

Платформа .NET Core содержит несколько классов, которые умеют сериализовать в форматы XML и JSON (и из них). Мы начнем изучение с классов `XmlSerializer` и `JsonSerializer`.

XML-сериализация

Начнем с изучения XML, вероятно наиболее используемого в мире формата сериализации (на данный момент). В качестве универсального примера мы определим пользовательский класс для хранения информации о человеке, а затем создадим граф объектов, задействуя список экземпляров `Person` с вложением.

1. Создайте проект консольного приложения `WorkingWithSerialization`, добавьте его в рабочую область `Chapter09` и определите проект как активный для OmniSharp.

2. Добавьте класс `Person` со свойством `Salary`. Обратите внимание: свойство `Salary` объявлено защищенным (`protected`), то есть доступно только внутри самого класса и производных классов. Для указания зарплаты класс содержит конструктор с единственным параметром, устанавливающим начальный оклад:

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

namespace Packt.Shared
{
  public class Person
  {
    public Person(decimal initialSalary)
    {
      Salary = initialSalary;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public HashSet<Person> Children { get; set; }
    protected decimal Salary { get; set; }
  }
}
```

3. Вернитесь к файлу `Program.cs` и импортируйте следующие пространства имен:

```
using System;                      // DateTime
using System.Collections.Generic; // List<T>, HashSet<T>
using System.Xml.Serialization;   // XmlSerializer
using System.IO;                   // FileStream
using Packt.Shared;                // Person
using static System.Console;
using static System.Environment;
using static System.IO.Path;
```

4. Добавьте следующие операторы в метод `Main`:

```
// создание графа объектов
var people = new List<Person>
{
  new Person(30000M) { FirstName = "Alice",
    LastName = "Smith",
    DateOfBirth = new DateTime(1974, 3, 14) },
  new Person(40000M) { FirstName = "Bob",
    LastName = "Jones",
    DateOfBirth = new DateTime(1969, 11, 23) },
  new Person(20000M) { FirstName = "Charlie",
    LastName = "Cox",
    DateOfBirth = new DateTime(1984, 5, 4),
```

```
    Children = new HashSet<Person>
    { new Person(0M) { FirstName = "Sally",
      LastName = "Cox",
      DateOfBirth = new DateTime(2000, 7, 12) } } }
};

// создание объекта, который отформатирует список людей в XML
var xs = new XmlSerializer(typeof(List<Person>));

// создание файла для записи
string path = Combine(CurrentDirectory, "people.xml");

using (FileStream stream = File.Create(path))
{
  // сериализация графа объектов в поток
  xs.Serialize(stream, people);
}

WriteLine("Written {0:N0} bytes of XML to {1}",
  arg0: new FileInfo(path).Length,
  arg1: path);

WriteLine();

// отображение сериализованного графа объектов
WriteLine(File.ReadAllText(path));
```

5. Запустите консольное приложение и проанализируйте результат. Обратите внимание на вызываемое исключение:

```
Unhandled Exception: System.InvalidOperationException: Packt.
Shared.Person cannot be serialized because it does not have a
parameterless constructor.
```

6. Вернитесь к файлу `Person.cs` и добавьте следующий оператор для определения конструктора без параметров:

```
public Person() { }
```

Обратите внимание: конструктору ничего не требуется делать, однако он должен присутствовать, чтобы класс `XmlSerializer` мог его вызвать для создания экземпляров `Person` в процессе десериализации.

7. Перезапустите консольное приложение и проанализируйте результат. Обратите внимание, что граф объектов сериализуется в формате XML, а свойство `Salary` не включено в результат:

```
Written 752 bytes of XML to
/Users/markjprice/Code/Chapter09/WorkingWithSerialization/people.xml
<?xml version="1.0"?>
```

```
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Person>
    <FirstName>Alice</FirstName>
    <LastName>Smith</LastName>
    <DateOfBirth>1974-03-14T00:00:00</DateOfBirth>
  </Person>
  <Person>
    <FirstName>Bob</FirstName>
    <LastName>Jones</LastName>
    <DateOfBirth>1969-11-23T00:00:00</DateOfBirth>
  </Person>
  <Person>
    <FirstName>Charlie</FirstName>
    <LastName>Cox</LastName>
    <DateOfBirth>1984-05-04T00:00:00</DateOfBirth>
    <Children>
      <Person>
        <FirstName>Sally</FirstName>
        <LastName>Cox</LastName>
        <DateOfBirth>2000-07-12T00:00:00</DateOfBirth>
      </Person>
    </Children>
  </Person>
</ArrayOfPerson>
```

Генерация компактного XML

Формат XML можно использовать более эффективно, если для некоторых полей вместо элементов применить атрибуты.

1. В файле `Person.cs` импортируйте пространство имен `System.Xml.Serialization`.
2. Измените все свойства, кроме `Children`, добавив атрибут `[XmlAttribute]`, и установите короткое имя для каждого свойства:

```
[XmlAttribute("fname")]
public string FirstName { get; set; }

[XmlAttribute("lname")]
public string LastName { get; set; }

[XmlAttribute("dob")]
public DateTime DateOfBirth { get; set; }
```

3. Перезапустите приложение и обратите внимание: размер файла был уменьшен с 752 до 462 байт, что экономит пространство более чем на треть, как показано ниже:

```
Written 462 bytes of XML to /Users/markjprice/Code/Chapter09/
WorkingWithSerialization/people.xml
<?xml version="1.0"?>
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchemainstance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Person fname="Alice" lname="Smith" dob="1974-03-14T00:00:00" />
  <Person fname="Bob" lname="Jones" dob="1969-11-23T00:00:00" />
  <Person fname="Charlie" lname="Cox" dob="1984-05-04T00:00:00">
    <Children>
      <Person fname="Sally" lname="Cox" dob="2000-07-12T00:00:00" />
    </Children>
  </Person>
</ArrayOfPerson>
```

XML-десериализация

Рассмотрим десериализацию файлов XML обратно в активные объекты в памяти.

1. Добавьте операторы в конец метода `Main` для открытия файла XML. Затем необходимо десериализовать его:

```
using (FileStream xmlLoad = File.Open(path, FileMode.Open))
{
  // десериализация и приведение графа объектов к списку людей
  var loadedPeople = (List<Person>)xs.Deserialize(xmlLoad);

  foreach (var item in loadedPeople)
  {
    WriteLine("{0} has {1} children.",
      item.LastName, item.Children.Count);
  }
}
```

2. Перезапустите приложение и обратите внимание, что информация о людях успешно загружается из XML-файла:

```
Smith has 0 children.
Jones has 0 chil dren.
Cox has 1 children.
```

Существует множество других атрибутов, которые могут использоваться при работе с форматом XML.

При использовании класса XmlSerializer помните, что учитываются только открытые (public) поля и свойства, а тип должен содержать конструктор без параметров. Кроме того, с помощью атрибутов можно настроить вывод.

JSON-сериализация

Одна из наиболее популярных библиотек .NET для работы с форматом сериализации JSON — `Newtonsoft.Json`, известная как *Json.NET*. Это эффективная библиотека для работы с JSON.

Рассмотрим ее на примере.

1. Отредактируйте файл `WorkingWithSerialization.csproj`, чтобы добавить ссылку на пакет для последней версии `Newtonsoft.Json`, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json"
      Version="12.0.3" />
  </ItemGroup>
</Project>
```

Чтобы найти последнюю поддерживаемую версию, вам необходимо найти пакеты NuGet в канале Microsoft NuGet, как показано на сайте https://www.nuget.org/packages/Newtonsoft.Json/.

2. Добавьте в конец метода `Main` следующие операторы, чтобы создать текстовый файл. Затем необходимо выполнить сериализацию людей в файл, используя JSON:

```
// создание файла для записи
string jsonPath = Combine(CurrentDirectory, "people.json");

using (StreamWriter jsonStream = File.CreateText(jsonPath))
{
  // создание объекта для форматирования в JSON
  var jss = new Newtonsoft.Json.JsonSerializer();

  // сериализация графа объектов в строку
  jss.Serialize(jsonStream, people);
}

WriteLine();
WriteLine("Written {0:N0} bytes of JSON to: {1}",
  arg0: new FileInfo(jsonPath).Length,
  arg1: jsonPath);

// отображение сериализованного графа объектов
WriteLine(File.ReadAllText(jsonPath));
```

3. Перезапустите приложение и обратите внимание, что формат JSON занимает более чем вполовину меньше байтов памяти по сравнению с форматом XML. Его размер даже меньше, чем XML с атрибутами:

```
Written 366 bytes of JSON to: /Users/markjprice/Code/Chapter09/
WorkingWithSerialization/people.json
[{"FirstName":"Alice","LastName":"Smith","DateOfBirth":"1974-03-
14T00:00:00","Children":null},{"FirstName":"Bob","LastName":"Jones",
"DateOfBirth":"1969-11-23T00:00:00","Children":null},{"FirstName":
"Charlie","LastName":"Cox","DateOfBirth":"1984-05-04T00:00:00",
"Children":[{"FirstName":"Sally","LastName":"Cox","DateOfBirth":
"2000-07-12T00:00:00","Children":null}]}]
```

Высокопроизводительный процессор JSON

Платформа .NET Core 3.0 представляет новое пространство имен для работы с JSON — `System.Text.Json`. Оно оптимизировано в целях повышения производительности благодаря использованию таких API, как `Span<T>`.

Кроме того, Json.NET реализован через чтение UTF-16. Было бы более эффективным читать и записывать документы JSON с помощью UTF-8, поскольку большинство сетевых протоколов, включая HTTP, используют UTF-8 и вы можете избежать перекодирования UTF-8 в строковые значения Unicode в Json.NET и обратно. Выпуск новых API позволил корпорации Microsoft добиться улучшения в 1,3–5 раз в зависимости от сценария.

Более подробную информацию о новых API System.Text.Json можно найти на сайте https://devblogs.microsoft.com/dotnet/try-the-new-system-text-json-apis/.

Первый автор Json.NET Джеймс Ньютон-Кинг присоединился к корпорации Microsoft и работал над созданием новых типов JSON. Как он сказал в комментарии при обсуждении новых API для JSON, «Json.NET никуда не уходит» (рис. 9.5).

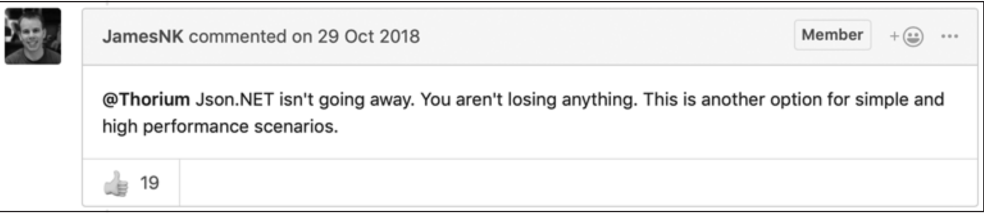


Рис. 9.5. Комментарий автора Json.NET

Более подробную информацию о проблемах, решаемых новыми JSON-API, включая комментарии Джеймса, можно найти на сайте https://github.com/dotnet/corefx/issues/33115.

Рассмотрим новые JSON-API на следующем примере.

1. Импортируйте пространство имен `System.Threading.Tasks`.
2. Измените метод `Main`, чтобы разрешить ожидание задач, изменив `void` на `async Task`, как показано ниже (выделено полужирным шрифтом):

```
static async Task Main(string[] args)
```

3. Импортируйте новый класс JSON для выполнения сериализации с помощью псевдонима, чтобы избежать конфликта по совпадению имен с ранее использованным Json.NET:

```
using NuJson = System.Text.Json.JsonSerializer;
```

4. Добавьте операторы, чтобы открыть файл JSON, десериализовать его и вывести в результате имена и количество людей:

```
using (FileStream jsonLoad = File.Open(
  jsonPath, FileMode.Open))
{
  // десериализация графа объектов в список Person
  var loadedPeople = (List<Person>)

  await NuJson.DeserializeAsync(
    utf8Json: jsonLoad,
    returnType: typeof(List<Person>));

  foreach (var item in loadedPeople)
  {
    WriteLine("{0} has {1} children.",
      item.LastName, item.Children?.Count ?? 0);
  }
}
```

5. Запустите консольное приложение и проанализируйте результат:

```
Smith has children.
Jones has children.
Cox has 1 children.
```

Выбирайте Json.NET для повышения производительности труда разработчиков и большого набор функций, а System.Text.Json — для высокой производительности кода.

В .NET 5 Microsoft добавила уточнения к типам в пространстве имен `System.Text.Json`, например методы расширения для `HttpResponse`, которые вы изучите в главе 18.

Если у вас есть код, использующий библиотеку `Newtonsoft Json.NET`, и вы хотите перейти в новое пространство имен `System.Text.Json`, для этого у Microsoft имеется специальная документация.

Как перейти с Newtonsoft.Json на System.Text.Json вы можете прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/serialization/system-text-json- мигрировать из newtonsoft-how-to.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 9.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Чем применение класса `File` отличается от использования класса `FileInfo`?
2. В чем разница между методами потока `ReadByte` и `Read`?
3. В каких случаях применяются классы `StringReader`, `TextReader` и `StreamReader`?
4. Для чего предназначен тип `DeflateStream`?
5. Сколько байтов на символ затрачивается при использовании кодировки UTF-8?
6. Что такое граф объектов?
7. Какой формат сериализации лучше всего подходит для минимизации затрат памяти?
8. Какой формат сериализации лучше всего подходит для кросс-платформенной совместимости?
9. Почему не рекомендуется использовать строковое значение типа `"\Code\Chapter01"` для представления пути и что необходимо выполнить вместо этого?
10. Где можно найти информацию о пакетах NuGet и их зависимостях?

Упражнение 9.2. XML-сериализация

Создайте консольное приложение `Exercise02`, генерирующее список фигур, выполняющее XML-сериализацию для сохранения его в файловой системе, а затем десериализующее его обратно.

```
// создание списка фигур для сериализации
var listOfShapes = new List<Shape>
{
  new Circle { Colour = "Red", Radius = 2.5 },
  new Rectangle { Colour = "Blue", Height = 20.0, Width = 10.0 },
  new Circle { Colour = "Green", Radius = 8.0 },
  new Circle { Colour = "Purple", Radius = 12.3 },
  new Rectangle { Colour = "Blue", Height = 45.0, Width = 18.0 }
};
```

Фигуры должны содержать свойство только для чтения `Area`, чтобы при десериализации вы могли выводить список фигур, включая их площадь:

```
List<Shape> loadedShapesXml =
  serializerXml.Deserialize(fileXml) as List<Shape>;
foreach (Shape item in loadedShapesXml)
{
  WriteLine("{0} is {1} and has an area of {2:N2}",
    item.GetType().Name, item.Colour, item.Area);
}
```

После запуска приложения вывод должен выглядеть примерно так:

```
Loading shapes from XML:
Circle is Red and has an area of 19.63
Rectangle is Blue and has an area of 200.00
Circle is Green and has an area of 201.06
Circle is Purple and has an area of 475.29
Rectangle is Blue and has an area of 810.00
```

Упражнение 9.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- файловая система и реестр (руководство по программированию на C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/file-system/;
- кодировка символов на платформе .NET: https://docs.microsoft.com/ru-ru/dotnet/articles/standard/base-types/character-encoding;
- сериализация (C#): https://docs.microsoft.com/ru-ru/dotnet/articles/csharp/programming-guide/concepts/serialization/;

- сериализация в файлы и объекты `TextWriters` и `XmlWriters`: https://docs.microsoft.com/ru-ru/dotnet/standard/linq/serialize-files-textwriters-xmlwriters;
- `Newtonsoft Json.NET`: https://www.newtonsoft.com/json.

Резюме

Вы узнали, как осуществлять чтение и запись в текстовые файлы и формат XML, сжимать и распаковывать файлы, кодировать/декодировать текст и сериализовать объекты в форматы JSON и XML (и десериализовать их обратно).

Далее вы займетесь защитой данных и файлов с помощью хеширования, шифрования подписи, аутентификации и авторизации.

10 Защита данных и приложений

Текущая глава посвящена защите данных от просмотра злоумышленником с помощью шифрования и от изменения и повреждения с помощью хеширования и цифровых подписей.

В .NET Core 2.1 корпорация Microsoft представила новые криптографические API, основанные на `Span<T>`, для хеширования, генерации случайных чисел, генерации и обработки асимметричной подписи и шифрования RSA.

Криптографические операции реализуются операционной системой, поэтому, когда в операционной системе исправлена уязвимость системы безопасности, приложения .NET немедленно получают выгоду. Однако это означает, что эти .NET-приложения могут использовать только те функции, которые поддерживает операционная система.

О том, какие функции и какой операционной системой поддерживаются, вы можете прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/security/cross-platform-cryptography.

В этой главе:

- терминология безопасности;
- шифрование и дешифрование данных;
- хеширование данных;
- подписывание данных;
- генерация случайных чисел;
- криптография: что нового;
- аутентификация и авторизация пользователей.

Терминология безопасности

Существует множество способов защиты данных; шесть самых популярных из них перечислены ниже. Вдобавок в этой главе вы ознакомитесь с их подробным описанием и практическими заданиями.

- *Шифрование и дешифрование* — двухсторонний процесс преобразования читаемого текста в шифрованный и обратно.
- *Хеши* — односторонний процесс генерации хеш-значения для безопасного хранения паролей или обнаружения вредоносных изменений либо повреждений данных.
- *Цифровые подписи* — метод проверки источника поступивших данных (то есть что данные поступили от того, кому вы доверяете) путем верификации цифровой подписи данных с помощью открытого ключа.
- *Аутентификация* — метод идентификации пользователя путем проверки его учетных данных.
- *Авторизация* — метод выдачи допуска на выполнение неких действий или работы с определенными данными путем проверки ролей или групп, к которым принадлежат пользователи.

Если безопасность имеет для вас значение (а так и должно быть!), то следует нанять опытного эксперта по безопасности, а не полагаться на советы из Интернета. Очень легко совершить незаметные ошибки и оставить свои приложения и данные уязвимыми до тех пор, пока не окажется слишком поздно!

Ключи и их размеры

В алгоритмах защиты часто используются *ключи*. Они представлены байтовыми массивами различного размера.

Для обеспечения более надежной защиты выберите больший размер ключа.

Ключи могут быть симметричными (также известны как общие или секретные, поскольку один и тот же ключ используется для шифрования и дешифрования) и асимметричными (пара из открытого и закрытого ключей, в которых открытый используется для шифрования, а только закрытый — для дешифрования).

Алгоритмы шифрования с помощью симметричных ключей быстры и позволяют шифровать большие объемы потоковых данных. Асимметричные алгоритмы шифрования ключей медленны и дают возможность шифровать только небольшие массивы байтов.

В своих проектах применяйте оба способа, используя симметричное шифрование для защиты самих данных и асимметричное — для распространения симметричного ключа. По такому принципу, к примеру, работает криптографический протокол Secure Sockets Layer (SSL) в Интернете.

Ключи в шифровании представлены массивами байтов разного размера.

Векторы инициализации и размеры блоков

Вполне вероятно, что при шифровании больших объемов данных повторяются некоторые из их фрагментов (последовательностей символов). Например, в английском тексте часто применяется последовательность символов `the`, которая каждый раз шифруется как `hQ2`. Умный хакер воспользовался бы этим и упростил бы себе работу по взлому шифра:

```
When the wind blew hard the umbrella broke.
5:s4&hQ2aj#D f9d1d£8fh"&hQ2s0)an DF8SFd#][1
```

Мы можем избежать повторения последовательностей, разделив данные на *блоки*. После шифрования блока из него генерируется значение массива байтов, которое передается в следующий блок с целью настройки алгоритма так, чтобы последовательность `the` шифровалась иначе. Зашифровать первый блок можно при наличии массива байтов для выполнения задачи. Это так называемый *вектор инициализации* (initialization vector, IV).

Чем меньше размер блока, тем более сильный получается шифр.

Соль

Соль представляет собой случайный массив байтов, который используется как дополнительный ввод для односторонней хеш-функции. Если вы не применяете соль при генерации хешей, то при условии, что многие из ваших пользователей регистрируются, указывая `123456` в качестве пароля (по данным на 2016-й год, примерно 8 % пользователей так и делают!), все они имеют одно и то же хеш-значение и их учетная запись будет уязвима для внешнего доступа через подбор пароля по словарю.

Более подробную информацию о словарной атаке можно прочитать на сайте blog.codinghorror.com/dictionary-attacks-101/.

Когда пользователь регистрируется, соль должна генерироваться случайным образом и конкатенироваться с указанным пользователем паролем до того, как будет хеширована. Результат этой операции (но не исходный пароль) сохраняется с солью в базе данных.

Когда пользователь авторизуется в системе и вводит пароль, вы просматриваете соль, объединяете ее с введенным паролем, восстанавливаете хеш и затем сравниваете значение с хешем, хранящимся в базе данных. Если значения совпадают, то пароль введен верно.

Генерация ключей и векторов инициализации

Ключи и векторы инициализации представляют собой массивы байтов. Обеим сторонам, которые хотят обмениваться зашифрованными данными, необходимы значения ключей и векторов инициализации, однако надежный обмен байтовыми массивами может быть затруднен.

Вы можете надежно генерировать ключи и векторы инициализации, используя стандарт *формирования ключа на основе пароля* (password-based key derivation function, PBKDF2). Прекрасно подойдет класс `Rfc2898DeriveBytes`, который принимает пароль, соль и счетчик итераций, а затем генерирует ключи и векторы инициализации, вызывая метод `GetBytes`.

Размер соли должен быть не меньше восьми байт, а счетчик итераций — больше нуля. Минимальное рекомендуемое количество итераций — 1000.

Шифрование и дешифрование данных

На платформе .NET Core доступно несколько алгоритмов шифрования.

Одни алгоритмы реализуются операционной системой, и их имена заканчиваются на `CryptoServiceProvider`, другие полностью реализованы в .NET, и их имена имеют суффикс `Managed`.

Некоторые алгоритмы используют симметричные ключи, а некоторые — асимметричные. Главный асимметричный алгоритм шифрования — RSA.

Для эффективного шифрования или дешифрования большого количества байтов алгоритмы симметричного шифрования используют `CryptoStream`. Асимметричные

алгоритмы могут обрабатывать только небольшое количество байтов, хранящихся в байтовом массиве вместо потока.

Ниже перечислены наиболее распространенные алгоритмы симметричного шифрования. Они наследуются от абстрактного класса `SymmetricAlgorithm`:

- AES;
- DESCryptoServiceProvider;
- TripleDESCryptoServiceProvider;
- RC2CryptoServiceProvider;
- RijndaelManaged.

В случае необходимости написать код для расшифровки неких данных, отправленных внешней системой, вам придется задействовать тот алгоритм, который используется внешней системой для шифрования данных. Или, если вам нужно отправить зашифрованные данные в систему, которая может дешифровать только с помощью специального алгоритма, вы снова не сможете выбрать алгоритм.

Если ваш код выполняет шифрование и дешифрование, то вы можете выбрать алгоритм, который наилучшим образом соответствует вашим требованиям к надежности, производительности и т. д.

Выберите Advanced Encryption Standard (AES) (симметричный алгоритм блочного шифрования), основанный на алгоритме Rijndael для симметричного шифрования. Выберите RSA (алгоритм шифрования с открытым ключом) для асимметричного шифрования. Не путайте RSA с DSA. Digital Signature Algorithm (DSA) (алгоритм цифровой подписи) не может зашифровать данные. Он может генерировать только хеши и подписи.

Симметричное шифрование с помощью алгоритма AES

Чтобы упростить повторное использование обеспечивающего безопасность кода в будущем, мы создадим статический класс `Protector` в своей библиотеке классов.

1. Создайте в папке Code папку Chapter10 с двумя подпапками, CryptographyLib и EncryptionApp.
2. В программе Visual Studio Code сохраните в папке Chapter10 рабочую область `Chapter10`.
3. Добавьте в рабочую область папку CryptographyLib.

4. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал).
5. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new classlib
```

6. Добавьте в рабочую область папку EncryptionApp.
7. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и далее папку EncryptionApp.
8. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new console
```

9. На панели EXPLORER (Проводник) раскройте папку CryptographyLib и переименуйте файл `Class1.cs` в `Protector.cs`.
10. Откройте в папке проекта `EncryptionApp` файл `EncryptionApp.csproj` и добавьте ссылку на библиотеку `CryptographyLib`, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference
      Include="..\CryptographyLib\CryptographyLib.csproj" />
  </ItemGroup>
</Project>
```

11. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet build
```

12. Откройте файл `Protector.cs` и измените его содержимое так, чтобы определить статический класс `Protector` с полями для хранения массива байтов соли и числа итераций, а также методами `Encrypt` и `Decrypt`:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Security.Cryptography;
using System.Security.Principal;
using System.Text;
using System.Xml.Linq;
using static System.Convert;

namespace Packt.Shared
```

```
{
  public static class Protector
  {
    // размер соли должен составлять не менее восьми байт,
    // мы будем использовать 16 байт
    private static readonly byte[] salt =
      Encoding.Unicode.GetBytes("7BANANAS");

    // количество итераций должно быть не меньше 1000,
    // мы будем использовать 2000 итераций
    private static readonly int iterations = 2000;

    public static string Encrypt(
      string plainText, string password)
    {
      byte[] encryptedBytes;
      byte[] plainBytes = Encoding.Unicode
        .GetBytes(plainText);

      var aes = Aes.Create(); // генерирующий метод абстрактных классов

      var pbkdf2 = new Rfc2898DeriveBytes(
        password, salt, iterations);

      aes.Key = pbkdf2.GetBytes(32); // установить 256-битный ключ
      aes.IV = pbkdf2.GetBytes(16); // установить 128-битный вектор
      инициализации
      using (var ms = new MemoryStream())
      {
        using (var cs = new CryptoStream(
          ms, aes.CreateEncryptor(),
          CryptoStreamMode.Write))
        {
          cs.Write(plainBytes, 0, plainBytes.Length);
        }
        encryptedBytes = ms.ToArray();
      }
      return Convert.ToBase64String(encryptedBytes);
    }

    public static string Decrypt(
      string cryptoText, string password)
    {
      byte[] plainBytes;
      byte[] cryptoBytes = Convert
        .FromBase64String(cryptoText);

      var aes = Aes.Create();

      var pbkdf2 = new Rfc2898DeriveBytes(
        password, salt, iterations);
```

```
      aes.Key = pbkdf2.GetBytes(32);
      aes.IV = pbkdf2.GetBytes(16);

      using (var ms = new MemoryStream())
      {
        using (var cs = new CryptoStream(
          ms, aes.CreateDecryptor(),
          CryptoStreamMode.Write))
        {
          cs.Write(cryptoBytes, 0, cryptoBytes.Length);
        }
        plainBytes = ms.ToArray();
      }
      return Encoding.Unicode.GetString(plainBytes);
    }
  }
}
```

Обратите внимание на следующие моменты:

- в примере удвоены рекомендованные размер соли и количество итераций;
- хотя размер соли и количество итераций могут быть жестко закодированными, пароль *должен* передаваться во время выполнения при вызове методов `Encrypt` и `Decrypt`;
- в примере используется временный тип `MemoryStream` для хранения результатов шифрования и дешифрования, а затем вызывается метод `ToArray` для преобразования потока в массив байтов;
- в примере зашифрованные массивы байтов переводятся в и из кодировки Base64, чтобы упростить их чтение.

Никогда не кодируйте жестко пароль в исходном коде, поскольку даже после компиляции пароль в сборке можно прочитать с помощью дизассемблера.

13. В проекте `EncryptionApp` откройте файл `Program.cs`, а затем импортируйте пространство имен для класса `Protector`, пространство имен для класса `CryptographicException` и статически импортируйте класс `Console`, как показано ниже:

```
using System.Security.Cryptography; // CryptographicException
using Packt.Shared;                 // Protector
using static System.Console;
```

14. В метод `Main` добавьте указанные ниже операторы, позволяющие запросить у пользователя сообщение и пароль, а затем выполняющие шифрование и дешифрование:

```
Write("Enter a message that you want to encrypt: ");
string message = ReadLine();

Write("Enter a password: ");
string password = ReadLine();

string cryptoText = Protector.Encrypt(message, password);

WriteLine($"Encrypted text: {cryptoText}");

Write("Enter the password: ");
string password2 = ReadLine();

try
{
  string clearText = Protector.Decrypt(cryptoText, password2);
  WriteLine($"Decrypted text: {clearText}");
}
catch (CryptographicException ex)
{
  WriteLine("{0}\nMore details: {1}",
    arg0: "You entered the wrong password!",
    arg1: ex.Message);
}
catch (Exception ex)
{
  WriteLine("Non-cryptographic exception: {0}, {1}",
    arg0: ex.GetType().Name,
    arg1: ex.Message);
}
```

15. Запустите консольное приложение, введите сообщение и пароль и проанализируйте результат:

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
Encrypted text: pV5qPDf1CCZmGzUMH2gapFSkn573lg7tMj5ajice3cQ=
Enter the password: secret
Decrypted text: Hello Bob
```

16. Перезапустите приложение и вновь введите сообщение и пароль, только на этот раз указав в пароле после шифрования намеренную ошибку. Проанализируйте результат:

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
Encrypted text: pV5qPDf1CCZmGzUMH2gapFSkn573lg7tMj5ajice3cQ=
Enter the password: 123456
You entered the wrong password!
More details: Padding is invalid and cannot be removed.
```

Хеширование данных

На платформе .NET доступно несколько алгоритмов хеширования. Одни не требуют использования каких-либо ключей, вторым необходимы симметричные ключи, а третьим — асимметричные.

При выборе алгоритма хеширования следует учитывать два важных фактора.

- *Устойчивость к коллизиям* — как часто два разных ввода могут иметь один и тот же хеш?
- *Устойчивость к нахождению прообраза* — в отношении хеша, насколько сложно обнаружить другой ввод, который имеет идентичный хеш?

В табл. 10.1 представлены некоторые универсальные алгоритмы хеширования.

Таблица 10.1

Алгоритм	Размер хеша	Описание
MD5	16 байт	Используется чаще всего, поскольку быстр в работе, но не устойчив к коллизиям
SHA1	20 байт	Применение алгоритма SHA1 в Интернете не рекомендуется с 2011 года
SHA256 SHA384 SHA512	32 байта 48 байт 64 байта	Алгоритмы из семейства «Безопасный алгоритм хеширования второго поколения (SHA2, Secure Hashing Algorithm)» с разными размерами хешей

Старайтесь не использовать алгоритмы MD5 и SHA1, поскольку у них выявлены существенные недостатки. Выбирайте алгоритм с большим размером хеша, чтобы уменьшить вероятность повторения хешей. Первая известная коллизия алгоритма MD5 произошла в 2010 году, а первая известная коллизия алгоритма SHA1 — в 2017-м. Более подробно можно прочитать на сайте https://arstechnica.co.uk/information-technology/2017/02/at-deaths-door-for-years-widely-used-sha1-function-is-now-dead/.

Хеширование с помощью алгоритма SHA256

Добавим класс пользователей, хранящихся в памяти, файле или в базе данных. Мы будем использовать словарь для хранения нескольких пользователей в памяти.

1. В проекте библиотеки классов `CryptographyLib` добавьте файл класса `User.cs`, как показано ниже:

```
namespace Packt.Shared
{
```

```
  public class User
  {
    public string Name { get; set; }
    public string Salt { get; set; }
    public string SaltedHashedPassword { get; set; }
  }
}
```

2. Добавьте в класс `Protector` код, показанный ниже. Мы применяем словарь для хранения в памяти информации о нескольких пользователях. Применяются два метода: для регистрации нового пользователя и для проверки введенного пароля при последующей авторизации:

```
private static Dictionary<string, User> Users =
  new Dictionary<string, User>();

public static User Register(
  string username, string password)
{
  // генерация случайной соли
  var rng = RandomNumberGenerator.Create();
  var saltBytes = new byte[16];
  rng.GetBytes(saltBytes);
  var saltText = Convert.ToBase64String(saltBytes);

  // генерация соленого и хешированного пароля
  var saltedhashedPassword = SaltAndHashPassword(
    password, saltText);

  var user = new User
  {
    Name = username, Salt = saltText,
    SaltedHashedPassword = saltedhashedPassword
  };
  Users.Add(user.Name, user);
  return user;
}

public static bool CheckPassword(
  string username, string password)
{
  if (!Users.ContainsKey(username))
  {
    return false;
  }
  var user = Users[username];

  // повторная генерация соленого и хешированного пароля
  var saltedhashedPassword = SaltAndHashPassword(
    password, user.Salt);
```

```
  return (saltedhashedPassword == user.SaltedHashedPassword);
}

private static string SaltAndHashPassword(
  string password, string salt)
{
  var sha = SHA256.Create();
  var saltedPassword = password + salt;
  return Convert.ToBase64String(
    sha.ComputeHash(Encoding.Unicode.GetBytes(saltedPassword)));
}
```

3. Создайте проект консольного приложения `HashingApp`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.
4. Добавьте в код ссылку на сборку `CryptographyLib`, как делали это ранее, а затем импортируйте пространство имен `Packt.Shared`.
5. В метод `Main` добавьте следующие операторы для регистрации пользователя и запроса регистрации другого пользователя, а также запроса авторизации под одним из логинов и проверки пароля:

```
WriteLine("Registering Alice with Pa$$w0rd.");
var alice = Protector.Register("Alice", "Pa$$w0rd");

WriteLine($"Name: {alice.Name}");
WriteLine($"Salt: {alice.Salt}");
WriteLine("Password (salted and hashed): {0}",
  arg0: alice.SaltedHashedPassword);
WriteLine();

Write("Enter a new user to register: ");
string username = ReadLine();
Write($"Enter a password for {username}: ");
string password = ReadLine();

var user = Protector.Register(username, password);

WriteLine($"Name: {user.Name}");
WriteLine($"Salt: {user.Salt}");
WriteLine("Password (salted and hashed): {0}",
  arg0: user.SaltedHashedPassword);
WriteLine();

bool correctPassword = false;
while (!correctPassword)
{
  Write("Enter a username to log in: ");
  string loginUsername = ReadLine();
  Write("Enter a password to log in: ");
  string loginPassword = ReadLine();
```

```
    correctPassword = Protector.CheckPassword(
      loginUsername, loginPassword);

    if (correctPassword)
    {
      WriteLine($"Correct! {loginUsername} has been logged in.");
    }
    else
    {
      WriteLine("Invalid username or password. Try again.");
    }
  }
```

При использовании нескольких проектов не забудьте применять панель TERMINAL (Терминал) для правильного консольного приложения, прежде чем выполнять команды `dotnet build` и `dotnet run`.

6. Запустите консольное приложение, зарегистрируйте нового пользователя с тем же паролем, что и у `Alice` (Алиса), и проанализируйте результат:

```
Registering Alice with Pa$$w0rd.
Name: Alice
Salt: I1I1dzIjkd7EYDf/6jaf4w==
Password (salted and hashed): pIoadjE4W/XaRFkqS3br3UuAuPv/3LVQ8kzj6mvcz+s=
Enter a new user to register: Bob
Enter a password for Bob: Pa$$w0rd
Name: Bob
Salt: 1X7ym/UjxTiuEWBC/vIHpw==
Password (salted and hashed): DoBFtDhKeN0aaaLVdErtrZ3mpZSvpWDQ9TXDosTq0sQ=
Enter a username to log in: Alice
Enter a password to log in: secret
Invalid username or password. Try again.
Enter a username to log in: Bob
Enter a password to log in: secret
Invalid username or password. Try again.
Enter a username to log in: Bob
Enter a password to log in: Pa$$w0rd
Correct! Bob has been logged in.
```

Даже если оба пользователя зарегистрируются с одним и тем же паролем, соль будет генерироваться случайным образом, поэтому их соленые и хешированные пароли будут различаться.

Подписывание данных

В качестве доказательства, что полученные данные на самом деле присланы доверенным отправителем, используются цифровые подписи. По сути, вы подписываете не сами данные, а только хеш этих данных.

Для этих целей мы воспользуемся сочетанием алгоритмов RSA и SHA256.

Мы могли бы применить алгоритм DSA как для хеширования, так и для подписывания. Генерация ключей и цифровая подпись с помощью DSA более быстрая, но RSA позволяет ускорить проверку подписи. Поскольку подпись генерируется один раз, но проверяется многократно, лучше проводить быстрее проверку, а не генерацию.

Алгоритм RSA основан на факторизации больших целых чисел по сравнению с алгоритмом DSA, который базируется на вычислении дискретного логарифма. Более подробно можно прочитать на сайте http://mathworld.wolfram.com/RSAEncryption.html.

Подписывание с помощью алгоритмов SHA256 и RSA

Рассмотрим подписывание данных и проверку подписи с помощью алгоритма шифрования с открытым ключом.

1. В проекте библиотеки классов `CryptographyLib` добавьте операторы в класс `Protector`, чтобы объявить поле для открытого ключа, два метода расширения для преобразования экземпляра RSA в и из XML и два метода для генерации и проверки подписи:

```
public static string PublicKey;

public static string ToXmlStringExt(
  this RSA, bool includePrivateParameters)
{
  var p = rsa.ExportParameters(includePrivateParameters);

  XElement xml;

  if (includePrivateParameters)
  {
    xml = new XElement("RSAKeyValue",
      new XElement("Modulus", ToBase64String(p.Modulus)),
      new XElement("Exponent", ToBase64String(p.Exponent)),
      new XElement("P", ToBase64String(p.P)),
      new XElement("Q", ToBase64String(p.Q)),
      new XElement("DP", ToBase64String(p.DP)),
      new XElement("DQ", ToBase64String(p.DQ)),
      new XElement("InverseQ", ToBase64String(p.InverseQ))
    );
  }
  else
  {
```

```
      xml = new XElement("RSAKeyValue",
        new XElement("Modulus", ToBase64String(p.Modulus)),
        new XElement("Exponent",
          ToBase64String(p.Exponent)));
    }
    return xml?.ToString();
  }

  public static void FromXmlStringExt(
    this RSA rsa, string parametersAsXml)
  {
    var xml = XDocument.Parse(parametersAsXml);
      var root = xml.Element("RSAKeyValue");

      var p = new RSAParameters
    {
      Modulus = FromBase64String(
        root.Element("Modulus").Value),
      Exponent = FromBase64String(
        root.Element("Exponent").Value)
    };

    if (root.Element("P") != null)
    {
      p.P = FromBase64String(root.Element("P").Value);
      p.Q = FromBase64String(root.Element("Q").Value);
      p.DP = FromBase64String(root.Element("DP").Value);
      p.DQ = FromBase64String(root.Element("DQ").Value);
      p.InverseQ = FromBase64String(
        root.Element("InverseQ").Value);
    }
    rsa.ImportParameters(p);
  }

  public static string GenerateSignature(string data)
  {
    byte[] dataBytes = Encoding.Unicode.GetBytes(data);
    var sha = SHA256.Create();
    var hashedData = sha.ComputeHash(dataBytes);
    var rsa = RSA.Create();

    PublicKey = rsa.ToXmlStringExt(false); // исключая приватный ключ

    return ToBase64String(rsa.SignHash(hashedData,
      HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1));
  }

  public static bool ValidateSignature(
    string data, string signature)
  {
    byte[] dataBytes = Encoding.Unicode.GetBytes(data);
    var sha = SHA256.Create();
```

```
  var hashedData = sha.ComputeHash(dataBytes);
  byte[] signatureBytes = FromBase64String(signature);
  var rsa = RSA.Create();
  rsa.FromXmlStringExt(PublicKey);
  return rsa.VerifyHash(hashedData, signatureBytes,
    HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
}
```

Обратите внимание на следующие моменты.

- Я использовал два полезных метода класса `RSA`: `ToXmlString` и `FromXmlString`. Они сериализуют/десериализуют структуру `RSAParameters`, которая содержит открытые и закрытые ключи. Однако реализация этих методов в операционной системе macOS вызывает исключение `PlatformNotSupportedException`. Мне пришлось заново реализовать их как методы расширения с именами `ToXmlStringExt` и `FromXmlStringExt` с помощью таких типов LINQ to XML, как, например, `XDocument`. О них вы узнаете в главе 12.
- Перед использованием кода проверки подписи должен быть доступен только открытый ключ из пары открытого/закрытого ключей, чтобы при вызове метода `ToXmlString` мы могли передать значение `false`. Закрытый ключ требуется для подписи данных и должен храниться в секрете, поскольку любой, кто имеет доступ к этому ключу, может подписывать данные от вашего имени!
- Алгоритм хеширования, применяемый для генерации хеша из данных с помощью вызова метода `SignHash`, должен соответствовать алгоритму хеширования, используемому при вызове метода `VerifyHash`. В вышеприведенном примере мы задействовали алгоритм `SHA256`.

Теперь мы можем протестировать подписывание каких-нибудь данных и проверку подписи.

2. Создайте проект консольного приложения `SigningApp`, добавьте его в рабочую область и выберите проект как активный для OmniSharp.
3. Добавьте ссылку на библиотеку `CryptographyLib`, в файле `Program.cs` импортируйте соответствующие пространства имен. В метод `Main` добавьте операторы, чтобы предложить пользователю ввести некий текст, поставить электронную подпись, затем проверьте его подпись. Измените текст так, чтобы намеренно вызвать ошибку, и снова проверьте подпись:

```
Write("Enter some text to sign: ");
string data = ReadLine();

var signature = Protector.GenerateSignature(data);

WriteLine($"Signature: {signature}");
WriteLine("Public key used to check signature:");
WriteLine(Protector.PublicKey);
```

```
if (Protector.ValidateSignature(data, signature))
{
  WriteLine("Correct! Signature is valid.");
}
else
{
  WriteLine("Invalid signature.");
}

// создаем поддельную подпись, заменив первый символ на X
var fakeSignature = signature.Replace(signature[0], 'X');

if (Protector.ValidateSignature(data, fakeSignature))
{
  WriteLine("Correct! Signature is valid.");
}
else
{
  WriteLine($"Invalid signature: {fakeSignature}");
}
```

4. Запустите консольное приложение и введите любой текст, как показано в следующем выводе (отредактированном по длине):

```
Enter some text to sign: The cat sat on the mat.
Signature: BXSTdM...4Wrg==
Public key used to check signature:
<RSAKeyValue>
  <Modulus>nHtwl3...mw3w==</Modulus>
  <Exponent>AQAB</Exponent>
</RSAKeyValue>
Correct! Signature is valid.
Invalid signature: XXSTdM...4Wrg==
```

Генерация случайных чисел

Иногда необходимо сгенерировать случайные числа, возможно при создании игры, симулирующей бросок игральных костей, либо для дальнейшего использования с криптографией в шифровании или постановке электронной подписи. В .NET существует несколько классов, генерирующих случайные числа.

Генерация случайных чисел для игр

В сценариях, не требующих истинно случайных чисел, например в играх, вы можете создать экземпляр класса `Random`:

```
var r = new Random();
```

В конструкторе класса Random содержится параметр для указания начального значения, используемого для инициализации генератора псевдослучайных чисел, как показано ниже:

```
var r = new Random(Seed: 12345);
```

Общие начальные значения работают как секретные ключи, поэтому если вы воспользуетесь одинаковым алгоритмом генерации случайных чисел с одинаковым начальным значением в двух приложениях, то они могут генерировать одинаковые последовательности «случайных» чисел. Иногда это необходимо, например при синхронизации GPS-приемника со спутником. Но обычно следует держать начальное значение в секрете.

Как вы помните из главы 2, имена параметров необходимо задавать с помощью так называемого верблюжьего регистра. Разработчик, определивший конструктор класса Random, нарушил это соглашение! Имя параметра следовало задать как seed, но не Seed.

Создав объект Random, вы можете вызывать его методы для генерации случайных чисел, как показано в следующих примерах кода:

```
int dieRoll = r.Next(minValue: 1, maxValue: 7); //возвращает число от 1 до 6

double randomReal = r.NextDouble(); // возвращает число от 0.0 до 1.0

var arrayOfBytes = new byte[256];
r.NextBytes(arrayOfBytes); // 256 случайных байт в массиве
```

Метод Next принимает два параметра: minValue и maxValue. Параметр maxValue — не максимальное значение, возвращаемое методом! Это *эксклюзивная* верхняя граница, то есть число на единицу больше максимального значения.

Генерация случайных чисел для криптографии

Класс Random генерирует псевдослучайные числа. Для криптографии его недостаточно! Если случайные числа — не случайные, то они повторяемы, а если повторяемы, то взломщик может сломать вашу защиту.

Для истинно случайных чисел необходимо использовать тип, наследующий от RandomNumberGenerator, например RNGCryptoServiceProvider.

Создадим метод для генерации истинно случайного байтового массива, который можно использовать в таких алгоритмах, как шифрование, для получения ключей и значений вектора инициализации IV.

1. В проекте библиотеки классов CryptographyLib добавьте в класс Protector следующие операторы, чтобы определить метод, генерирующий случайный

ключ или вектор инициализации IV для использования в шифровании, как показано ниже:

```
public static byte[] GetRandomKeyOrIV(int size)
{
  var r = RandomNumberGenerator.Create();
  var data = new byte[size];
  r.GetNonZeroBytes(data);
  // в переменной data сейчас массив
  // криптографически сильных случайных байтов
  return data;
}
```

Теперь мы можем протестировать случайные байты, сгенерированные для истинно случайного ключа шифрования или вектора инициализации.

2. Создайте проект консольного приложения `RandomizingApp`, добавьте его в рабочую область и выберите проект как активный для OmniSharp.
3. Добавьте ссылку на библиотеку `CryptographyLib`, импортируйте соответствующие пространства имен, а затем в метод `Main` добавьте операторы, чтобы предложить пользователю ввести размер байтового массива. Далее сгенерируйте случайные байтовые значения и запишите их в консоль:

```
Write("How big do you want the key (in bytes): ");
string size = ReadLine();

byte[] key = Protector.GetRandomKeyOrIV(int.Parse(size));
WriteLine($"Key as byte array:");
for (int b = 0; b < key.Length; b++)
{
  Write($"{key[b]:x2} ");
  if (((b + 1) % 16) == 0) WriteLine();
}
WriteLine();
```

4. Запустите консольное приложение, введите размерность ключа, например `256`, и проанализируйте случайно сгенерированный ключ:

```
How big do you want the key (in bytes): 256
Key as byte array:
f1 57 3f 44 80 e7 93 dc 8e 55 04 6c 76 6f 51 b9
e8 84 59 e5 8d eb 08 d 5 e6 59 65 20 b1 56 fa 68
...
```

Криптография: что нового

Преимущество использования .NET Core 3.0 заключается в том, что для лучшей производительности алгоритмы хеширования, HMAC, генерации случайных чисел, генерации и обработки асимметричной подписи и шифрования RSA были пере-

писаны под применение `Span<T>`. Например, класс `Rfc2898DeriveBytes` работает приблизительно на 15 % быстрее.

Некоторые полезные в сложных сценариях обновления были внесены в API шифрования, включая:

- подписание и проверку сообщений CMS/PKCS #7;
- методы `X509Certificate.GetCertHash` и `X509Certificate.GetCertHashString` для получения значений отпечатка сертификата могут использовать алгоритмы, отличные от SHA-1;
- класс `CryptographicOperations` с полезными методами, такими как `ZeroMemory`, для безопасной очистки памяти;
- класс `RandomNumberGenerator`, который содержит метод `Fill`, заполняющий интервал случайными значениями и не требующий ручного управления ресурсом `IDisposable`;
- API для чтения, проверки и создания значений типа `TimestampToken` согласно стандарту RFC 3161;
- поддержку протокола Диффи — Хеллмана на эллиптических кривых с использованием классов `ECDiffieHellman`;
- поддержку для алгоритмов RSA-OAEP-SHA2 и RSA-PSS на платформах Linux.

Аутентификация и авторизация пользователей

Аутентификация — это процесс проверки подлинности личности пользователя с использованием проверки его учетных данных через некоторый удостоверяющий центр. Учетные данные могут включать в себя имя пользователя и пароль, или отпечаток пальца, или сканирование лица.

После проверки подлинности удостоверяющий центр может выдавать утверждения о пользователе, например, каков его адрес электронной почты и к каким группам или ролям он принадлежит.

Авторизация — это процесс определения принадлежности к группе или роли перед тем, как выдать доступ к таким ресурсам, как функции приложения или его данные. Хотя авторизация может основываться на индивидуальной идентификации, рекомендуется выполнять авторизацию на основе принадлежности к роли или группе (что можно указать в утверждениях), даже если в группе или роли имеется только один пользователь, потому что это позволяет в дальнейшем менять информацию о принадлежности пользователя, не прибегая к повторному назначению индивидуальных прав доступа.

Например, вместо того, чтобы назначать права доступа для нанесения ядерного удара Дональду Трампу (пользователю), вы назначаете права доступа для запуска ядерного удара президенту Соединенных Штатов (роль), а затем добавляете Дональда Трампа в качестве участника этой роли.

Существует несколько механизмов аутентификации и авторизации. Все они реализуют пару интерфейсов в пространстве имен `System.Security.Principal`: `IIdentity` и `IPrincipal`.

Интерфейс `IIdentity` представляет пользователя, поэтому имеет свойства `Name` и `IsAuthenticated`, чтобы указать, является ли пользователь анонимным или успешно прошедшим аутентификацию по своим учетным данным.

Наиболее распространенным классом, реализующим этот интерфейс, является `GenericIdentity`, который наследуется от класса `ClaimsIdentity`, как показано на следующей схеме (рис. 10.1).

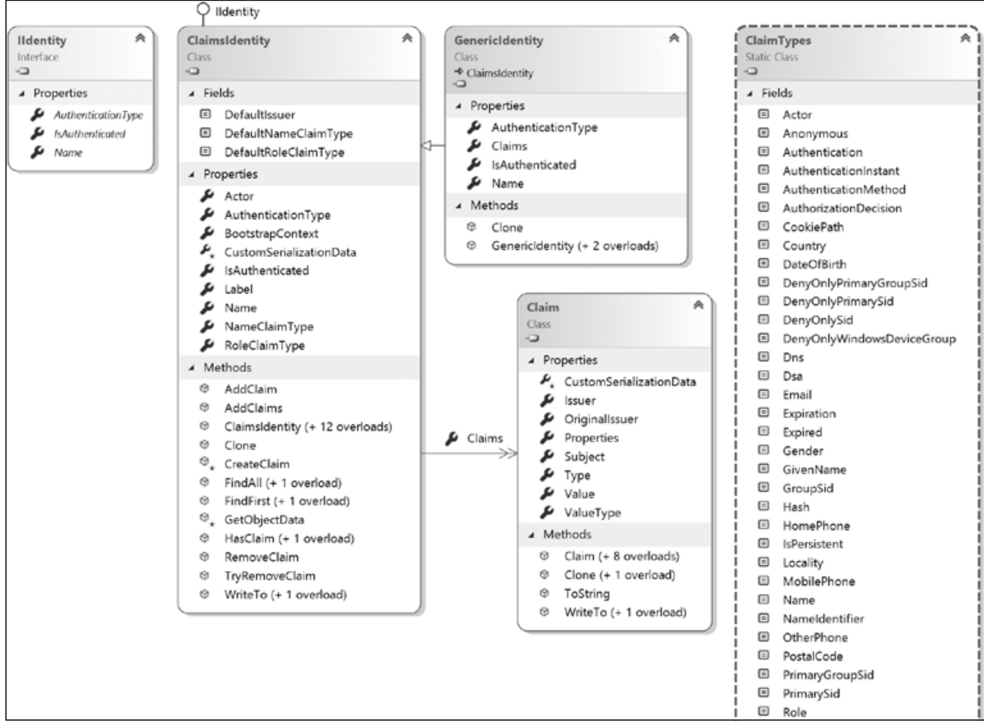


Рис. 10.1. Типы для работы с аутентификацией

Каждый класс `ClaimsIdentity` содержит свойство `Claims`, представленное на рис. 10.1 в виде двойной стрелки между классами `ClaimsIdentity` и `Claim`.

Объекты Claim имеют свойство Type, указывающее, к чему относится утверждение: к имени пользователя, его принадлежности к роли или группе, дате рождения и т. д.

Интерфейс IPrincipal служит для связи пользователя с ролями и группами, членом которых он является, поэтому IPrincipal можно использовать для целей авторизации. Текущий поток, выполняющий код приложения, содержит свойство CurrentPrincipal, которому может быть присвоен в качестве значения любой объект, реализующий интерфейс IPrincipal. В дальнейшем система будет обращаться к CurrentPrincipal для получения разрешения на выполнение защищенного действия.

Наиболее распространенным классом, реализующим этот интерфейс, является GenericPrincipal, который наследуется от ClaimsPrincipal, как показано на следующей схеме (рис. 10.2).

Рис. 10.2. Типы для работы с аутентификацией

Реализация аутентификации и авторизации

Рассмотрим использование аутентификации и авторизации на примере.

1. В проекте библиотеки классов CryptographyLib добавьте следующее свойство в класс User для хранения массива ролей:

```
public string[] Roles { get; set; }
```

2. Измените метод `Register` в классе `Protector` так, чтобы разрешить передачу массива ролей в качестве необязательного параметра, как показано в коде ниже (выделено полужирным шрифтом):

```
public static User Register(
  string username, string password,
  string[] roles = null)
```

3. Измените метод `Register` в классе `Protector` так, чтобы установить массив ролей в объекте `User`:

```
var user = new User
{
  Name = username, Salt = saltText,
  SaltedHashedPassword = saltedhashedPassword,
  Roles = roles
};
```

4. В проекте библиотеки классов `CryptographyLib` добавьте операторы в класс `Protector`, чтобы определить метод `LogIn` для входа в систему пользователя, и используйте `GenericIdentity` и `GenericPrincipal` для установки значения свойства `CurrentPrincipal` текущего потока:

```
public static void LogIn(string username, string password)
{
  if (CheckPassword(username, password))
  {
    var identity = new GenericIdentity(
      username, "PacktAuth");
    var principal = new GenericPrincipal(
      identity, Users[username].Roles);
    System.Threading.Thread.CurrentPrincipal = principal;
  }
}
```

5. Создайте проект консольного приложения `SecureApp`, добавьте его в рабочую область и выберите проект как активный для OmniSharp.
6. Добавьте ссылку на библиотеку `CryptographyLib`, а затем в файле `Program.cs` импортируйте следующие пространства имен:

```
using static System.Console;
using Packt.Shared;
using System.Threading;
using System.Security;
using System.Security.Permissions;
using System.Security.Principal;
using System.Security.Claims;
```

7. В метод `Main` добавьте операторы для регистрации трех пользователей по имени `Alice` (Алиса), `Bob` (Боб) и `Eve` (Ева), имеющих различные роли. Затем

попросите пользователя войти в систему и выведите сведения о них, как показано ниже:

```
Protector.Register("Alice", "Pa$$w0rd",
  new[] { "Admins" });

Protector.Register("Bob", "Pa$$w0rd",
  new[] { "Sales", "TeamLeads" });

Protector.Register("Eve", "Pa$$w0rd");

Write($"Enter your user name: ");
string username = ReadLine();

Write($"Enter your password: ");
string password = ReadLine();

Protector.LogIn(username, password);

if (Thread.CurrentPrincipal == null)
{
  WriteLine("Log in failed.");
  return;
}

var p = Thread.CurrentPrincipal;

WriteLine(
  $"IsAuthenticated: {p.Identity.IsAuthenticated}");
WriteLine(
  $"AuthenticationType: {p.Identity.AuthenticationType}");
WriteLine($"Name: {p.Identity.Name}");
WriteLine($"IsInRole(\"Admins\"): {p.IsInRole("Admins")}");
WriteLine($"IsInRole(\"Sales\"): {p.IsInRole("Sales")}");

if (p is ClaimsPrincipal)
{
  WriteLine(
    $"{p.Identity.Name} has the following claims:");

  foreach (Claim in (p as ClaimsPrincipal).Claims)
  {
    WriteLine($"{claim.Type}: {claim.Value}");
  }
}
```

8. Запустите консольное приложение, войдите в систему под именем `Alice` (Алиса) с паролем `Pa$$word` и проанализируйте результат:

```
Enter your user name: Alice
Enter your password: Pa$$w0rd
```

```
IsAuthenticated: True
AuthenticationType: PacktAuth
Name: Alice
IsInRole("Admins"): True
IsInRole("Sales"): False
Alice has the following claims:
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: Alice
http://schemas.microsoft.com/ws/2008/06/identity/claims/role: Admins
```

9. Запустите консольное приложение, войдите в систему под именем `Alice` (Алиса) с паролем `secret` и проанализируйте результат:

```
Enter your user name: Alice
Enter your password: secret
Log in failed.
```

10. Запустите консольное приложение, войдите в систему под именем `Bob` (Боб) с паролем `Pa$$word` и проанализируйте результат:

```
Enter your user name: Bob
Enter your password: Pa$$w0rd
IsAuthenticated: True
AuthenticationType: PacktAuth
Name: Bob
IsInRole("Admins"): False
IsInRole("Sales"): True
Bob has the following claims:
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: Bob
http://schemas.microsoft.com/ws/2008/06/identity/claims/role: Sales
http://schemas.microsoft.com/ws/2008/06/identity/claims/role: TeamLeads
```

Защита приложения

Рассмотрим применение авторизации для предотвращения доступа некоторых пользователей к некоторым функциям приложения.

1. Добавьте метод в класс `Program`, защищенный проверкой разрешения внутри метода, и сгенерируйте соответствующие исключения, если пользователь анонимный или не член роли `Admins`:

```
static void SecureFeature()
{
  if (Thread.CurrentPrincipal == null)
  {
    throw new SecurityException(
      "A user must be logged in to access this feature.");
  }

  if (!Thread.CurrentPrincipal.IsInRole("Admins"))
  {
    throw new SecurityException(
```

```
        "User must be a member of Admins to access this feature.");
    }

    WriteLine("You have access to this secure feature.");
  }
```

2. Добавьте операторы в конец метода `Main` для вызова метода `SecureFeature` в операторе `try`:

```
try
{
  SecureFeature();
}
catch (System.Exception ex)
{
  WriteLine($"{ex.GetType()}: {ex.Message}");
}
```

3. Запустите консольное приложение, войдите в систему под именем `Alice` (Алиса) с паролем `Pa$$word` и проанализируйте результат:

```
You have access to this secure feature.
```

4. Запустите консольное приложение, войдите в систему под именем `Bob` (Боб) с паролем `Pa$$word` и проанализируйте результат:

```
System.Security.SecurityException: User must be a member of Admins to
access this feature.
```

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 10.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какой из алгоритмов шифрования, доступных на платформе .NET, лучше всего подойдет для симметричного шифрования?
2. Какой из алгоритмов шифрования, доступных на платформе .NET, лучше всего подойдет для асимметричного шифрования?
3. Что такое радужная атака?
4. При использовании алгоритмов шифрования лучше применять блоки большого или малого размера?

5. Что такое хеширование данных?
6. Что такое подписывание данных?
7. В чем разница между симметричным и асимметричным шифрованием?
8. Что такое RSA?
9. Почему пароли должны быть засолены перед сохранением?
10. SHA1 — это алгоритм хеширования, разработанный Национальным агентством безопасности США. Почему его не следует использовать?

Упражнение 10.2. Защита данных с помощью шифрования и хеширования

Создайте консольное приложение `Exercise02`, предназначенное для защиты XML-файла:

```
<?xml version="1.0" encoding="utf-8" ?>
<customers>
  <customer>
    <name>Bob Smith</name>
    <creditcard>1234-5678-9012-3456</creditcard>
    <password>Pa$$w0rd</password>
  </customer>
  ...
</customers>
```

Обратите внимание: номер банковской карты и пароль клиента хранятся в открытом виде. Номер карты должен быть зашифрован, чтобы ее можно было расшифровать и использовать позже, а пароль следует засолить и хешировать.

Упражнение 10.3. Дешифрование данных

Создайте консольное приложение `Exercise03`, открывающее XML-файл, защитой которого вы занимались в предыдущем упражнении, и расшифровывающее номер банковской карты.

Упражнение 10.4. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- ключевые концепции безопасности: https://docs.microsoft.com/ru-ru/dotnet/standard/security/key-security-concepts;

- шифрование данных: https://docs.microsoft.com/ru-ru/dotnet/standard/security/encrypting-data;
- криптографические подписи: https://docs.microsoft.com/ru-ru/dotnet/standard/security/cryptographic-signatures.

Резюме

Вы узнали, как зашифровать и дешифровать данные с помощью симметричного шифрования, генерировать соленый хеш, подписывать данные и проверять цифровые подписи, генерировать истинно случайные числа и использовать аутентификацию и авторизацию для защиты ваших приложений.

Далее вы узнаете, как работать с базами данных с помощью Entity Framework Core.

11 Работа с базами данных с помощью Entity Framework Core

Текущая глава посвящена чтению и записи в такие базы данных, как Microsoft SQL Server, SQLite и Azure Cosmos DB, с помощью технологии объектно-реляционного отображения данных *Entity Framework Core (EF Core)*.

В этой главе:

- современные базы данных;
- настройка EF Core;
- определение моделей EF Core;
- запрос данных из модели EF Core;
- схемы загрузки данных при использовании EF Core;
- управление данными с помощью EF Core.

Современные базы данных

Два наиболее распространенных места для хранения данных — это *реляционная система управления базами данных* (РСУБД; Relational Database Management System, RDBMS), такая как Microsoft SQL Server, PostgreSQL, MySQL и SQLite, или хранилище данных *NoSQL*, такое как Microsoft Azure Cosmos DB, Redis, MongoDB и Apache Cassandra.

В этой главе основное внимание будет уделено таким РСУБД, как SQL Server и SQLite. Если вы хотите узнать больше о базах данных NoSQL, таких как Cosmos DB и MongoDB, и о том, как их использовать с Entity Framework Core, то я рекомендую следующие ссылки:

- вас приветствует Azure Cosmos DB: docs.microsoft.com/ru-ru/azure/cosmos-db/introduction;

- использование базы данных NoSQL в качестве инфраструктуры сохраняемости: docs.microsoft.com/ru-ru/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/nosql-database-persistence-infrastructure;
- поставщики документных баз данных для Entity Framework Core: github.com/BlueshiftSoftware/EntityFrameworkCore.

Введение в Entity Framework

Технология Entity Framework (EF) была впервые выпущена как часть .NET Framework 3.5 с Service Pack 1 в конце 2008 года. С тех пор эта технология значительно развилась, поскольку корпорация Microsoft пристально наблюдала за тем, как программисты используют этот инструмент *объектно-реляционного отображения* (object-relational mapping, ORM) при работе над своими продуктами.

Суть ORM в определении сопоставления между столбцами в таблицах и свойствами в классах. Затем разработчик может взаимодействовать с объектами различных типов привычным для него способом вместо того, чтобы изучать способы сохранения значений в реляционной таблице или другой структуре, предоставляемой хранилищем данных NoSQL.

Entity Framework 6 (EF6) — это версия EF, включенная в последнюю версию .NET Framework. Она полная, стабильная и поддерживает устаревший способ определения модели EDMX (XML-файл), а также сложные модели наследования и некоторые другие расширенные функции.

Версия EF 6.3 и более поздние версии были извлечены из .NET Framework в виде отдельного пакета, поэтому данные версии могут поддерживаться в .NET Core 3.0 и более поздних версиях, включая .NET 5. Это позволяет переносить существующие проекты, такие как веб-приложения и сервисы. Тем не менее EF6 следует считать устаревшей технологией, поскольку она имеет некоторые ограничения при работе с разными платформами и в нее не добавляются новые функции.

Более подробную информацию о технологии Entity Framework 6.3 и поддержке ее со стороны .NET Core 3.0 и более поздних версий можно найти на сайте https://devblogs.microsoft.com/dotnet/announcing-ef-core-3-0-and-ef-6-3-general-availability/.

Чтобы использовать устаревшую версию Entity Framework в проекте .NET Core 3.0 или более поздней версии, необходимо добавить ссылку на пакет в файл проекта, как показано ниже:

```
<PackageReference Include="EntityFramework" Version="6.4.4" />
```

Используйте устаревшую версию EF6 только при необходимости. Данная книга посвящена современной кросс-платформенной разработке, поэтому в оставшейся части этой главы я буду рассматривать только современное ядро Entity Framework Core. Вам не нужно будет ссылаться на устаревший пакет EF6, как показано выше в проектах для этой главы.

Entity Framework Core

По-настоящему кросс-платформенная версия EF Core отличается от устаревшей Entity Framework. Хотя EF Core имеет похожее имя, вы должны знать, чем оно отличается от EF6. Например, помимо традиционных СУБД, EF Core также поддерживает современные облачные, нереляционные хранилища данных без схемы, такие как Microsoft Azure Cosmos DB и MongoDB, иногда со сторонними поставщиками.

EF Core 5.0 работает на платформах, поддерживающих .NET Standard 2.1, то есть .NET Core 3.0 и 3.1, а также .NET 5. EF Core 5.0 не будет работать на платформах .NET Standard 2.0, таких как .NET Framework 4.8.

Более подробную информацию о планах разработки EF Core можно найти на сайте https://docs.microsoft.com/ru-ru/ef/core/what-is-new/ef-core-5.0/plan.

EF Core 5.0 содержит множество обновлений, и я не могу все описать в этой главе. Поэтому я сосредоточусь на основных принципах, которые должны знать все разработчики .NET, и на некоторых интересных новых функциях.

Полный список новых функций в EF Core 5 можно узнать на сайте https://docs.microsoft.com/ru-ru/ef/core/what-is-new/ef-core-5.0/whatsnew.

Использование образца реляционной базы данных

Чтобы научиться управлять реляционной базой данных с помощью .NET Core, было бы полезно иметь под рукой образец средней сложности, но с достойным количеством заготовленных записей. Корпорация Microsoft предлагает несколько образцов БД, большинство из которых слишком сложны для наших целей, поэтому мы воспользуемся базой данных `Northwind`, впервые созданной в начале 1990-х годов.

Ниже приведена схема базы данных `Northwind`, которую вы можете использовать для справки, когда мы будем в дальнейшем писать код и запросы (рис. 11.1).

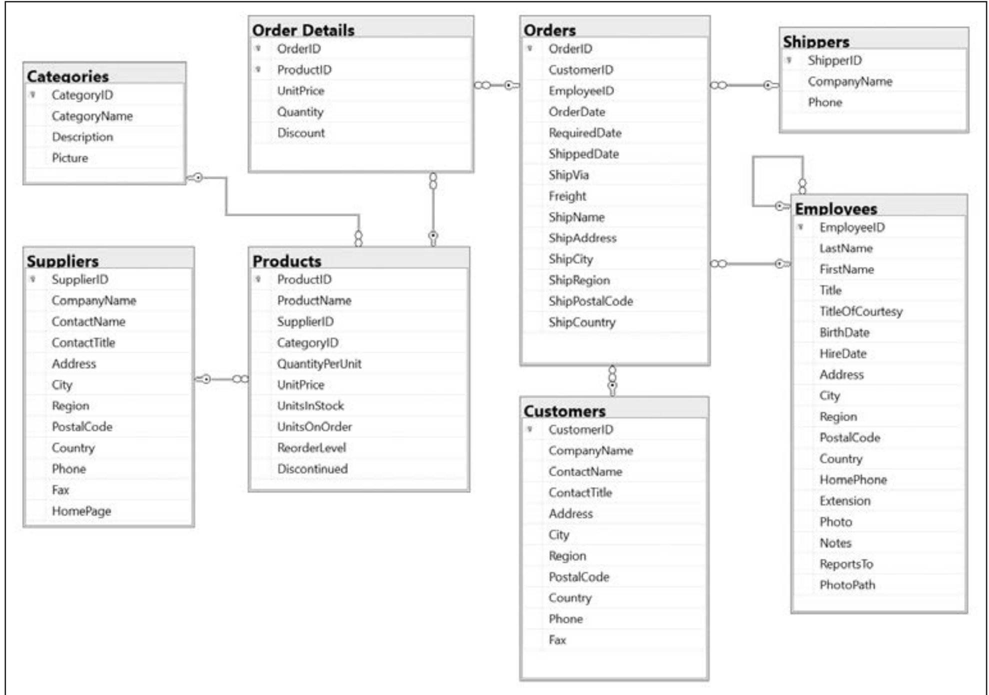


Рис. 11.1. Таблицы и связи базы данных Northwind

Далее в этой и последующих главах мы напишем код для работы с таблицами `Categories` и `Products`. Но сначала обратите внимание на следующие моменты:

- каждая категория имеет уникальный идентификатор, имя, описание и изображение;
- каждый товар имеет уникальный идентификатор, имя, цену за единицу, единицы на складе и другие поля;
- каждый товар связан с категорией, сохраняя уникальный идентификатор категории;
- связи между `Categories` и `Products` «один ко многим», то есть каждая категория может иметь ноль или более товаров.

SQLite — небольшая кросс-платформенная самодостаточная СУБД, доступная в публичном домене. Это самая распространенная база данных для мобильных платформ, таких как iOS (iPhone и iPad) и Android.

Настройка SQLite в macOS

SQLite включена в каталог /usr/bin/ операционной системы macOS в виде консольного приложения `sqlite3`.

Настройка SQLite в Windows

SQLite можно скачать и установить для других операционных систем. Для работы в Windows нам также необходимо добавить имя папки с SQLite в системный путь, чтобы программа была обнаружена при вводе команд в командной строке.

1. Запустите ваш любимый браузер и перейдите по ссылке www.sqlite.org/download.html.
2. Прокрутите страницу вниз до раздела Precompiled Binaries for Windows (Предварительно скомпилированные двоичные файлы для Windows).
3. Выберите файл `sqlite-tools-win32-x86-3330000.zip`. Обратите внимание, что после публикации этой книги файл может содержать более высокий номер версии.
4. Извлеките содержимое ZIP-файла в папку C:\Sqlite\.
5. Перейдите к окну Windows Settings (Настройки Windows).
6. Найдите `environment` и выберите пункт Edit the system environment variables (Изменить системные переменные среды).
7. Нажмите кнопку Environment Variables (Переменные среды).
8. В разделе System variables (Системные переменные) выберите в списке пункт Path (Путь) и нажмите кнопку Edit... (Редактировать).
9. Выберите пункт New (Новый), введите `C:\Sqlite` и нажмите клавишу Enter.
10. Далее трижды нажмите кнопку OK.
11. Закройте окно Windows Settings (Настройки Windows).

Создание образца базы данных Northwind для SQLite

Теперь мы можем создать образец базы данных `Northwind` с помощью SQL-скрипта.

1. Создайте папку Chapter11 с подпапкой WorkingWithEFCore.
2. Если вы ранее не клонировали репозиторий GitHub для этой книги, сделайте это сейчас, используя следующую ссылку: https://github.com/markjprice/cs9dotnet5/.
3. Скачайте скрипт для создания базы данных `Northwind` для SQLite по следующей ссылке в локальном репозитории Git: `/sql-scripts/Northwind.sql` — в папку WorkingWithEFCore.

4. В Visual Studio Code откройте папку WorkingWithEFCore.
5. Перейдите к панели TERMINAL (Терминал) и передайте скрипт в SQLite для создания базы данных Northwind.db, как показано в следующей команде:

```
sqlite3 Northwind.db -init Northwind.sql
```

6. Вам потребуется немного терпения, поскольку на создание всей структуры базы данных у этой команды может уйти некоторое время.

```
-- Loading resources from Northwind.sql
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
sqlite>
```

7. Для выхода из командного режима SQLite нажмите сочетание клавиш Ctrl+C в Windows или Ctrl+D в macOS.

Об операторах SQL, поддерживаемых SQLite, можно прочитать на сайте https://sqlite.org/lang.html.

Управление образцом базы данных Northwind в SQLiteStudio

Для простого управления базами данных SQLite вы можете использовать кросс-платформенный графический менеджер баз данных *SQLiteStudio*.

1. Перейдите по ссылке http://sqlitestudio.pl и скачайте и установите приложение.
2. Запустите *SQLiteStudio*.
3. В меню Database (База данных) выберите пункт Add a database (Добавить базу данных).
4. В диалоговом окне Database (База данных) щелкните на значке папки, чтобы перейти к существующему файлу базы данных на локальном компьютере. В папке WorkingWithEFCore выберите файл Northwind.db и нажмите кнопку OK.
5. Щелкните правой кнопкой мыши на базе данных Northwind и выберите Connect to the database (Подключиться к базе данных). Вы увидите таблицы, созданные с помощью скрипта.
6. Щелкните правой кнопкой мыши на таблице Products и выберите команду Edit the table (Редактировать таблицу).

   В окне редактора таблицы вы можете увидеть структуру таблицы Products, включая имена столбцов, типы данных, ключи и ограничения (рис. 11.2).

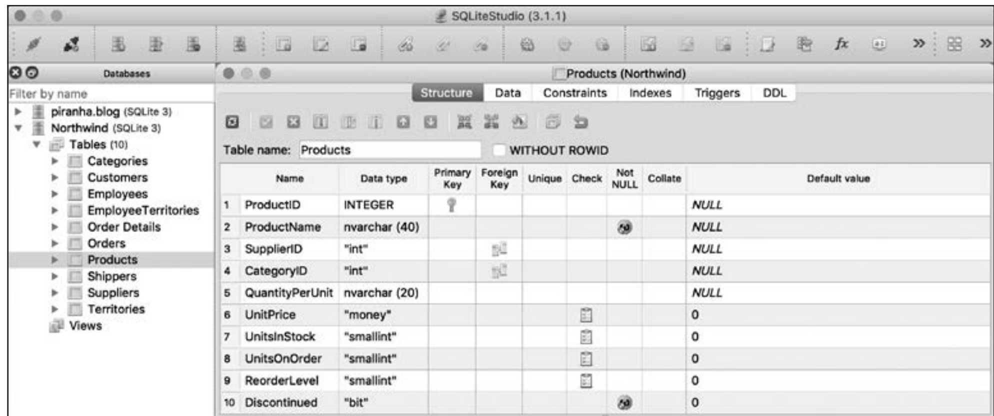


Рис. 11.2. Редактор таблиц в SQLiteStudio, отображающий структуру таблицы Products

7. В окне редактора таблицы перейдите на вкладку Data (Данные). Вы увидите 77 строк с наименованием товаров (рис. 11.3).

Рис. 11.3. Вкладка Data (Данные) со строками в таблице Products (Товары)

Настройка EF Core

Прежде чем углубляться в практические аспекты управления базами данных с помощью Entity Framework Core, немного поговорим о выборе *поставщиков данных Entity Framework Core*.

Выбор поставщика данных Entity Framework Core

Для управления данными в конкретной БД нам нужны классы, которые знают, как эффективно *«общаться»* с базой. Поставщики данных Entity Framework Core — это наборы классов, оптимизированные для работы с конкретным хранилищем данных. Существует даже поставщик для хранения данных в памяти текущего процесса, который полезен для высокопроизводительного модульного тестирования, поскольку ему не требуется общаться с внешней системой.

Они поставляются в пакетах NuGet (табл. 11.1).

Таблица 11.1

Управляемая СУРБД	**NuGet-пакет**
Microsoft SQL Server 2012 или новее	Microsoft.EntityFrameworkCore.SqlServer
SQLite 3.7 или новее	Microsoft.EntityFrameworkCore.SQLite
MySQL	MySQL.Data.EntityFrameworkCore
In-memory (для тестирования элементов)	Microsoft.EntityFrameworkCore.InMemory
Azure Cosmos DB SQL API	Microsoft.EntityFrameworkCore.Cosmos
Oracle DB 11.2	Oracle.EntityFrameworkCore

Devart — это независимый производитель, предлагающий поставщики Entity Framework Core для широкого спектра баз данных. Более подробную информацию можно найти, перейдя по ссылке https://www.devart.com/dotconnect/entityframework.html.

Настройка инструмента dotnet-ef

.NET содержит инструмент командной строки под названием dotnet. Его можно расширить возможностями, полезными для работы с EF Core. Во время разработки с помощью данного инструмента можно выполнять задачи, такие как создание и применение миграции из старой модели в новую, а также создание кода для модели из существующей базы данных.

Начиная с версии .NET Core 3.0, средство командной строки `dotnet-ef` автоматически не устанавливается. Этот пакет необходимо установить как глобальный или локальный инструмент. Если вы уже установили инструмент, вам следует удалить любую существующую версию.

1. На панели TERMINAL (Терминал) проверьте, не установили ли вы `dotnet-ef` как глобальный инструмент, как показано в следующей команде:

```
dotnet tool list –global
```

2. Проверьте в списке, установлен ли инструмент:

```
Package Id       Version                     Commands
-------------------------------------------------------
dotnet-ef        3.1.0                       dotnet-ef
```

3. Если старая версия уже установлена, удалите инструмент:

```
dotnet tool uninstall --global dotnet-ef
```

4. Установите последнюю версию:

```
dotnet tool install --global dotnet-ef --version 5.0.0
```

Подключение к базе данных

Чтобы подключиться к SQLite, необходимо знать только имя файла базы данных, которое мы указываем в строке подключения.

1. В программе Visual Studio Code убедитесь, что вы открыли папку WorkingWithEFCore, а затем на панели TERMINAL (Терминал) введите команду `dotnet new console`.
2. Отредактируйте файл `WorkingWithEFCore.csproj`, чтобы добавить ссылку на пакет поставщика данных Entity Framework Core для SQLite, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="5.0.0" />
  </ItemGroup>
</Project>
```

3. На панели TERMINAL (Терминал) соберите проект, чтобы восстановить пакеты, как показано ниже:

```
dotnet build
```

Самую последнюю версию можно проверить на сайте www.nuget.org/packages/Microsoft.EntityFrameworkCore.Sqlite/.

Определение моделей EF Core

В EF Core для создания модели во время выполнения используется сочетание *соглашений*, *атрибутов аннотаций* и операторов *Fluent API*, таким образом, что любое действие, произведенное над классами, впоследствии может быть автоматически транслировано в действие, совершаемое над действительной базой данных. Сущностный класс представляет собой структуру таблицы, а экземпляр класса — строку в этой таблице.

В первую очередь мы рассмотрим три метода определения модели с примерами кода, а затем создадим несколько классов, реализующих эти подходы.

Соглашения EF Core

В программном коде, который мы станем писать, будут использоваться нижеследующие соглашения:

- предполагается, что имя таблицы совпадает с именем свойства DbSet<T> в классе DbContext, например Products;
- предполагается, что имена столбцов совпадают с именами свойств в классе, например ProductID;
- предполагается, что тип string в .NET соответствует типу nvarchar в базе данных;
- предполагается, что тип .NET int — это тип int в базе данных;
- предполагается, что свойство с именем ID либо <имя_класса>ID (например, ProductID) — первичный ключ. Если данное свойство имеет любой целочисленный тип или тип Guid, то также предполагается, что это свойство — Identity (то есть его значение присваивается автоматически при добавлении строки в базу данных).

Существует большое количество других соглашений (более того — вы можете определить собственные), однако это выходит за рамки данной книги. Более подробную информацию можно найти на сайте https://docs.microsoft.com/ru-ru/ef/core/modeling/.

Атрибуты аннотаций Entity Framework Core

Зачастую соглашений недостаточно для определения сопоставления для всех классов и объектов базы данных. Проще всего добавить изящества в вашу модель, применив атрибуты аннотаций.

Например, в базе данных максимальная длина наименования товара равна 40, и данное значение не может быть пустым (null) в следующем коде *языка определения данных (DDL)* (выделено полужирным шрифтом).

```
CREATE TABLE Products (
  ProductID       INTEGER       PRIMARY KEY,
  ProductName     NVARCHAR (40) NOT NULL,
  SupplierID      "INT",
  CategoryID      "INT",
  QuantityPerUnit NVARCHAR (20),
```

```
  UnitPrice       "MONEY" CONSTRAINT DF_Products_UnitPrice DEFAULT (0),
  UnitsInStock    "SMALLINT" CONSTRAINT DF_Products_UnitsInStock DEFAULT (0),
  UnitsOnOrder    "SMALLINT" CONSTRAINT DF_Products_UnitsOnOrder DEFAULT (0),
  ReorderLevel    "SMALLINT" CONSTRAINT DF_Products_ReorderLevel DEFAULT (0),
  Discontinued    "BIT"      NOT NULL
                             CONSTRAINT DF_Products_Discontinued DEFAULT (0),
  CONSTRAINT FK_Products_Categories FOREIGN KEY (
    CategoryID
  )
  REFERENCES Categories (CategoryID),
  CONSTRAINT FK_Products_Suppliers FOREIGN KEY (
    SupplierID
  )
  REFERENCES Suppliers (SupplierID),
  CONSTRAINT CK_Products_UnitPrice CHECK (UnitPrice >= 0),
  CONSTRAINT CK_ReorderLevel CHECK (ReorderLevel >= 0),
  CONSTRAINT CK_UnitsInStock CHECK (UnitsInStock >= 0),
  CONSTRAINT CK_UnitsOnOrder CHECK (UnitsOnOrder >= 0)
);
```

Для указания этих ограничений в классе `Product` мы можем применить следующие атрибуты:

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

Атрибуты могут использоваться при отсутствии очевидного соответствия между типами .NET и типами базы данных.

Например, в базе данных тип столбца `UnitPrice` таблицы `Product` — `money`. В .NET нет типа `money`, поэтому вместо него мы должны воспользоваться типом `decimal`:

```
[Column(TypeName = "money")]
public decimal? UnitPrice { get; set; }
```

Ниже приведен еще один пример — таблица Categories, как показано в следующем коде DDL:

```
CREATE TABLE Categories (
    CategoryID   INTEGER       PRIMARY KEY,
    CategoryName NVARCHAR (15) NOT NULL,
    Description  "NTEXT",
    Picture      "IMAGE"
);
```

Столбец `Description` может составлять максимум 8000 символов, которые могут быть сохранены в переменной `nvarchar`, поэтому вместо этого его необходимо сопоставить с `ntext`:

```
[Column(TypeName = "ntext")]
public string Description { get; set; }
```

Entity Framework Core Fluent API

И последний способ, с помощью которого можно определить модель, — использовать *Fluent API*. Он может применяться вместо атрибутов или вдобавок к ним. Например, взглянем на следующие два атрибута в классе `Product`:

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

Эти атрибуты можно удалить из класса, чтобы сам класс был проще, и заменить их с помощью следующего оператора Fluent API в методе `OnModelBuilding` класса контекста:

```
modelBuilder.Entity<Product>()
  .Property(product => product.ProductName)
  .IsRequired()
  .HasMaxLength(40);
```

Заполнение таблиц базы данных

Для заполнения БД вы можете использовать Fluent API. EF Core автоматически определяет, какие операции вставки, обновления или удаления должны быть выполнены. Если бы мы хотели убедиться в существовании хотя бы одной строки в таблице `Product` в новой базе данных, то вызвали бы метод `HasData`, как показано ниже:

```
modelBuilder.Entity<Product>()
  .HasData(new Product
  {
    ProductID = 1,
    ProductName = "Chai",
    UnitPrice = 8.99M
  });
```

Наша модель будет сопоставлена с существующей БД, которая уже заполнена данными, поэтому нам не придется использовать эту технику в нашем коде.

Более подробную информацию о заполнении данных можно получить на сайте https://docs.microsoft.com/ru-ru/ef/core/modeling/data-seeding.

Создание модели Entity Framework Core

Теперь, когда вы уже изучили соглашения, используемые при определении моделей, создадим модель для представления двух таблиц и базы данных `Northwind`. Чтобы сделать классы более пригодными для повторного использования, мы

определим их в пространстве имен `Packt.Shared`. Эти три класса будут ссылаться друг на друга, поэтому во избежание ошибок компилятора мы сперва создадим три класса без каких-либо членов.

1. Добавьте три файла классов в проект `WorkingWithEFCore` с именами `Northwind.cs`, `Category.cs` и `Product.cs`.
2. В файле `Northwind.cs` определите класс `Northwind`:

```
namespace Packt.Shared
{
   public class Northwind
   {
   }
}
```

3. В файле `Category.cs` определите класс `Category`:

```
namespace Packt.Shared
{
   public class Category
   {
   }
}
```

4. В файле `Product.cs` определите класс `Product`:

```
namespace Packt.Shared
{
   public class Product
   {
   }
}
```

Определение сущностных классов Category и Product

Класс `Category` будет использоваться для представления строки в таблице `Categories`, состоящей из четырех столбцов (рис. 11.4).

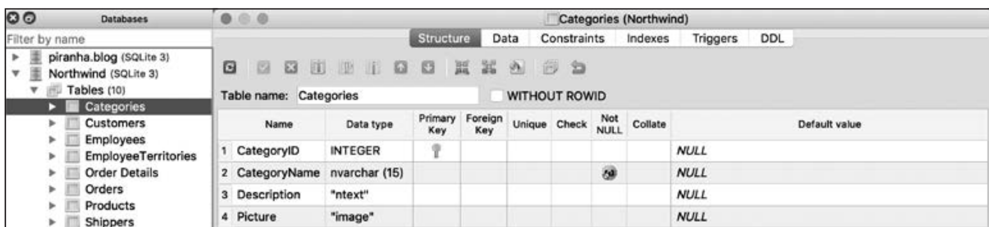


Рис. 11.4. Структура таблицы Categories

Мы будем использовать соглашения, чтобы определить три из четырех свойств (не станем отображать столбец Picture), первичный ключ и связь «один ко многим» к таблице Products. Чтобы сопоставить столбец Description с правильным типом базы данных, нам необходимо дополнить соответствующее свойство типа string атрибутом Column.

Далее в этой главе мы будем использовать Fluent API с целью определить, что CategoryName не может иметь значение null и может содержать не более 15 символов.

1. Отредактируйте код файла Category.cs:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace Packt.Shared
{
  public class Category
  {
    // эти свойства соотносятся со столбцами в базе данных
    public int CategoryID { get; set; }
    public string CategoryName { get; set; }

     [Column(TypeName = "ntext")]
    public string Description { get; set; }

    // определяет навигационное свойство для связанных строк
    public virtual ICollection<Product> Products { get; set; }

    public Category()
    {
      // чтобы позволить разработчикам добавлять товары в Category,
      // мы должны инициализировать навигационное свойство пустым списком
      this.Products = new HashSet<Product>();
    }
  }
}
```

Класс Product будет использоваться для представления строки в таблице Products, состоящей из десяти столбцов. Вам не нужно включать все столбцы из таблицы в качестве свойств класса. Мы сопоставим только следующие шесть свойств: ProductID, ProductName, UnitPrice, UnitsInStock, Discontinued и CategoryID.

Столбцы, которые не сопоставлены со свойствами, не могут быть прочитаны или установлены с помощью экземпляров класса. При использовании класса для создания нового объекта новая строка в таблице будет иметь значение null или какое-либо другое по умолчанию для значений несопоставленных столбцов в этой строке. В данном сценарии строки уже имеют значения данных,

и я решил, что у меня нет необходимости считывать эти значения в консольном приложении.

Мы можем переименовать столбец, определив свойство с другим именем, например `Cost`, а затем дополнить свойство атрибутом `[Column]` и указать имя его столбца, скажем `UnitPrice`.

Свойство `CategoryID` связано со свойством `Category`, с помощью которого мы соотнесем каждый товар с его родительской категорией.

2. Отредактируйте код файла `Product.cs`:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Packt.Shared
{
  public class Product
  {
    public int ProductID { get; set; }

    [Required]
    [StringLength(40)]
    public string ProductName { get; set; }

    [Column("UnitPrice", TypeName = "money")]
    public decimal? Cost { get; set; } // property name != field name

    [Column("UnitsInStock")]
    public short? Stock { get; set; }

    public bool Discontinued { get; set; }

    // эти две строки определяют связь внешнего ключа
    // и таблицы Categories
    public int CategoryID { get; set; }
    public virtual Category Category { get; set; }
  }
}
```

Два свойства, `Category.Products` и `Product.Category`, относящиеся к двум элементам, отмечены как виртуальные. Это позволяет EF наследовать и переопределять указанные свойства в целях предоставления дополнительной функциональности, например ленивой загрузки (недоступна в .NET Core 2.0 или более ранней версии).

Определение класса контекста базы данных Northwind

Для представления БД будет использоваться класс `Northwind`. Чтобы можно было воспользоваться технологией EF Core, класс должен наследовать от `DbContext`.

Этот класс знает, как взаимодействовать с базами данных и динамически генерировать операторы SQL для запроса и обработки данных.

Внутри вашего класса, производного от DbContext, вы должны определить хотя бы одно свойство типа DbSet<T>. Эти свойства представляют таблицы. Чтобы сообщить EF Core, какие столбцы содержит каждая таблица, тип DbSet использует дженерики для указания класса, представляющего строку таблицы, а свойства этого класса представляют столбцы таблицы.

Производный от DbContext класс должен содержать переопределенный метод OnConfiguring, который установит строку подключения к базе данных.

Аналогично производный от DbContext класс может дополнительно иметь переопределенный метод OnModelCreating. В нем вы можете добавить операторы Fluent API как альтернативу дополнения ваших сущностных классов атрибутами.

1. Отредактируйте код файла Northwind.cs:

```
using Microsoft.EntityFrameworkCore;

namespace Packt.Shared
{
  // управление соединением с базой данных
  public class Northwind : DbContext
  {
    // свойства, сопоставляемые с таблицами в базе данных
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(
      DbContextOptionsBuilder optionsBuilder)
    {
      string path = System.IO.Path.Combine(
        System.Environment.CurrentDirectory, "Northwind.db");

      optionsBuilder.UseSqlite($"Filename={path}");
    }

    protected override void OnModelCreating(
      ModelBuilder modelBuilder)
    {
      // пример использования Fluent API вместо атрибутов
      // для ограничения длины имени категории до 15 символов
      modelBuilder.Entity<Category>()
        .Property(category => category.CategoryName)
        .IsRequired() // не NULL
        .HasMaxLength(15);
```

```
      // добавлено "исправление" отсутствия поддержки
      // десятичных чисел в SQLite
      modelBuilder.Entity<Product>()
        .Property(product => product.Cost)
        .HasConversion<double>();
    }
  }
}
```

В EF Core 3.0 и более поздних версиях десятичный тип не поддерживается для сортировки и других операций. Мы можем исправить это, указав SQLite, что десятичные значения могут быть преобразованы в значения типа double. Это фактически не выполняет никакого преобразования во время выполнения.

Теперь, когда вы увидели несколько примеров определения модели сущности вручную, рассмотрим технологию, которая может сделать часть работы за вас.

Создание моделей с использованием существующей базы данных

Моделирование — это процесс использования инструмента для создания классов, представляющих модель существующей базы данных с использованием обратного проектирования. Хороший инструмент для создания шаблонов позволяет вам расширять автоматически сгенерированные классы, а затем регенерировать эти классы без потери этих расширенных классов.

Если вы знаете, что никогда не будете регенерировать классы с помощью данного инструмента, вы можете изменять код для автоматически сгенерированных классов сколько угодно раз. Код, созданный инструментом, является наилучшим вариантом.

Рассмотрим пример и ответим на вопрос, генерирует ли инструмент ту же модель, что и мы вручную.

1. На панели `TERMINAL` (Терминал) в проект `WorkingWithEFCore` добавьте пакет EF Core:

   ```
   dotnet add package Microsoft.EntityFrameworkCore.Design
   ```

2. В новой папке `AutoGenModels` создайте модель для таблиц `Categories` и `Products` с именем:

   ```
   dotnet ef dbcontext scaffold "Filename=Northwind.db" Microsoft.
   EntityFrameworkCore.Sqlite --table Categories --table Products --outputdir
   AutoGenModels --namespace Packt.Shared.AutoGen --data-annotations
   --context Northwind
   ```

Обратите внимание на следующее.

- Команда для выполнения: `dbcontext scaffold`.
- Строка подключения: `"Filename=Northwind.db"`.
- Поставщик базы данных: Microsoft.EntityFrameworkCore.Sqlite.
- Таблицы для создания моделей: `--table Categories --table` Products.
- Выходная папка: `--output-dir AutoGenModels`.
- Пространство имен: `--namespace Packt.Shared.AutoGen`.
- Чтобы использовать аннотации данных, а также Fluent API: `--data-annotations`.
- Чтобы переименовать контекст из `[`*`название_базы`*`]Context`: `--context Northwind`.

3. Обратите внимание на полученное сообщение и предупреждение, проанализируйте результаты вывода:

```
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string,
you should move it out of source code. You can avoid scaffolding the
connection string by using the Name= syntax to read it from configuration
- see https://go.microsoft.com/fwlink/?linkid=2131148. For more guidance
on storing connection strings, see http://go.microsoft.com/
/?LinkId=723263.
Could not scaffold the foreign key '0'. The referenced table could not be
found. This most likely occurred because the referenced table was excluded
from scaffolding.
```

4. Откройте папку `AutoGenModels` и обратите внимание на автоматически созданные три файла классов: `Category.cs`, `Northwind.cs` и `Product.cs`.
5. Откройте файл `Category.cs` и обратите внимание на отличия от созданного вручную:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

#nullable disable

namespace Packt.Shared.AutoGen
{
  [Index(nameof(CategoryName), Name = "CategoryName")]
  public partial class Category
  {
    public Category()
```

```
    {
      Products = new HashSet<Product>();
    }

    [Key]
    [Column("CategoryID")]
    public long CategoryId { get; set; }
    [Required]
    [Column(TypeName = "nvarchar (15)")]
    public string CategoryName { get; set; }
    [Column(TypeName = "ntext")]
    public string Description { get; set; }
    [Column(TypeName = "image")]
    public byte[] Picture { get; set; }

    [InverseProperty(nameof(Product.Category))]
    public virtual ICollection<Product> Products { get; set; }
  }
}
```

Обратите внимание на следующее.

- В настоящее время инструмент `dotnet-ef` не может использовать функцию языка ссылочных типов, допускающих значение `NULL`, поэтому он явно отключает возможность использования `NULL`.
- Данный инструмент дополняет класс сущности атрибутом `[Index]`, представленным в EF Core 5.0, который указывает свойства, соответствующие полям, которые должны содержать индекс. В более ранних версиях для определения индексов поддерживался только Fluent API.
- Имя таблицы — `Categories`, но инструмент `dotnet-ef` использует стороннюю библиотеку `Humanizer` для автоматического преобразования имени класса в единственном (или множественном) числе в категорию `Categories`, что является более естественным именем при создании единого объекта.

О библиотеке Humanizer и о том, как вы можете использовать ее в своих приложениях, вы можете прочитать на сайте http://humanizr.net.

- Класс сущности объявляется с использованием ключевого слова `partial`, чтобы вы могли создать соответствующий частичный класс для добавления дополнительного кода. Это позволяет вам повторно запустить инструмент и регенерировать класс сущности без потери лишнего кода.
- Для указания первичного ключа для данной сущности свойство `CategoryId` дополнено атрибутом `[Key]`. Также было присвоено некорректное имя, поскольку соглашения об именах Microsoft гласят, что двухбуквенные сокращения или акронимы должны содержать все прописные буквы, а не регистр заголовков.

- Свойство `Products` использует атрибут `[InverseProperty]` для определения связи внешнего ключа со свойством `Category` в классе сущности `Product`.

6. Откройте файл `Product.cs` и обратите внимание на отличия от созданного вручную.
7. Откройте файл `Northwind.cs` и обратите внимание на различия по сравнению с файлом, который вы создали вручную, как показано в следующем отредактированном коде:

```
using Microsoft.EntityFrameworkCore;

#nullable disable

namespace Packt.Shared.AutoGen
{
  public partial class Northwind : DbContext
  {
    public Northwind()
    {
    }

    public Northwind(DbContextOptions<Northwind> options)
        : base(options)
    {
    }
    public virtual DbSet<Category> Categories { get; set; }
    public virtual DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(
      DbContextOptionsBuilder optionsBuilder)
    {
      if (!optionsBuilder.IsConfigured)
      {
        # warning To protect potentially sensitive information in your
        connection string, you should move it out of source code.
        You can avoid scaffolding the connection string by using the
        Name= syntax to read it from configuration - see https://
        go.microsoft.com/fwlink/?linkid=2131148. For more guidance
        on storing connection strings, see http://go.microsoft.com/
        fwlink/?LinkId=723263.
        optionsBuilder.UseSqlite("Filename=Northwind.db");
      }
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<Category>(entity =>
      {
        entity.Property(e => e.CategoryId)
          .ValueGeneratedNever()
          .HasColumnName("CategoryID");
```

```
        entity.Property(e => e.CategoryName)
          .HasAnnotation("Relational:ColumnType", "nvarchar (15)");

        entity.Property(e => e.Description)
          .HasAnnotation("Relational:ColumnType", "ntext");

        entity.Property(e => e.Picture)
          .HasAnnotation("Relational:ColumnType", "image");
      });

      modelBuilder.Entity<Product>(entity =>
      {
        ...
      });

      OnModelCreatingPartial(modelBuilder);
    }

    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
  }
}
```

Обратите внимание на следующее.

- Класс контекста данных `Northwind` является частичным, что позволяет в дальнейшем его расширять и восстанавливать.
- Класс содержит два конструктора: один — без параметров по умолчанию и один — позволяет передавать параметры. Это полезно в приложениях, где вы хотите указать строку подключения во время выполнения.
- В методе `OnConfiguring`, если параметры не были указаны в конструкторе, по умолчанию используется строка подключения, которая ищет файл базы данных в текущей папке. Предупреждение компилятора напоминает, что вам не следует жестко кодировать информацию о безопасности в этой строке подключения.
- В методе `OnModelCreating` Fluent API используется для настройки двух классов сущностей, а затем вызывается частичный метод с именем `OnModelCreatingPartial`. Это позволяет вам реализовать этот частичный метод в вашем собственном частичном классе `Northwind`, чтобы добавить свою собственную конфигурацию Fluent API, которая не будет потеряна, если вы регенерируете классы модели.

8. Закройте автоматически сгенерированные файлы классов.

Более подробно о моделировании можно прочитать на сайте https://docs.microsoft.com/ru-ru/ef/core/managing-schemas/scaffolding?tabs=dotnet-core-cli.

В оставшейся части этой главы мы будем использовать классы, которые вы создали вручную.

Запрос данных из моделей EF Core

Теперь, когда у нас есть модель, сопоставляемая с базой данных `Northwind` и двумя ее таблицами, мы можем написать несколько простых запросов LINQ для извлечения данных. В главе 12 вы получите намного больше информации о написании запросов LINQ. А пока просто напишите код и проанализируйте результат.

1. Откройте файл `Program.cs` и импортируйте следующие пространства имен:

```
using static System.Console;
using Packt.Shared;
using Microsoft.EntityFrameworkCore;
using System.Linq;
```

2. В классе `Program` определите метод `QueryingCategories` и добавьте операторы для выполнения таких действий, как:

- создание экземпляра класса `Northwind`, который будет управлять базой данных. Экземпляры контекста базы данных рассчитаны на короткое время существования единицы работы. Их следует удалить как можно скорее, поэтому их необходимо заключить в оператор `using`;
- создание запроса для всех категорий, включая связанные товары;
- перечисление всех категорий с выводом названия и количества товаров в каждой из них.

```
static void QueryingCategories()
{
  using (var db = new Northwind())
  {
    WriteLine("Categories and how many products they have:");

    // запрос для получения всех категорий
    // и связанных с ними товаров
    IQueryable<Category> cats = db.Categories
      .Include(c => c.Products);

    foreach (Category c in cats)
    {
      WriteLine($"{c.CategoryName} has {c.Products.Count} products.");
    }
  }
}
```

3. В методе `Main` вызовите метод `QueryingCategories`:

```
static void Main(string[] args)
{
   QueryingCategories();
}
```

4. Запустите приложение и проанализируйте результат:

```
Categories and how many products they have:
Beverages has 12 products.
Condiments has 12 products.
Confections has 13 products.
Dairy Products has 10 products.
Grains/Cereals has 7 products.
Meat/Poultry has 6 products.
Produce has 5 products.
Seafood has 12 products.
```

Фильтрация включенных сущностей

В EF Core 5.0 представлены фильтры, что означает, что вы можете указать лямбда-выражение в вызове метода `Include`, чтобы отфильтровать, какие сущности возвращаются в результатах.

1. В программе определите метод `FilteredIncludes` и добавьте операторы для выполнения этих задач.
   - Создайте экземпляр класса `Northwind`, который будет управлять базой данных.
   - Предложите пользователю ввести минимальное значение для единиц на складе.
   - Создайте запрос для категорий, в которых содержатся продукты с минимальным количеством единиц на складе.
   - Перечислить все категории и продукты, выводя название и единицы на складе для каждого.

```
static void FilteredIncludes()
{
  using (var db = new Northwind())
  {
    Write("Enter a minimum for units in stock: ");
    string unitsInStock = ReadLine();
    int stock = int.Parse(unitsInStock);

    IQueryable<Category> cats = db.Categories
      .Include(c => c.Products.Where(p => p.Stock >= stock));
```

```
    foreach (Category c in cats)
    {
      WriteLine($"{c.CategoryName} has {c.Products.Count} products with
      a minimum of {stock} units in stock.");

      foreach(Product p in c.Products)
      {
        WriteLine($" {p.ProductName} has {p.Stock} units in stock.");
      }
    }
  }
}
```

2. В `Main` закомментируйте метод `QueryingCategories` и вызовите метод `FilteredIncludes`:

```
static void Main(string[] args)
{
  // QueryingCategories();
  FilteredIncludes();
}
```

3. Запустите приложение, введите минимальное количество единиц, имеющееся на складе, например 100, и проанализируйте результат:

```
Enter a minimum for units in stock: 100
Beverages has 2 products with a minimum of 100 units in stock.
  Sasquatch Ale has 111 units in stock.
  Rhönbräu Klosterbier has 125 units in stock.
Condiments has 2 products with a minimum of 100 units in stock.
  Grandma's Boysenberry Spread has 120 units in stock.
  Sirop d'érable has 113 units in stock.
Confections has 0 products with a minimum of 100 units in stock.
Dairy Products has 1 products with a minimum of 100 units in stock.
  Geitost has 112 units in stock.
Grains/Cereals has 1 products with a minimum of 100 units in stock.
  Gustaf's Knäckebröd has 104 units in stock.
Meat/Poultry has 1 products with a minimum of 100 units in stock.
  Pâté chinois has 115 units in stock.
Produce has 0 products with a minimum of 100 units in stock.
Seafood has 3 products with a minimum of 100 units in stock.
  Inlagd Sill has 112 units in stock.
  Boston Crab Meat has 123 units in stock.
  Röd Kaviar has 101 units in stock.
```

Более подробно о фильтрации можно прочитать на сайте https://docs.microsoft.com/ru-ru/ef/core/querying/related-data/eager#filtered-include.

Фильтрация и сортировка товаров

Рассмотрим более сложный запрос, который фильтрует и сортирует данные.

1. В классе `Program` определите метод `QueryingProducts` и добавьте операторы для выполнения следующих действий:
   - создание экземпляра класса `Northwind`, который будет управлять базой данных;
   - запрос у пользователя стоимости товаров;
   - с помощью запроса LINQ вывод товаров, стоимость которых выше указанной цены;
   - циклическая обработка результатов, вывод идентификатора, имени, стоимости (в долларах США) и количества единиц на складе.

```
static void QueryingProducts()
{
  using (var db = new Northwind())
  {
    WriteLine("Products that cost more than a price, highest at top.");
    string input;
    decimal price;
    do
    {
      Write("Enter a product price: ");
      input = ReadLine();
    } while(!decimal.TryParse(input, out price));

    IQueryable<Product> prods = db.Products
      .Where(product => product.Cost > price)
      .OrderByDescending(product => product.Cost);

    foreach (Product item in prods)
    {
      WriteLine(
        "{0}: {1} costs {2:$#,##0.00} and has {3} in stock.",
        item.ProductID, item.ProductName, item.Cost, item.Stock);
    }
  }
}
```

2. В методе `Main` закомментируйте предыдущий метод и вызовите следующий, как показано ниже:

```
static void Main(string[] args)
{
  // QueryingCategories();
```

```
    // FilteredIncludes();
    QueryingProducts();
  }
```

3. Запустите приложение и введите число `50`, когда программа запросит указать стоимость товара:

```
Products that cost more than a price, highest at top.
Enter a product price: 50
38: Côte de Blaye costs $263.50 and has 17 in stock.
29: Thüringer Rostbratwurst costs $123.79 and has 0 in stock.
9: Mishi Kobe Niku costs $97.00 and has 29 in stock.
20: Sir Rodney's Marmalade costs $81.00 and has 40 in stock.
18: Carnarvon Tigers costs $62.50 and has 42 in stock.
59: Raclette Courdavault costs $55.00 and has 79 in stock.
51: Manjimup Dried Apples costs $53.00 and has 20 in stock.
```

Существует ограничение с консолью, предоставленной корпорацией Microsoft в версиях Windows, выпущенных ранее Windows 10 Fall Creators Update. По умолчанию консоль не может отображать символы Unicode. Вы можете временно изменить кодовую страницу (также известную как набор символов) в консоли на Unicode UTF-8, введя следующую команду в командной строке перед запуском приложения:

```
chcp 65001
```

Получение сгенерированного SQL-кода

Вам может быть интересно, насколько хорошо написаны операторы SQL, которые генерируются из запросов C#. EF Core 5.0 представляет собой быстрый и простой способ увидеть сгенерированный SQL-код.

1. В методе `FilteredIncludes` после определения запроса добавьте оператор для вывода сгенерированного SQL:

```
IQueryable<Category> cats = db.Categories
  .Include(c => c.Products.Where(p => p.Stock >= stock));

WriteLine($"ToQueryString: {cats.ToQueryString()}");
```

2. Измените метод `Main`, чтобы закомментировать вызов метода `QueryingProducts` и раскомментировать вызов метода `FilteredIncludes`.
3. Запустите приложение, введите минимальное количество единиц, имеющихся на складе, например `99`, и проанализируйте результат:

```
Enter a minimum for units in stock: 99
ToQueryString: .param set @__stock_0 99
```

```
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description",
"t"."ProductID", "t"."CategoryID", "t"."UnitPrice", "t"."Discontinued",
"t"."ProductName", "t"."UnitsInStock"
FROM "Categories" AS "c"
LEFT JOIN (
    SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
    FROM "Products" AS "p"
    WHERE ("p"."UnitsInStock" >= @__stock_0)
) AS "t" ON "c"."CategoryID" = "t"."CategoryID"
ORDER BY "c"."CategoryID", "t"."ProductID"
Beverages has 2 products with a minimum of 99 units in stock.
  Sasquatch Ale has 111 units in stock.
  Rhönbräu Klosterbier has 125 units in stock.
...
```

4. Обратите внимание, что для параметра SQL с именем `@__ stock_0` установлено минимальное значение запаса `99`.

Логирование в EF Core

Чтобы отслеживать взаимодействие EF Core и базы данных, нам потребуется включить логирование. Это требует выполнения следующих двух задач:

- регистрации *провайдера логов*;
- реализации *логера*.

Рассмотрим следующий пример.

1. Добавьте в проект файл `ConsoleLogger.cs`.
2. Отредактируйте код файла, определив два класса, реализующие интерфейсы `ILoggerProvider` и `ILogger`, как показано в коде ниже, и обратите внимание на следующие моменты:
   - класс `ConsoleLoggerProvider` возвращает экземпляр `ConsoleLogger`. Ему не нужны неуправляемые ресурсы, поэтому метод `Dispose` ничего не выполняет, но должен существовать;
   - класс `ConsoleLogger` отключен для уровней логирования `None`, `Trace` и `Information`. Включен для всех остальных уровней;
   - класс `ConsoleLogger` реализует свой метод `Log` путем записи в `Console`.

```
using Microsoft.Extensions.Logging;
using System;
using static System.Console;

namespace Packt.Shared
{
```

```
public class ConsoleLoggerProvider : ILoggerProvider
{
  public ILogger CreateLogger(string categoryName)
  {
    return new ConsoleLogger();
  }

  // если ваш логер использует неуправляемые ресурсы,
  // то вы можете освободить память здесь
  public void Dispose() { }
}

public class ConsoleLogger : ILogger
{
  // если ваш логер использует неуправляемые ресурсы,
  // то здесь вы можете вернуть класс, реализующий IDisposable
  public IDisposable BeginScope<TState>(TState state)
  {
    return null;
  }

  public bool IsEnabled(LogLevel logLevel)
  {
    // чтобы избежать ведения лишних логов,
    // можно отфильтровать по уровню логирования
    switch(logLevel)
    {
      case LogLevel.Trace:
      case LogLevel.Information:
      case LogLevel.None:
        return false;
      case LogLevel.Debug:
      case LogLevel.Warning:
      case LogLevel.Error:
      case LogLevel.Critical:
      default:
        return true;
    };
  }

  public void Log<TState>(LogLevel,
    EventId eventId, TState state, Exception exception,
    Func<TState, Exception, string> formatter)
  {
    // регистрация идентификатора уровня и события
    Write($"Level: {logLevel}, Event ID: {eventId.Id}");

    // вывод только состояния или исключения при наличии
    if (state != null)
    {
      Write($", State: {state}");
```

```
      }
      if (exception != null)
      {
        Write($", Exception: {exception.Message}");
      }
      WriteLine();
    }
  }
}
```

3. В начало файла `Program.cs` добавьте следующие операторы для импорта пространств имен:

```
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
```

4. Для методов `QueryingCategories` и `QueryingProducts` добавьте операторы в блоке `using` для контекста базы данных `Northwind`, получающие фабрику логеров и регистрирующие наш консольный логер, как показано ниже (выделено полужирным шрифтом):

```
using (var db = new Northwind())
{
    var loggerFactory = db.GetService<ILoggerFactory>();
    loggerFactory.AddProvider(new ConsoleLoggerProvider());
```

5. Запустите консольное приложение и проанализируйте логи, которые частично показаны в следующем выводе:

```
Level: Debug, Event ID: 20000, State: Opening connection to database
'main' on server '/Users/markjprice/Code/Chapter11/WorkingWithEFCore/
Northwind.db'.
Level: Debug, Event ID: 20001, State: Opened connection to database
'main' on server '/Users/markjprice/Code/Chapter11/WorkingWithEFCore/
Northwind.db'.
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT "product"."ProductID", "product"."CategoryID",
"product"."UnitPrice", "product"."Discontinued", "product"."ProductName",
"product"."UnitsInStock"
FROM "Products" AS "product"
ORDER BY "product"."UnitPrice" DESC
```

Значения идентификатора события и их смысл будут зависеть от поставщика данных .NET. Если мы хотим узнать, как запрос LINQ был преобразован в операторы SQL и выполняется, то выводимое значение `Event ID` будет равно `20100`.

6. Измените метод `Log` в классе `ConsoleLogger` для вывода только событий с `Id`, равным `20100`, как показано в следующем коде (выделено полужирным шрифтом):

```
public void Log<TState>(LogLevel logLevel, EventId eventId, TState
state,
  Exception exception,Func<TState, Exception, string> formatter)
{
  if (eventId.Id == 20100)
  {
    // регистрация идентификатора уровня и события
    Write("Level: {0}, Event ID: {1}, Event: {2}"
      logLevel, eventId.Id, eventId.Name);

    // вывод только состояния или исключения при наличии
    if (state != null)
    {
      Write($", State: {state}");
    }
    if (exception != null)
    {
      Write($", Exception: {exception.Message}");
    }
    WriteLine();
  }
}
```

7. В методе `Main` раскомментируйте метод `QueryingCategories` и закомментируйте метод `QueryingProducts`, чтобы можно было отслеживать операторы SQL, которые генерируются при объединении двух таблиц.
8. Запустите консольное приложение и обратите внимание на следующие операторы SQL, которые были залогированы:

```
Categories and how many products they have:
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description"
FROM "Categories" AS "c"
ORDER BY "c"."CategoryID"
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT "c.Products"."ProductID", "c.Products"."CategoryID",
"c.Products"."UnitPrice", "c.Products"."Discontinued",
"c.Products"."ProductName", "c.Products"."UnitsInStock"
FROM "Products" AS "c.Products"
INNER JOIN (
SELECT "c0"."CategoryID"
FROM "Categories" AS "c0"
```

```
) AS "t" ON "c.Products"."CategoryID" = "t"."CategoryID"
ORDER BY "t"."CategoryID"
Beverages has 12 products.
Condiments has 12 products.
Confections has 13 products.
Dairy Products has 10 products.
Grains/Cereals has 7 products.
Meat/Poultry has 6 products.
Produce has 5 products.
Seafood has 12 products.
```

Теги запросов

При логировании запросов LINQ может быть сложно соотнести с этими запросами записи в логах в сложных сценариях. В EF Core 2.2 добавлена функция тегов запросов, позволяющая добавлять комментарии SQL в лог.

Вы можете аннотировать запрос LINQ с помощью метода `TagWith`, как показано ниже:

```
IQueryable<Product> prods = db.Products
  .TagWith("Products filtered by price and sorted.")
  .Where(product => product.Cost > price)
  .OrderByDescending(product => product.Cost);
```

Этот код добавит комментарий SQL в лог:

```
-- Products filtered by price and sorted.
```

Более подробную информацию о тегах запросов можно найти на сайте https://docs.microsoft.com/ru-ru/ef/core/querying/tags.

Сопоставление с образцом с помощью оператора Like

EF Core поддерживает распространенные операторы SQL, включая `Like` для сопоставления с образцом.

1. В классе `Program` добавьте метод `QueryingWithLike`, как показано в коде ниже, и обратите внимание на следующие моменты:
   - мы включили ведение лога;
   - мы предлагаем пользователю ввести фрагмент наименования товара, а затем применить метод `EF.Functions.Like` для поиска в любом месте свойства `ProductName`;
   - для каждого соответствующего товара мы выводим его название, наличие на складе и информацию о том, снят ли он с производства.

```
static void QueryingWithLike()
{
  using (var db = new Northwind())
  {
    var loggerFactory = db.GetService<ILoggerFactory>();
    loggerFactory.AddProvider(new ConsoleLoggerProvider());
    Write("Enter part of a product name: ");
    string input = ReadLine();

    IQueryable<Product> prods = db.Products
      .Where(p => EF.Functions.Like(p.ProductName, $"%{input}%"));

    foreach (Product item in prods)
    {
      WriteLine("{0} has {1} units in stock. Discontinued? {2}",
        item.ProductName, item.Stock, item.Discontinued);
    }
  }
}
```

2. В методе `Main` закомментируйте существующие методы и вызовите метод `QueryingWithLike`.
3. Запустите консольное приложение, введите фрагмент наименования товара, например `che`, и проанализируйте результат:

```
Enter part of a product name: che
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[@__
Format_1='?' (Size = 5)], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE "p"."ProductName" LIKE @__Format_1
Chef Anton's Cajun Seasoning has 53 units in stock. Discontinued? False
Chef Anton's Gumbo Mix has 0 units in stock. Discontinued? True
Queso Manchego La Pastora has 86 units in stock. Discontinued? False
Gumbär Gummibärchen has 15 units in stock. Discontinued? False
```

Определение глобальных фильтров

Продукция `Northwind` может быть снята с производства, поэтому возникает необходимость убедиться в том, что такие товары никогда не будут возвращаться в результате, даже если разработчик забывает использовать фильтрацию.

1. Измените метод `OnModelCreating` в классе `Northwind`, чтобы добавить глобальный фильтр для удаления снятых с производства товаров, как показано ниже (выделено полужирным шрифтом):

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  // пример использования Fluent API вместо атрибутов
  // для ограничения длины имени категории до 15
  modelBuilder.Entity<Category>()
    .Property(category => category.CategoryName)
    .IsRequired() // NOT NULL
    .HasMaxLength(15);

  // добавление поддержки десятичных чисел в SQLite
  modelBuilder.Entity<Product>()
    .Property(product => product.Cost)
    .HasConversion<double>();

  // глобальный фильтр для удаления снятой с производства продукции
  modelBuilder.Entity<Product>()
    .HasQueryFilter(p => !p.Discontinued);
}
```

2. Запустите консольное приложение, введите фрагмент наименования товара `che`, проанализируйте результат и обратите внимание, что *Chef Anton's Gumbo Mix* теперь отсутствует, поскольку сгенерированный оператор SQL содержит фильтр для столбца `Discontinued`:

```
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND "p"."ProductName" LIKE @__Format_1
Chef Anton's Cajun Seasoning has 53 units in stock. Discontinued? False
Queso Manchego La Pastora has 86 units in stock. Discontinued? False
Gumbär Gummibärchen has 15 units in stock. Discontinued? False
```

Схемы загрузки данных при использовании EF Core

Entity Framework предоставляет три наиболее популярные *схемы загрузки данных*: *ленивую*, *жадную* и *явную загрузку*. В этом разделе я опишу каждую из них.

Жадная загрузка элементов

В методе `QueryingCategories` программного кода в настоящий момент свойство `Categories` служит для циклического прохождения каждой категории и вывода имени категории и количества товаров в ней. Этот код работает потому, что при написании запроса мы использовали метод `Include`, позволяющий воспользоваться жадной загрузкой (также известной как *ранняя загрузка*) связанных товаров.

1. Измените запрос, закомментировав вызов метода `Include`:

```
IQueryable<Category> cats =
  db.Categories; //.Include(c => c.Products);
```

2. В методе `Main` закомментируйте все методы, за исключением `QueryingCategories`.
3. Запустите консольное приложение и проанализируйте результаты вывода (фрагмент):

```
Beverages has 0 products.
Condiments has 0 products.
Confections has 0 products.
Dairy Products has 0 products.
Grains/Cereals has 0 products.
Meat/Poultry has 0 products.
Produce has 0 products.
Seafood has 0 products.
```

Каждый элемент в цикле `foreach` — это экземпляр класса `Category`, у которого есть свойство `Products`, представляющее, в свою очередь, список товаров в данной категории. Поскольку исходный запрос осуществляет выборку только из таблицы `Categories`, значение данного свойства является пустым для каждой категории.

Использование ленивой загрузки

Ленивая загрузка была введена в EF Core 2.1 и может автоматически загружать отсутствующие связанные данные.

Чтобы разрешить ленивую загрузку, разработчики должны выполнить следующие действия:

- сослаться на пакет NuGet для прокси;
- настроить ленивую загрузку для использования прокси.

1. Откройте файл `WorkingWithEFCore.csproj` и добавьте ссылку на пакет, как показано ниже:

```
<PackageReference
  Include="Microsoft.EntityFrameworkCore.Proxies"
  Version="5.0.0" />
```

2. На панели TERMINAL (Терминал) соберите проект для восстановления пакетов, как показано в следующей команде:

```
dotnet build
```

3. Откройте файл `Northwind.cs`, импортируйте пространство имен `Microsoft.EntityFrameworkCore.Proxies` и выполните вызов метода расширения, чтобы

применить прокси с ленивой загрузкой перед использованием SQLite, как показано ниже (выделено полужирным шрифтом):

```
optionsBuilder.UseLazyLoadingProxies()
  .UseSqlite($"Filename={path}");
```

Теперь система будет автоматически проверять состояние загрузки элементов при каждом циклическом перечислении и попытке считывания свойства `Products`. Если требуемые элементы еще не загружены, то EF Core «лениво» загрузит их для нас, выполнив выражение `Select` для подгрузки только набора товаров выбранной категории, после чего корректное количество товаров будет возвращено в программный вывод.

4. Запустите консольное приложение, и вы увидите, что проблема ленивой загрузки заключается в необходимости каждый раз связываться с сервером базы данных для постепенной выборки всех данных, как показано в следующем фрагменте вывода:

```
Categories and how many products they have:
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description"
FROM "Categories" AS "c"
Level: Debug, Event ID: 20100, State: Executing DbCommand
[Parameters=[@__p_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryID" = @__p_0)
Beverages has 11 products.
Level: Debug, Event ID: 20100, State: Executing DbCommand
[Parameters=[@__p_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryID" = @__p_0)
Condiments has 11 products.
```

Явная загрузка элементов

Еще один тип — явная загрузка. Она работает так же, как ленивая, с тем лишь исключением, что вы контролируете, какие именно связанные данные будут загружены в тот или иной момент времени.

1. В методе QueryingCategories измените следующие операторы, чтобы отключить ленивую загрузку, а затем запросите у пользователя, желает ли он выполнить жадную и явную загрузки:

```
IQueryable<Category> cats;
// = db.Categories;
// .Include(c => c.Products);

db.ChangeTracker.LazyLoadingEnabled = false;

Write("Enable eager loading? (Y/N): ");
bool eagerloading = (ReadKey().Key == ConsoleKey.Y);
bool explicitloading = false;
WriteLine();

if (eagerloading)
{
  cats = db.Categories.Include(c => c.Products);
}
else
{
  cats = db.Categories;

  Write("Enable explicit loading? (Y/N): ");
  explicitloading = (ReadKey().Key == ConsoleKey.Y);
  WriteLine();
}
```

2. В цикл foreach перед вызовом метода WriteLine добавьте операторы, чтобы проверить, включена ли явная загрузка. Если да, то спросите пользователя, хочет ли он явно загрузить каждую отдельную категорию:

```
if (explicitloading)
{
  Write($"Explicitly load products for {c.CategoryName}? (Y/N): ");
  ConsoleKeyInfo key = ReadKey().Key;
  WriteLine();
  if (key.Key == ConsoleKey.Y)
  {
    var products = db.Entry(c).Collection(c2 => c2.Products);
    if (!products.IsLoaded) products.Load();
  }
}
WriteLine($"{c.CategoryName} has {c.Products.Count} products.");
```

3. Запустите консольное приложение; нажмите клавишу N, чтобы отключить жадную загрузку, и клавишу Y, чтобы включить явную. По желанию для каждой категории нажмите клавишу Y или N для загрузки товаров, содержащихся в этой

категории. Я решил загружать товары, относящиеся только к двум из восьми категорий — Beverages и Seafood:

```
Categories and how many products they have:
Enable eager loading? (Y/N): n
Enable explicit loading? (Y/N): y
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description"
FROM "Categories" AS "c"
Explicitly load products for Beverages? (Y/N): y
Level: Debug, Event ID: 20100, State: Executing DbCommand
[Parameters=[@__p_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryID" = @__p_0)
Beverages has 11 products.
Explicitly load products for Condiments? (Y/N): n
Condiments has 0 products.
Explicitly load products for Confections? (Y/N): n
Confections has 0 products.
Explicitly load products for Dairy Products? (Y/N): n
Dairy Products has 0 products.
Explicitly load products for Grains/Cereals? (Y/N): n
Grains/Cereals has 0 products.
Explicitly load products for Meat/Poultry? (Y/N): n
Meat/Poultry has 0 products.
Explicitly load products for Produce? (Y/N): n
Produce has 0 products.
Explicitly load products for Seafood? (Y/N): y
Level: Debug, Event ID: 20100, State: Executing DbCommand
[Parameters=[@__p_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductID", "p"."CategoryID", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryID" = @__p_0)
Seafood has 12 products.
```

Внимательно выбирайте схему загрузки данных, подходящую именно для вашего программного кода. Использование ленивой загрузки может сделать вас буквально ленивым разработчиком баз данных! Больше о схемах загрузки можно узнать на сайте https://docs.microsoft.com/ru-ru/ef/core/querying/related-data.

Управление данными с помощью EF Core

EF Core позволяет легко добавлять, обновлять и удалять элементы. DbContext автоматически поддерживает отслеживание изменений, что позволяет отслеживать множественные изменения сущностей локально, включая добавление новых, изменение существующих и удаление сущностей. Когда вы будете готовы отправить эти изменения в основную базу данных, вызовите метод SaveChanges. Количество успешно измененных объектов будет возвращено.

Добавление элементов

Добавим новую строку в таблицу.

1. В классе Program создайте метод AddProduct:

```
static bool AddProduct(
  int categoryID, string productName, decimal? price)
{
  using (var db = new Northwind())
  {
    var newProduct = new Product
    {
      CategoryID = categoryID,
      ProductName = productName,
      Cost = price
    };

    // пометить товар как отслеживаемый на предмет изменений
    db.Products.Add(newProduct);

    // сохранить отслеживаемые изменения в базе данных
    int affected = db.SaveChanges();
    return (affected == 1);
  }
}
```

2. В классе Program создайте метод ListProducts, который выводит идентификатор, наименование, стоимость, запас и товары, снятые с производства, отсортированные по стоимости от самого дорогого:

```
static void ListProducts()
{
  using (var db = new Northwind())
  {
    WriteLine("{0,-3} {1,-35} {2,8} {3,5} {4}",
      "ID", "Product Name", "Cost", "Stock", "Disc.");
```

```
    foreach (var item in db.Products.OrderByDescending(p => p.Cost))
    {
      WriteLine("{0:000} {1,-35} {2,8:$#,##0.00} {3,5} {4}",
        item.ProductID, item.ProductName, item.Cost,
        item.Stock, item.Discontinued);
    }
  }
}
```

Помните, что `1, -35` означает, что аргумент № 1 должен выравниваться по левому краю в столбце шириной 35 символов, а `3,5` — что аргумент № 3 должен выравниваться по правому краю в столбце шириной пять символов.

3. В методе `Main` закомментируйте предыдущие вызовы методов, а затем вызовите методы `AddProduct` и `ListProducts`:

```
static void Main(string[] args)
{
  // QueryingCategories();
  // FilteredIncludes();
  // QueryingProducts();
  // QueryingWithLike();

  if (AddProduct(6, "Bob's Burgers", 500M))
  {
    WriteLine("Add product successful.");
  }
  ListProducts();
}
```

4. Запустите приложение, проанализируйте результат и обратите внимание, что новый товар был добавлен, как показано ниже во фрагменте вывода:

```
Add product successful.
ID Product Name              Cost Stock Disc.
078 Bob's Burgers         $500.00       False
038 Côte de Blaye         $263.50    17 False
020 Sir Rodney's Marmalade $81.00    40 False
...
```

Обновление элементов

Изменим существующую строку в таблице.

1. В классе `Program` добавьте метод, чтобы на 20 долларов увеличить цену первого товара, название которого начинается со слова `Bob`:

```
static bool IncreaseProductPrice(string name, decimal amount)
{
```

```
  using (var db = new Northwind())
  {
    // получить первый товар, название которого начинается с name
    Product updateProduct = db.Products.First(
      p => p.ProductName.StartsWith(name));

    updateProduct.Cost += amount;
    int affected = db.SaveChanges();
    return (affected == 1);
  }
}
```

2. В методе `Main` закомментируйте весь блок оператора `if`, вызывающий метод `AddProduct`, и добавьте вызов метода `IncreaseProductPrice` перед вызовом списка товаров, как показано ниже (выделено полужирным шрифтом):

```
if (IncreaseProductPrice("Bob", 20M))
{
  WriteLine("Update product price successful.");
}

ListProducts();
```

3. Запустите консольное приложение, проанализируйте результат и обратите внимание, что стоимость уже существующего товара Bob's Burgers увеличилась на 20 долларов:

```
Update product price successful.
ID Product Name          Cost Stock Disc.
078 Bob's Burgers     $520.00       False
038 Côte de Blaye     $263.50    17 False
...
```

Удаление элементов

Удалим строку из таблицы.

1. В классе `Program` импортируйте пространство имен `System.Collections.Generic`, а затем добавьте метод для удаления всех товаров, название которых начинается со слова `Bob`:

```
static int DeleteProducts(string name)
{
  using (var db = new Northwind())
  {
    IEnumerable<Product> products = db.Products.Where(
      p => p.ProductName.StartsWith(name));
```

```
      db.Products.RemoveRange(products);
      int affected = db.SaveChanges();
      return affected;
    }
  }
```

Вы можете удалить отдельные объекты с помощью метода `Remove`. При удалении нескольких объектов используется более эффективный метод `RemoveRange`.

2. В методе `Main` закомментируйте весь блок оператора `if`, вызывающий метод `IncreaseProductPrice`, и добавьте вызов метода `DeleteProducts`, как показано ниже (выделено полужирным шрифтом):

```
int deleted = DeleteProducts("Bob");
WriteLine($"{deleted} product(s) were deleted.");
ListProducts();
```

3. Запустите консольное приложение и проанализируйте результат:

```
1 product(s) were deleted.
ID Product Name Cost Stock Disc.
038 Côte de Blaye $263.50 17 False
020 Sir Rodney's Marmalade $81.00 40 False
```

Удаляются все наименования товаров, начинающиеся со слова `Bob`. В качестве дополнительной задачи раскомментируйте операторы, добавив три новых наименования товаров, начинающихся со слова `Bob`, а затем удалите их.

Пулы соединений с базами данных

Класс `DbContext` требует очистки неуправляемых ресурсов и спроектирован по принципу «одна единица работы» (single-unit-of-work). В предыдущих примерах кода мы создавали все наследуемые от класса `DbContext` экземпляры `Northwind` в блоке `using`.

Особенность ASP.NET Core, связанная с EF Core, делает ваш код более эффективным, объединяя контексты базы данных при создании веб-приложений и веб-сервисов.

Это позволяет вам создавать и удалять столько наследуемых от класса `DbContext` объектов, сколько вы хотите, зная, что ваш код по-прежнему очень эффективен.

Узнать больше о пулах соединений с базами данных можно на сайте https://docs.microsoft.com/ru-ru/ef/core/what-is-new/ef-core-2.0#dbcontext-pooling.

Транзакции

Каждый раз, когда вы вызываете метод `SaveChanges`, система запускает *неявную транзакцию* таким образом, что если нечто пойдет не так, то система автоматически отменит все внесенные изменения. В случае же успешного завершения выполнения всех изменений транзакция считается совершенной.

Транзакции позволяют сохранить целостность вашей базы данных с помощью блокировки чтения и записи до момента завершения последовательности операций.

В англоязычной литературе для характеристики транзакций принято использовать аббревиатуру *ACID*.

- *A* (atomic — «неделимый») — совершаются либо все операции текущей транзакции, либо ни одна из операций.
- *C* (consistent — «согласованный») — ваша база данных согласована как до, так и после совершения транзакции. Это зависит от логики вашего программного кода. Например, при переводе денег между банковскими счетами ваша бизнес-логика должна гарантировать, что, дебетуя 100 долларов США на одном счете, вы кредитуете 100 долларов США на другом.
- *I* (isolated — «изолированный») — во время выполнения транзакции вносимые изменения не видны другим процессам. Существует несколько уровней изолированности, которые вы можете установить (табл. 11.2). Чем выше уровень, тем выше целостность данных. Однако для этого требуется применить множество блокировок, что негативно отразится на работе других процессов. Снимки состояния — особый уровень изоляции, при использовании которого создаются копии строк таблицы, позволяющие избежать блокировок, что приводит к увеличению размера базы данных при выполнении транзакций.
- *D* (durable — «надежный») — в случае возникновения ошибки при выполнении транзакции исходное состояние базы данных может быть восстановлено. В данном случае «надежный» — антоним «неустойчивого» (volatile).

Таблица 11.2

Уровень изолированности	Блокировка (-и)	Допустимые проблемы целостности данных
ReadUncommitted	Нет	Грязное чтение, неповторяемое чтение, фантомные данные
ReadCommitted	При редактировании применяется блокировка, предотвращающая чтение записи (-ей) другими пользователями до завершения транзакции	Неповторяемое чтение и фантомные данные

Продолжение ⇗

Таблица 11.2 *(продолжение)*

Уровень изолированности	Блокировка (-и)	Допустимые проблемы целостности данных
RepeatableRead	При чтении применяется блокировка, предотвращающая редактирование записи (-ей) другими пользователями до завершения транзакции	Фантомные данные
Serializable	Применяются блокировки уровня диапазона ключа, предотвращающие любые действия, способные повлиять на результат, в том числе вставка и удаление данных	Нет
Snapshot	Нет	Нет

Определение явной транзакции

Вы можете управлять явными транзакциями с помощью свойства `Database` контекста базы данных.

1. Импортируйте следующее пространство имен, чтобы подключить возможность использования интерфейса `IDbContextTransaction`:

```
using Microsoft.EntityFrameworkCore.Storage;
```

2. В методе `DeleteProducts` после создания экземпляра переменной `db` добавьте операторы (выделены полужирным шрифтом), чтобы запустить явную транзакцию и вывести ее уровень изолированности. В конце метода подтвердите транзакцию и закройте фигурную скобку:

```
static int DeleteProducts(string name)
{
  using (var db = new Northwind())
  {
    using (IDbContextTransaction t = db.Database.BeginTransaction())
    {
      WriteLine("Transaction isolation level: {0}",
        t.GetDbTransaction().IsolationLevel);

      var products = db.Products.Where(
        p => p.ProductName.StartsWith(name));

      db.Products.RemoveRange(products);

      int affected = db.SaveChanges();
      t.Commit();
```

```
            return affected;
        }
    }
}
```

3. Запустите консольное приложение и проанализируйте результат:

```
Transaction isolation level: Serializable
```

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 11.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какой тип вы бы использовали для свойства, представляющего таблицу, например для свойства `Products` контекста базы данных?
2. Какой тип вы бы применили для свойства, которое представляет отношение «один ко многим», скажем для свойства `Products` объекта `Category`?
3. Какое соглашение, касающееся первичных ключей, действует в EF?
4. Когда бы вы воспользовались атрибутом аннотаций в классе сущности?
5. Почему вы предпочли бы задействовать Fluent API, а не атрибуты аннотаций?
6. Что означает уровень изолированности транзакции Serializable?
7. Что метод `DbContext.SaveChanges()` возвращает в результате?
8. В чем разница между жадной и явной загрузкой?
9. Как определить сущностный класс EF Core для соответствия следующей таблице?

```
CREATE TABLE Employees(
  EmpID INT IDENTITY,
  FirstName NVARCHAR(40) NOT NULL,
  Salary MONEY
)
```

10. Какую выгоду вы получаете от объявления свойств навигации как `virtual`?

Упражнение 11.2. Экспорт данных с помощью различных форматов сериализации

Создайте консольное приложение `Exercise02`, которое запрашивает базу данных `Northwind` для всех категорий и товаров, а затем сериализует данные, используя как минимум три формата сериализации, доступные для .NET.

Какой формат сериализации задействует наименьшее количество байтов?

Упражнение 11.3. Изучение документации EF Core

Более углубленно темы, рассматриваемые в этой главе, вы можете изучить на сайте https://docs.microsoft.com/ru-ru/ef/core/.

Резюме

Вы научились подключаться к базе данных, выполнять простой запрос LINQ и обрабатывать результаты. Вы научились добавлять, изменять и удалять данные, а также создавать модели сущностей для имеющейся базы данных, такой как `Northwind`.

Далее вы узнаете, как писать более сложные запросы LINQ, предназначенные для выбора, фильтрации, сортировки, объединения и группировки.

12 Создание запросов и управление данными с помощью LINQ

Данная глава посвящена технологии *LINQ* (Language Integrated Query — запрос, интегрированный в язык), представляющей собой набор языковых расширений, которые добавляют возможность работы с последовательностью элементов с их дальнейшей фильтрацией, сортировкой и проецированием в различные структуры данных.

В этой главе:

- написание запросов LINQ;
- работа с множествами с помощью LINQ;
- использование LINQ с EF Core;
- подслащение синтаксиса LINQ с помощью синтаксического сахара;
- использование нескольких потоков и параллельного LINQ;
- создание собственных методов расширения LINQ;
- работа с LINQ to XML.

Написание запросов LINQ

Несмотря на то что в главе 11 мы уже написали несколько запросов LINQ, я не объяснил подробно работу технологии LINQ.

LINQ состоит из нескольких частей, одни из них обязательные, а другие — опциональные.

- *Методы расширения (обязательно)* — набор методов включает в себя `Where`, `OrderBy`, `Select`. Эти методы — как раз то, что предоставляет функциональность LINQ.
- *Поставщики данных LINQ (обязательно)* — данный набор включает в себя LINQ to Objects, LINQ to Entities, LINQ to XML, LINQ to OData, LINQ to Amazon.

Эти поставщики конвертируют стандартные операции LINQ в команды, специфичные для различных видов данных.

- *Лямбда-выражения (опционально)* — могут использоваться вместо имен методов для упрощения вызовов методов расширения LINQ.
- *Понятный синтаксис запросов LINQ (опционально)* — включает в себя такие ключевые слова, как `from`, `in`, `where`, `orderby`, `descending`, `select`. Вышеперечисленное — ключевые слова языка C#, являющиеся псевдонимами для некоторых методов расширения LINQ. Использование этих псевдонимов поможет упростить написание запросов, особенно если у вас уже есть опыт работы с другими языками написания запросов, такими как *Structured Query Language (SQL)*.

При первом знакомстве с LINQ многие разработчики зачастую полагают, что понятный синтаксис запросов и есть LINQ. Однако это одна из необязательных частей LINQ!

Расширение последовательностей с помощью класса Enumerable

Класс `Enumerable` предоставляет такие методы расширения, как `Where` и `Select`, любому типу, реализующему интерфейс `IEnumerable<T>`. Такой тип также называют *последовательностью*.

Например, массив любого типа реализует `IEnumerable<T>`, где `T` — это тип элемента массива, а значит, все массивы поддерживают создание запросов и управление с помощью LINQ.

Все дженерик-коллекции, такие как `List<T>`, `Dictionary<TKey, TValue>`, `Stack<T>` и `Queue<T>`, реализуют `IEnumerable<T>`, так что и для этих коллекций можно создавать запросы и управлять ими с помощью LINQ.

Класс `Enumerable` определяет более 45 методов расширения, как вы можете увидеть в табл. 12.1.

Таблица 12.1

Метод (-ы)	Описание
First, FirstOrDefault, Last, LastOrDefault	Получает первый или последний элемент в последовательности или возвращает значение по умолчанию для типа, например 0 для типа int, null для ссылочного типа при отсутствии первого или последнего элемента
Where	Возвращает последовательность элементов, соответствующих указанному фильтру

Метод (-ы)	Описание
Single, SingleOrDefault	Возвращает элемент, который соответствует определенному фильтру, или генерирует исключение, или возвращает значение по умолчанию для типа при отсутствии строго одного совпадения
ElementAt, ElementAtOrDefault	Возвращает элемент в указанной позиции индекса, или генерирует исключение, или возвращает значение по умолчанию для типа при отсутствии в этой позиции элемента
Select, SelectMany	Проецирует элементы в другую форму (другой тип) и делает плоской вложенную иерархию элементов
OrderBy, OrderByDescending, ThenBy, ThenByDescending	Сортирует элементы по указанному свойству
Reverse	Меняет порядок элементов
GroupBy, GroupJoin, Join	Группирует и объединяет последовательности
Skip, SkipWhile	Пропускает несколько элементов или пропускает элементы, пока выражение true (верно)
Take, TakeWhile	Берет несколько элементов или берет элементы, пока выражение true (верно)
Aggregate, Average, Count, LongCount, Max, Min, Sum	Рассчитывает агрегированное значение
All, Any, Contains	Возвращает значение true, если все или какие-либо элементы соответствуют фильтру или последовательность содержит указанный элемент
Cast	Преобразует элементы в указанный тип
OfType	Удаляет элементы, не соответствующие указанному типу
Except, Intersect, Union	Выполняет операции, которые возвращают множества. Множества не могут иметь повторяющиеся элементы. Входные данные этих методов могут быть любой последовательностью, поэтому могут иметь дубликаты, однако результат всегда является множеством
Append, Concat, Prepend	Выполняет операции объединения последовательностей
Zip	Выполняет операцию сопоставления в зависимости от положения элементов
Distinct	Удаляет из последовательности повторяющиеся элементы.
ToArray, ToList, ToDictionary, ToLookup	Преобразует последовательность в массив или коллекцию

Фильтрация элементов с помощью метода Where

Наиболее распространенная причина применения LINQ заключается в фильтрации элементов в последовательности с помощью метода расширения `Where`. Рассмотрим

фильтрацию элементов, определив последовательность имен, а затем применив к ней операции LINQ.

1. Создайте в папке Code папку Chapter12 с подпапкой LinqWithObjects.
2. В программе Visual Studio Code в папке Chapter12 сохраните рабочую область под именем `Chapter12.code-workspace`.
3. Добавьте в рабочую область папку LinqWithObjects.
4. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал).
5. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new console
```

6. В файле `Program.cs` добавьте метод `LinqWithArrayOfStrings`, который определяет массив строковых значений, а затем вызывает метод расширения `Where`, как показано ниже:

```
static void LinqWithArrayOfStrings()
{
  var names = new string[] { "Michael", "Pam", "Jim", "Dwight",
    "Angela", "Kevin", "Toby", "Creed" };
  var query = names.
}
```

7. Во время ввода метода `Where` обратите внимание, что метод с таким именем не показывается в списке членов массива `string` в подсказке IntelliSense (рис. 12.1).

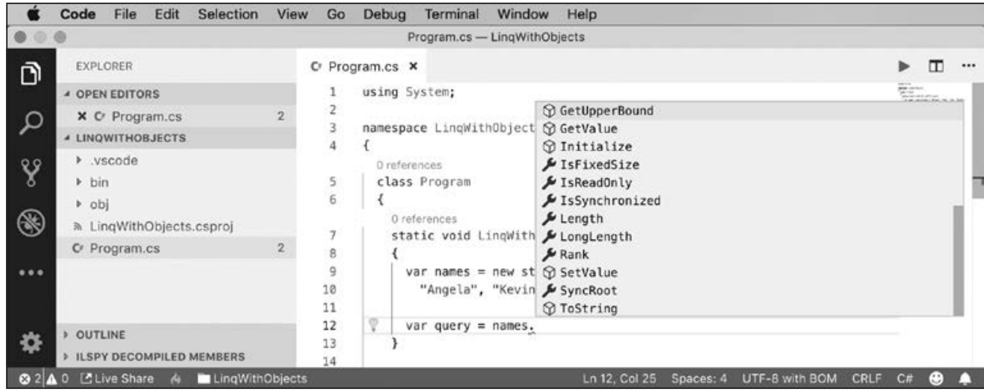


Рис. 12.1. IntelliSense не отображает метод Where

Это происходит потому, что `Where` — *метод расширения*. Он не существует для типа массива. Данный метод существует в отдельной сборке и пространстве

имен. Чтобы сделать `Where` доступным, мы должны импортировать пространство имен `System.Linq`.

8. Добавьте следующий оператор в начало файла `Program.cs`:

```
using System.Linq;
```

9. Повторно введите метод `Where` и обратите внимание, что в подсказке IntelliSense отображено еще несколько методов, включая методы расширения, добавленные классом `Enumerable` (рис. 12.2).

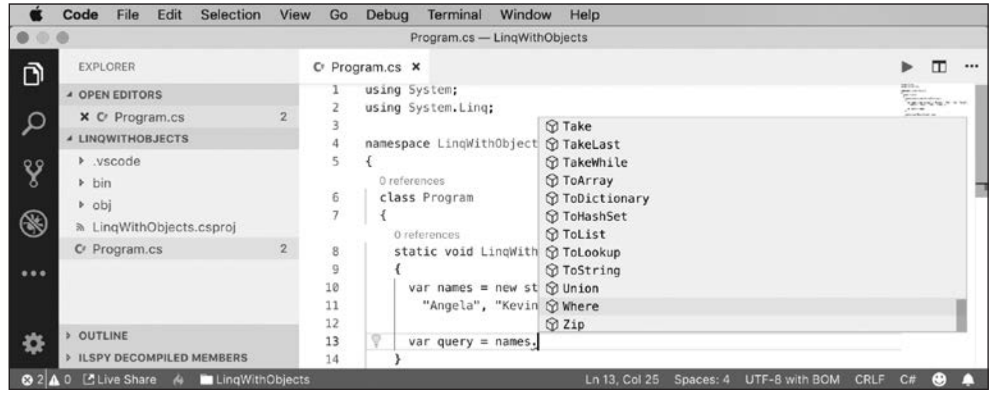


Рис. 12.2. IntelliSense теперь отображает намного больше методов

10. Теперь при вводе скобок после имени метода `Where` обратите внимание на подсказки IntelliSense. Система сообщает, что для вызова метода `Where` нам необходимо передать его в экземпляре делегата `Func<string, bool>` (рис. 12.3).

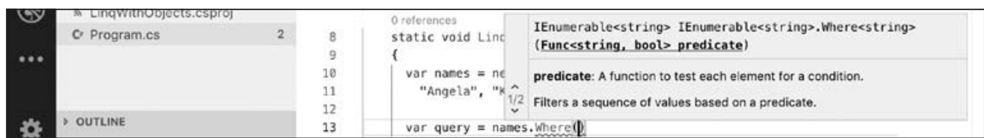


Рис. 12.3. Сигнатура метода, показывающая, что необходимо передать делегат Func<string, bool>

11. Введите следующее выражение для создания нового экземпляра делегата `Func<string, bool>` и обратите внимание, что мы еще не указали имя метода, поскольку определим его далее:

```
var query = names.Where(new Func<string, bool>())
```

Делегат `Func<string, bool>` сообщает нам, что для каждой строковой переменной, передаваемой методу, он должен возвращать значение `bool`. Если метод возвращает

true, то это значит, что нам следует включить строку в результаты, а если false — исключить ее.

Ссылка на именованные методы

Далее определим метод, включающий только имена длиннее четырех символов.

1. Добавьте в класс Program метод:

```
static bool NameLongerThanFour(string name)
{
  return name.Length > 4;
}
```

2. Вернувшись в метод LinqWithArrayOfStrings, передайте имя метода в делегат Func<string, bool>, а затем переберите элементы запроса, как показано ниже:

```
var query = names.Where(
  new Func<string, bool>(NameLongerThanFour));

foreach (string item in query)
{
  WriteLine(item);
}
```

3. В методе Main вызовите метод LinqWithArrayOfStrings, запустите консольное приложение и проанализируйте результат, отметив, что перечислены только имена длиннее четырех символов:

```
Michael
Dwight
Angela
Kevin
Creed
```

Упрощение кода путем удаления явного создания экземпляров делегатов

Мы можем упростить программный код, удалив явное создание экземпляра делегата Func<string, bool>. Компилятор C# создаст этот экземпляр за нас.

1. Чтобы помочь вам узнать улучшенный код, скопируйте и вставьте следующий запрос.
2. Закомментируйте первый пример:

```
// var query = names.Where(
// new Func<string, bool>(NameLongerThanFour));
```

3. Измените копию, чтобы удалить явное создание экземпляра делегата, как показано ниже:

```
var query = names.Where(NameLongerThanFour);
```

4. Перезапустите приложение и обратите внимание, что его поведение не изменилось.

Использование лямбда-выражения

Мы можем еще больше упростить код с помощью *лямбда-выражений* вместо именованных методов.

Несмотря на то что изначально это может казаться сложным, лямбда-выражение, по сути, просто *безымянная функция*. В этих выражениях используется символ `=>` (читается как «идет в») для обозначения возвращаемого значения.

1. Скопируйте и вставьте запрос, закомментируйте второй пример и измените запрос:

   ```
   var query = names.Where(name => name.Length > 4);
   ```

   Обратите внимание: синтаксис лямбда-выражения включает в себя все важнейшие части метода `NameLongerThanFour`, но ничего кроме них. Для лямбда-выражения требуется лишь определить:

   - имена входящих параметров;
   - выражение возвращаемого значения.

   Тип входящего параметра `name` предполагается исходя из того, что последовательность содержит строковые значения. При этом для корректной работы метода `Where` тип возвращаемого значения должен быть `bool`, поэтому выражение после символа `=>` также должно возвращать значение типа `bool`.

   Компилятор сделает большую часть работы за нас, поэтому наш код может быть настолько кратким, насколько возможно в принципе.

2. Перезапустите приложение и обратите внимание на то, что его поведение не изменилось.

Сортировка элементов

Для сортировки последовательности используются и такие популярные методы расширения, как `OrderBy` и `ThenBy`.

Методы расширения могут вызываться цепочкой, если предыдущий метод возвращает другую последовательность, то есть тип, реализующий интерфейс `IEnumerable<T>`.

Сортировка элементов с помощью OrderBy

Для изучения сортировки продолжим работу с текущим проектом.

1. Добавьте вызов метода `OrderBy` в конец уже написанного запроса, как показано ниже:

```
var query = names
  .Where(name => name.Length > 4)
  .OrderBy(name => name.Length);
```

Для повышения удобочитаемости форматируйте оператор LINQ таким образом, чтобы вызов каждого метода расширения происходил в отдельной строке.

2. Перезапустите приложение и обратите внимание, что теперь имена отсортированы по увеличению количества символов:

```
Kevin
Creed
Dwight
Angela
Michael
```

Чтобы поместить самое длинное имя в начало списка, воспользуйтесь методом `OrderByDescending`.

Сортировка по нескольким свойствам с помощью метода ThenBy

Нам может потребоваться выполнить сортировку по более чем одному свойству.

1. Добавьте вызов метода `ThenBy` в конце существующего запроса, как показано ниже (выделено полужирным шрифтом):

```
var query = names
  .Where(name => name.Length > 4)
  .OrderBy(name => name.Length)
  .ThenBy(name => name);
```

2. Перезапустите приложение и обратите внимание на небольшое изменение в порядке сортировки. В рамках группы имен с одинаковым количеством букв имена отсортированы по алфавиту по полному значению строки, таким образом, имя `Creed` следует перед именем `Kevin`, а `Angela` — перед `Dwight`:

```
Creed
Kevin
Angela
Dwight
Michael
```

Фильтрация по типу

Метод расширения `Where` отлично работает для фильтрации по значению такой последовательности, как текст и числа. Но что, если последовательность содержит несколько типов и вы хотите отфильтровать по какому-то определенному типу с учетом иерархии наследования?

Представьте, что у вас есть последовательность исключений. Они имеют сложную иерархию, как показано на следующей схеме (рис. 12.4).

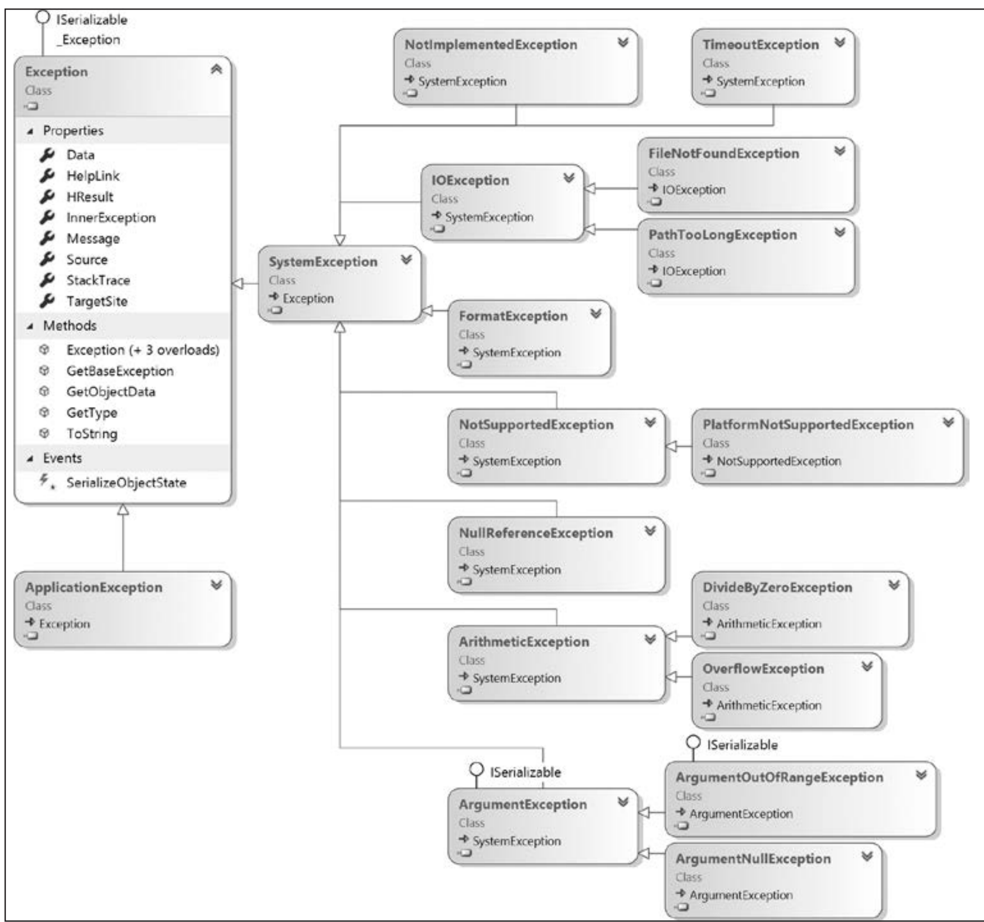


Рис. 12.4. Иерархия наследования исключений

Рассмотрим фильтрацию по типу.

1. В класс `Program` добавьте метод `LinqWithArrayOfExceptions`, который определяет массив объектов, наследуемый от класса `Exception`:

```
static void LinqWithArrayOfExceptions()
{
  var errors = new Exception[]
  {
    new ArgumentException(),
    new SystemException(),
    new IndexOutOfRangeException(),
    new InvalidOperationException(),
    new NullReferenceException(),
    new InvalidCastException(),
    new OverflowException(),
    new DivideByZeroException(),
    new ApplicationException()
  };
}
```

2. Добавьте операторы, используя метод расширения `OfType<T>`, чтобы отфильтровать исключения, которые не являются арифметическими, и записать их в консоль:

```
var numberErrors = errors.OfType<ArithmeticException>();

foreach (var error in numberErrors)
{
  WriteLine(error);
}
```

3. В методе `Main` закомментируйте вызов метода `LinqWithArrayOfStrings` и добавьте вызов метода `LinqWithArrayOfExceptions`.
4. Запустите консольное приложение и обратите внимание, что результаты включают только исключения типа `ArithmeticException` или типа, наследуемого от класса `ArithmeticException`:

```
System.OverflowException: Arithmetic operation resulted in an overflow.
System.DivideByZeroException: Attempted to divide by zero.
```

Работа с множествами с помощью LINQ

Множества — одна из наиболее фундаментальных концепций в математике. *Множество* — это коллекция, состоящая из одного или нескольких объектов. *Мультимножество*, или *множество с повторяющимися элементами*, — коллекция из одного или нескольких объектов, которые могут иметь дубликаты. Возможно, вы помните, как проходили в школе диаграммы Венна. Типовые операции с множествами — это *пересечение* и *объединение*.

Создайте проект консольного приложения, определяющий три массива строковых значений для группы учащихся, а затем выполните некоторые типовые операции для множеств и мультимножеств над ними.

1. Создайте проект консольного приложения `LinqWithSets`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.
2. Импортируйте следующие дополнительные пространства имен:

```
using System.Collections.Generic; // для IEnumerable<T>
using System.Linq; // для методов расширения LINQ
```

3. В класс `Program` перед методом `Main` добавьте следующий метод для вывода в консоль последовательности переменных `string` в виде одной строки с запятыми в качестве разделителя и дополнительной строки описания:

```
static void Output(IEnumerable<string> cohort,
  string description = "")
{
  if (!string.IsNullOrEmpty(description))
  {
    WriteLine(description);
  }
  Write(" ");
  WriteLine(string.Join(", ", cohort.ToArray()));
}
```

4. Добавьте в метод `Main` следующие операторы для определения трех массивов имен. Затем выведите их и выполните с их помощью различные операции над множествами:

```
var cohort1 = new string[]
  { "Rachel", "Gareth", "Jonathan", "George" };
var cohort2 = new string[]
  { "Jack", "Stephen", "Daniel", "Jack", "Jared" };
var cohort3 = new string[]
  { "Declan", "Jack", "Jack", "Jasmine", "Conor" };
Output(cohort1, "Cohort 1");
Output(cohort2, "Cohort 2");
Output(cohort3, "Cohort 3");
WriteLine();

Output(cohort2.Distinct(), "cohort2.Distinct():");
WriteLine();
Output(cohort2.Union(cohort3), "cohort2.Union(cohort3):");
WriteLine();
Output(cohort2.Concat(cohort3), "cohort2.Concat(cohort3):");
WriteLine();
Output(cohort2.Intersect(cohort3), "cohort2.Intersect(cohort3):");
WriteLine();
Output(cohort2.Except(cohort3), "cohort2.Except(cohort3):");
WriteLine();
Output(cohort1.Zip(cohort2,(c1, c2) => $"{c1} matched with {c2}"),
  "cohort1.Zip(cohort2):");
```

5. Запустите консольное приложение и проанализируйте результат:

```
Cohort 1
 Rachel, Gareth, Jonathan, George
Cohort 2
 Jack, Stephen, Daniel, Jack, Jared
Cohort 3
 Declan, Jack, Jack, Jasmine, Conor
cohort2.Distinct():
 Jack, Stephen, Daniel, Jared
cohort2.Union(cohort3):
 Jack, Stephen, Daniel, Jared, Declan, Jasmine, Conor
cohort2.Concat(cohort3):
 Jack, Stephen, Daniel, Jack, Jared, Declan, Jack, Jack, Jasmine, Conor
cohort2.Intersect(cohort3):
 Jack
cohort2.Except(cohort3):
 Stephen, Daniel, Jared
cohort1.Zip(cohort2):
 Rachel matched with Jack, Gareth matched with Stephen, Jonathan matched
with Daniel, George matched with Jack
```

При использовании метода `Zip`, если в последовательностях неравное количество элементов, некоторые из них останутся без подходящей пары. Элементы, не имеющие пары, не будут включены в результат.

Использование LINQ с EF Core

Для изучения *проекции* лучше использовать более сложные последовательности, поэтому в нашем следующем проекте мы задействуем образец базы данных `Northwind`.

Создание модели EF Core

Для представления базы данных и таблиц, с которыми мы будем работать, определим модель ядра *Entity Framework (EF)*. Мы определим классы сущностей вручную, чтобы получить полный контроль и предотвратить автоматическое определение связей. Позже вы будете использовать LINQ для соединения двух наборов сущностей.

1. Создайте проект консольного приложения `LinqWithEFCore`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.
2. Отредактирукйте код файла `LinqWithEFCore.csproj`, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="5.0.0" />
  </ItemGroup>
</Project>
```

3. На панели TERMINAL (Терминал) загрузите указанный пакет и скомпилируйте текущий проект:

```
dotnet build
```

4. Скопируйте файл `Northwind.sql` в папку LinqWithEFCore, а затем, используя TERMINAL (Терминал), создайте базу данных `Northwind`, выполнив следующую команду:

```
sqlite3 Northwind.db -init Northwind.sql
```

5. Наберитесь терпения, так как работа этой команды может занять некоторое время для создания структуры базы данных:

```
-- Loading resources from Northwind.sql
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
sqlite>
```

6. Чтобы выйти из командного режима SQLite, нажмите сочетание клавиш Ctrl+D в macOS или Ctrl+C в Windows.
7. Добавьте в проект три файла классов `Northwind.cs`, `Category.cs` и `Product.cs`.
8. Измените файл класса `Northwind.cs` следующим образом:

```
using Microsoft.EntityFrameworkCore;

namespace Packt.Shared
{
  // управляет подключением к базе данных
  public class Northwind : DbContext
  {
    // эти свойства сопоставляются с таблицами в базе данных
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(
      DbContextOptionsBuilder optionsBuilder)
```

```
      {
        string path = System.IO.Path.Combine(
          System.Environment.CurrentDirectory, "Northwind.db");
        optionsBuilder.UseSqlite($"Filename={path}");
      }

      protected override void OnModelCreating(
        ModelBuilder modelBuilder)
      {
        modelBuilder.Entity<Product>()
          .Property(product => product.UnitPrice)
          .HasConversion<double>();
      }
    }
  }
```

9. Измените файл `Category.cs` следующим образом:

```
using System.ComponentModel.DataAnnotations;

namespace Packt.Shared
{
  public class Category
  {
    public int CategoryID { get; set; }
    [Required]
    [StringLength(15)]
    public string CategoryName { get; set; }
    public string Description { get; set; }
  }
}
```

10. Измените файл класса с именем `Product.cs` следующим образом:

```
using System.ComponentModel.DataAnnotations;
namespace Packt.Shared
{
  public class Product
  {
    public int ProductID { get; set; }
    [Required]
    [StringLength(40)]
    public string ProductName { get; set; }
    public int? SupplierID { get; set; }
    public int? CategoryID { get; set; }
    [StringLength(20)]
    public string QuantityPerUnit { get; set; }
    public decimal? UnitPrice { get; set; }
    public short? UnitsInStock { get; set; }
    public short? UnitsOnOrder { get; set; }
    public short? ReorderLevel { get; set; }
```

```
    public bool Discontinued { get; set; }
  }
}
```

Фильтрация и сортировка последовательностей

Добавим операторы для фильтрации и сортировки последовательностей строк из таблиц.

1. Откройте файл `Program.cs` и импортируйте следующие пространства имен:

```
using static System.Console;
using Packt.Shared;
using Microsoft.EntityFrameworkCore;
using System.Linq;
```

2. Создайте метод для фильтрации и сортировки продуктов:

```
static void FilterAndSort()
{
  using (var db = new Northwind())
  {
    var query = db.Products
      // запрос DbSet<Product>
      .Where(product => product.UnitPrice < 10M)
      // запрос IQueryable<Product>
      .OrderByDescending(product => product.UnitPrice);

    WriteLine("Products that cost less than $10:");
    foreach (var item in query)
    {
      WriteLine("{0}: {1} costs {2:$#,##0.00}",
      item.ProductID, item.ProductName, item.UnitPrice);
    }
    WriteLine();
  }
}
```

Метод `DbSet<T>` реализует интерфейс `IQueryable<T>`, в свою очередь реализующий интерфейс `IEnumerable<T>`, поэтому LINQ можно использовать для запроса и управления коллекциями сущностей в моделях, созданных для EF Core.

Вы, возможно, также заметили, что последовательности реализуют интерфейсы `IQueryable<T>` и `IOrderedQueryable<T>` вместо интерфейсов `IEnumerable<T>` и `IOrderedEnumerable<T>`.

Это показывает, что мы используем поставщик данных LINQ, который применяет отложенное исполнение и создает запрос в памяти с помощью деревьев выражений. Они представляют собой код в древовидной структуре данных и позволяют

создавать динамические запросы, что полезно для построения запросов LINQ для внешних поставщиков данных, таких как SQLite.

Более подробную информацию о деревьях выражений можно получить на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/expression-trees/.

Запрос LINQ будет преобразован в другой язык запросов, такой как SQL. Перечисление запросов с помощью оператора `foreach` или вызов метода наподобие `ToArray` приведет к незамедлительному исполнению запроса.

3. В методе `Main` вызовите метод `FilterAndSort`.
4. Запустите консольное приложение и проанализируйте результат:

```
Products that cost less than $10:
41: Jack's New England Clam Chowder costs $9.65
45: Rogede sild costs $9.50
47: Zaanse koeken costs $9.50
19: Teatime Chocolate Biscuits costs $9.20
23: Tunnbröd costs $9.00
75: Rhönbräu Klosterbier costs $7.75
54: Tourtière costs $7.45
52: Filo Mix costs $7.00
13: Konbu costs $6.00
24: Guaraná Fantástica costs $4.50
33: Geitost costs $2.50
```

Данный запрос распечатывает нужную нам информацию, однако делает это настолько неэффективно, что возвращает все столбцы таблицы `Products` вместо трех необходимых нам столбцов. Это эквивалентно следующему оператору SQL:

```
SELECT * FROM Products;
```

В главе 11 вы научились логировать команды SQL, выполненные для SQLite.

Проецирование последовательностей в новые типы

Прежде чем мы рассмотрим проекцию, нам необходимо вспомнить синтаксис инициализации объекта. Имея определенный класс, вы можете создать экземпляр объекта, используя оператор `new`, имя класса и фигурные скобки, чтобы установить начальные значения для полей и свойств:

```
var alice = new Person
{
  Name = "Alice Jones",
  DateOfBirth = new DateTime(1998, 3, 7)
};
```

C# 3.0 и более поздние версии позволяют создавать экземпляры *анонимных типов*, как показано ниже:

```
var anonymouslyTypedObject = new
{
  Name = "Alice Jones",
  DateOfBirth = new DateTime(1998, 3, 7)
};
```

Хотя мы не указали имя типа, компилятор может вывести анонимный тип из установки двух свойств `Name` и `DateOfBirth`. Эта возможность особенно полезна при написании запросов LINQ для проецирования существующего типа в новый, не прибегая к явному определению последнего. Поскольку тип — анонимный, он может работать только с локальными переменными, объявленными с помощью оператора `var`.

Добавим вызов метода `Select`, чтобы сделать команду SQL, исполняемую для таблицы базы данных, более эффективной, проецируя экземпляры класса `Product` в экземпляры нового анонимного типа только с тремя свойствами.

1. В методе `Main` отредактируйте запрос LINQ для использования метода `Select` и возврата лишь необходимых нам трех свойств (столбцов таблицы), как показано ниже (выделено полужирным шрифтом):

```
var query = db.Products
  // запрос DbSet<Product>
  .Where(product => product.UnitPrice < 10M)
  // запрос IQueryable<Product>
  .OrderByDescending(product => product.UnitPrice)
  // query is now an IOrderedQueryable<Product>
  .Select(product => new // анонимный тип
  {
    product.ProductID,
    product.ProductName,
    product.UnitPrice
  });
```

2. Запустите консольное приложение и убедитесь в том, что его вывод не изменился.

Объединение и группировка

Существует два метода расширения для объединения и группировки:

- `Join` принимает четыре параметра: последовательность, с которой требуется объединиться; свойство или свойства *левой* последовательности, по которым нужно найти соответствие; свойство или свойства *правой* последовательности, по которым нужно найти соответствие, и проекцию;

- GroupJoin принимает те же параметры, только сочетает совпадения в групповой объект, где Key используется для совпавшего значения, а интерфейс IEnumerable<T> — для нескольких совпадений.

Рассмотрим данные методы на примере работы с двумя таблицами: Categories и Products.

1. Создайте метод для выбора категорий и товаров, объедините их и выведите, как показано ниже:

```
static void JoinCategoriesAndProducts()
{
  using (var db = new Northwind())
  {
    // присоединяем каждый товар к своей категории,
    // чтобы вернуть 77 совпадений
    var queryJoin = db.Categories.Join(
      inner: db.Products,
      outerKeySelector: category => category.CategoryID,
      innerKeySelector: product => product.CategoryID,
      resultSelector: (c, p) =>
        new { c.CategoryName, p.ProductName, p.ProductID });

    foreach (var item in queryJoin)
    {
      WriteLine("{0}: {1} is in {2}.",
        arg0: item.ProductID,
        arg1: item.ProductName,
        arg2: item.CategoryName);
    }
  }
}
```

   Объединяются две последовательности: outer и inner. В предыдущем примере categories — внешняя последовательность, а products — внутренняя.

2. В методе Main закомментируйте вызов метода FilterAndSort и вызовите метод JoinCategoriesAndProducts.

3. Запустите консольное приложение и проанализируйте результат. Обратите внимание: каждый из 77 товаров выводится в отдельной строке (отредактированных, чтобы включить только первые десять элементов).

```
1: Chai is in Beverages.
2: Chang is in Beverages.
3: Aniseed Syrup is in Condiments.
4: Chef Anton's Cajun Seasoning is in Condiments.
5: Chef Anton's Gumbo Mix is in Condiments.
6: Grandma's Boysenberry Spread is in Condiments.
```

```
7: Uncle Bob's Organic Dried Pears is in Produce.
8: Northwoods Cranberry Sauce is in Condiments.
9: Mishi Kobe Niku is in Meat/Poultry.
10: Ikura is in Seafood.
```

4. Для сортировки по `CategoryName` в конце существующего запроса вызовите метод `OrderBy`:

```
.OrderBy(cp => cp.CategoryName);
```

5. Перезапустите консольное приложение и проанализируйте результаты. Обратите внимание, что для каждого из 77 товаров имеется отдельная строка и в результатах сначала отображаются все товары в категории `Beverages`, затем в категории `Condiments` и т. д., что показано во фрагменте вывода ниже:

```
1: Chai is in Beverages.
2: Chang is in Beverages.
24: Guaraná Fantástica is in Beverages.
34: Sasquatch Ale is in Beverages.
35: Steeleye Stout is in Beverages.
38: Côte de Blaye is in Beverages.
39: Chartreuse verte is in Beverages.
43: Ipoh Coffee is in Beverages.
67: Laughing Lumberjack Lager is in Beverages.
70: Outback Lager is in Beverages.
75: Rhönbräu Klosterbier is in Beverages.
76: Lakkalikööri is in Beverages.
3: Aniseed Syrup is in Condiments.
4: Chef Anton's Cajun Seasoning is in Condiments.
```

6. Создайте метод для группировки и объединения, отобразите сначала имя группы, а затем все элементы в каждой группе:

```
static void GroupJoinCategoriesAndProducts()
{
  using (var db = new Northwind())
  {
    // группируем все товары по соответствующим категориям,
    // чтобы вернуть восемь совпадений
    var queryGroup = db.Categories.AsEnumerable().GroupJoin(
      inner: db.Products,
      outerKeySelector: category => category.CategoryID,
      innerKeySelector: product => product.CategoryID,
      resultSelector: (c, matchingProducts) => new {
        c.CategoryName,
        Products = matchingProducts.OrderBy(p => p.ProductName)
      });

      foreach (var item in queryGroup)
      {
```

```
        WriteLine("{0} has {1} products.",
            arg0: item.CategoryName,
        arg1: item.Products.Count());

      foreach (var product in item.Products)
      {
        WriteLine($" {product.ProductName}");
      }
    }
  }
}
```

Если мы не вызовем метод `AsEnumerable`, то возникнет исключение времени выполнения:

```
Unhandled exception. System.NotImplementedException: The method or
operation is not implemented.
at Microsoft.EntityFrameworkCore.Relational.Query.Pipeline.
RelationalQueryableMethodTranslatingExpressionVisitor.TranslateGroupJoin(S
hapedQueryExpression outer, ShapedQueryExpression inner, LambdaExpression
outerKeySelector, LambdaExpression innerKeySelector, LambdaExpression
resultSelector)
```

Это связано с тем, что не все методы расширения LINQ можно преобразовать из деревьев выражений в другой синтаксис запроса, такой как SQL. В подобных случаях мы можем преобразовать интерфейс `IQueryable<T>` в `IEnumerable<T>` с помощью вызова метода `AsEnumerable`, при котором в обработке запросов используется LINQ to EF Core только для передачи данных в приложение, а затем применяется LINQ to Objects для выполнения сложной обработки в памяти. Часто этот способ менее эффективен.

7. В методе `Main` закомментируйте предыдущий вызов метода и вызовите метод `GroupJoinCategoriesAndProducts`.
8. Перезапустите консольное приложение, проанализируйте результат. Обратите внимание: все товары в каждой категории отсортированы по алфавиту, как и было задано в запросе, что и показано во фрагменте вывода ниже:

```
Beverages has 12 products.
 Chai
 Chang
 Chartreuse verte
 Côte de Blaye
 Guaraná Fantástica
 Ipoh Coffee
 Lakkalikööri
 Laughing Lumberjack Lager
 Outback Lager
 Rhönbräu Klosterbier
```

```
 Sasquatch Ale
 Steeleye Stout
 Condiments has 12 products.
 Aniseed Syrup
 Chef Anton's Cajun Seasoning
 Chef Anton's Gumbo Mix
...
```

Агрегирование последовательностей

Существуют методы расширения LINQ для выполнения функций агрегирования, например, `Average` и `Sum`. Напишем код, чтобы изучить некоторые из этих методов в действии, извлекая информацию из таблицы `Products`.

1. Создайте метод, показывающий применение методов расширения для агрегирования:

```
static void AggregateProducts()
{
  using (var db = new Northwind())
  {
    WriteLine("{0,-25} {1,10}",
      arg0: "Product count:",
      arg1: db.Products.Count());

    WriteLine("{0,-25} {1,10:$#,##0.00}",
      arg0: "Highest product price:",
      arg1: db.Products.Max(p => p.UnitPrice));

    WriteLine("{0,-25} {1,10:N0}",
      arg0: "Sum of units in stock:",
      arg1: db.Products.Sum(p => p.UnitsInStock));

    WriteLine("{0,-25} {1,10:N0}",
      arg0: "Sum of units on order:",
      arg1: db.Products.Sum(p => p.UnitsOnOrder));

    WriteLine("{0,-25} {1,10:$#,##0.00}",
      arg0: "Average unit price:",
      arg1: db.Products.Average(p => p.UnitPrice));

    WriteLine("{0,-25} {1,10:$#,##0.00}",
      arg0: "Value of units in stock:",
        arg1: db.Products.AsEnumerable()
          .Sum(p => p.UnitPrice * p.UnitsInStock));
  }
}
```

2. В методе `Main` закомментируйте предыдущий метод и вызовите метод `AggregateProducts`.
3. Запустите консольное приложение и проанализируйте результат:

```
Product count:                   77
Highest product price:      $263.50
Sum of units in stock:        3,119
Sum of units on order:          780
Average unit price:          $28.87
Value of units in stock: $74,050.85
```

В Entity Framework Core 3.0 и более поздних версиях операции LINQ, которые не могут быть преобразованы в SQL, больше не выполняются автоматически на стороне клиента, поэтому необходимо принудительно вызвать метод `AsEnumerable`, чтобы форсировать дальнейшую обработку запроса на стороне клиента.

Более подробную информацию об этом существенном изменении можно получить на сайте https://docs.microsoft.com/ru-ru/ef/core/what-is-new/ef-core-3.x/breaking-changes#linq-queries-are-no-longer-evaluated-on-the-client.

Подслащение синтаксиса LINQ с помощью синтаксического сахара

В версии языка C# 3.0 в 2008 году были представлены новые ключевые слова, которые упростили программистам работу с SQL при написании запросов LINQ. Иногда этот *синтаксический сахар* называется *понятным синтаксисом запросов LINQ*.

Функциональность понятного синтаксиса запросов LINQ весьма ограниченна. Она предоставляет только ключевые слова C# для наиболее часто используемых функций LINQ. Для доступа ко всем функциям LINQ следует применять методы расширения. Более подробную информацию, почему этот синтаксис называется понятным, можно получить на сайте https://stackoverflow.com/questions/6229187/linq-why-is-it-called-comprehension-syntax.

Рассмотрим следующий массив значений `string`:

```
var names = new string[] { "Michael", "Pam", "Jim", "Dwight",
  "Angela", "Kevin", "Toby", "Creed" };
```

Для фильтрации и сортировки по имени вы можете использовать *методы расширения* и *лямбда-выражения*:

```
var query = names
  .Where(name => name.Length > 4)
  .OrderBy(name => name.Length)
  .ThenBy(name => name);
```

Или вы можете достичь тех же результатов, используя *понятный синтаксис запросов*:

```
var query = from name in names
  where name.Length > 4
  orderby name.Length, name
  select name;
```

Компилятор самостоятельно выполнит преобразование понятного синтаксиса запросов в методы расширения и лямбда-выражения.

Понятному синтаксису запросов LINQ всегда требуется ключевое слово `select`. Метод расширения `Select` необязателен при использовании методов расширений и лямбда-выражений, поскольку в этом случае неявным образом будет выбран элемент целиком.

Не все методы расширения имеют эквивалентное ключевое слово на языке C#, например методы `Skip` и `Take`, которые обычно используются для постраничного просмотра большого объема данных.

Запрос, использующий `Skip` и `Take`, не может быть написан только с помощью понятного синтаксиса запросов, поэтому мы можем написать запрос, используя исключительно методы расширения:

```
var query = names
  .Where(name => name.Length > 4)
  .Skip(80)
  .Take(10);
```

В качестве альтернативы вы можете заключить понятный синтаксис запроса в скобки и перейти к использованию методов расширения:

```
var query = (from name in names
            where name.Length > 4
            select name)
  .Skip(80)
  .Take(10);
```

Вам следует изучить не только методы расширения и лямбда-выражения, но и понятный синтаксис запросов для написания запросов LINQ, поскольку, скорее всего, вам придется сопровождать код, в котором применяется все вышеперечисленное.

Использование нескольких потоков и параллельного LINQ

По умолчанию для выполнения одного запроса LINQ применяется только один поток. С помощью *параллельного LINQ (PLINQ)* можно получить возможность использования нескольких потоков для выполнения одного запроса LINQ.

Не следует полагать, что применение параллельных потоков улучшит производительность ваших приложений. Всегда измеряйте реальные временные показатели и использование ресурсов.

Разработка приложения с помощью нескольких потоков

Чтобы понять информацию данного подраздела в действии, начнем с написания кода, использующего только один поток для возведения в квадрат 200 миллионов целых чисел. Для измерения изменений в производительности мы будем применять тип `StopWatch`.

Для наблюдения за центральным процессором и использованием его ядра мы задействуем инструменты операционной системы. Это упражнение будет малоэффективным в случае отсутствия нескольких процессоров или хотя бы нескольких ядер!

1. Создайте проект консольного приложения `LinqInParallel`, добавьте его в рабочую область и выберите проект в качестве активного для OmniSharp.
2. Чтобы можно было использовать тип `StopWatch`, импортируйте пространство имен `System.Diagnostics`, а для применения типа `IEnumerable<T>` — пространство имен `System.Collections.Generic`, `System.Linq` для использования LINQ, а также статически импортируйте тип `System.Console`.
3. Добавьте операторы в метод `Main`, чтобы создать секундомер, записывающий временные интервалы, запустить таймер по нажатию клавиши, создать 200 миллионов целых чисел, возвести каждое из них в квадрат, остановить таймер и отобразить количество прошедших миллисекунд, как показано ниже:

```
var watch = new Stopwatch();
Write("Press ENTER to start: ");
ReadLine();
watch.Start();
```

```
IEnumerable<int> numbers = Enumerable.Range(1, 2_000_000_000);

var squares = numbers.Select(number => number * number).ToArray();

watch.Stop();
WriteLine("{0:#,##0} elapsed milliseconds.",
  arg0: watch.ElapsedMilliseconds);
```

4. Запустите консольное приложение, но пока *не нажимайте* клавишу Enter.

Для Windows 10

1. В операционной системе Windows 10 щелкните правой кнопкой мыши на кнопке Start (Пуск) или нажмите сочетание клавиш Ctrl+Alt+Del, а затем выберите пункт Task Manager (Диспетчер задач).
2. В нижней части окна Task Manager (Диспетчер задач) нажмите кнопку More details (Более подробно). В верхней части этого же окна перейдите на вкладку Performance (Производительность).
3. Щелкните правой кнопкой мыши на графике CPU Utilization (Использование ЦП) и выберите пункт контекстного меню Change graph to ▸ Logical processors (Изменить график ▸ Логические процессоры).

Для macOS

1. В операционной системе macOS запустите приложение Activity Monitor (Монитор активности).
2. Выберите View ▸ Update Frequency ▸ Very often (1 sec) (Просмотр ▸ Частота обновления ▸ Очень часто (1 с)).
3. Для просмотра графиков центрального процессора выберите пункт Window ▸ CPU History (Окно ▸ История ЦП).

Для всех операционных систем

1. Расположите окна Task Manager (Диспетчер задач), или окна CPU History (История ЦП), или ваш инструмент Linux и программу Visual Studio Code так, чтобы они были рядом.
2. Дождитесь стабилизации графиков центрального процессора и нажмите клавишу Enter для запуска секундомера и обработки запроса.

   В результате на экране отобразится количество прошедших миллисекунд, как показано в коде ниже и на рис. 12.5:

```
Press ENTER to start.
173,689 elapsed milliseconds.
```

Окно Task Manager (Диспетчер задач) или CPU History (История ЦП) должно показать, что по большей части были использованы только один или два центральных процессора. Другие процессоры могут выполнять одновременно фоновые задачи, такие как очистка памяти от ненужных данных, поэтому графики этих процессоров тоже не будут плоскими. Но можно с уверенностью сказать, что работа не будет поровну распределена между всеми возможными центральными процессорами.

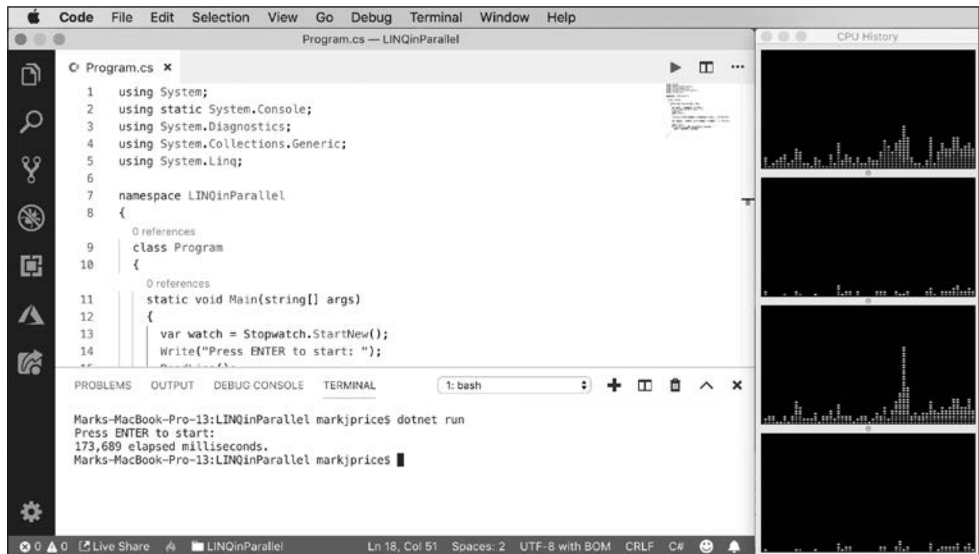


Рис. 12.5. Приложение, использующее один поток

3. В методе `Main` измените запрос так, чтобы осуществить вызов метода расширения `AsParallel`:

```
var squares = numbers.AsParallel()
  .Select(number => number * number).ToArray();
```

4. Повторно запустите приложение.
5. Дождитесь нормализации графиков в окне Task Manager (Диспетчер задач) или CPU History (История ЦП) и нажмите клавишу Enter для запуска секундомера и обработки запроса. В этот раз приложение должно справиться с задачей быстрее (впрочем, ускорение может быть не столь большим, как вам хотелось бы: управление несколькими потоками тоже требует усилий!):

```
Press ENTER to start.
145,904 elapsed milliseconds.
```

6. Окно Task Manager (Диспетчер задач) или CPU History (История ЦП) должно показать, что для выполнения запроса LINQ были использованы все центральные процессоры в равной степени (рис. 12.6).

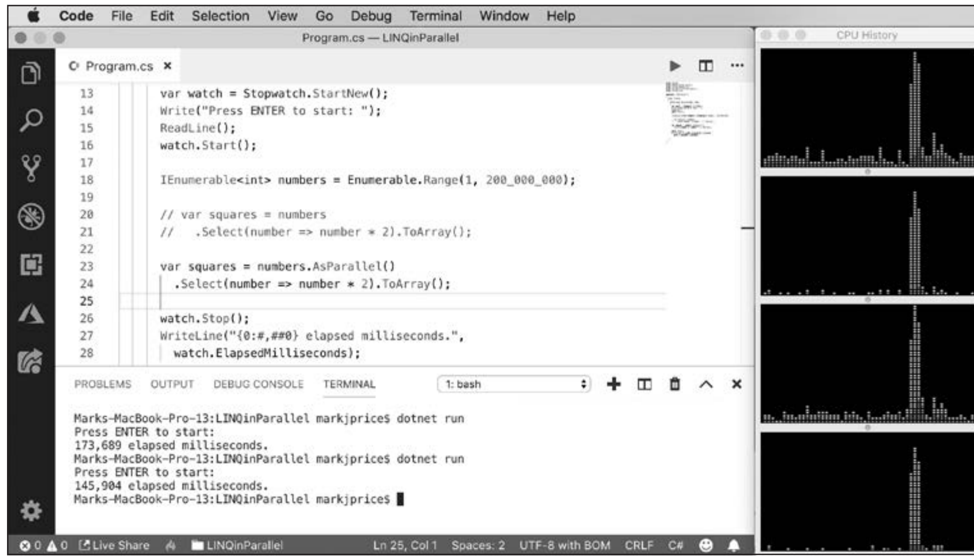


Рис. 12.6. Приложение, использующее несколько потоков

В главе 13 вы более подробно изучите принципы управления несколькими потоками.

Создание собственных методов расширения LINQ

Из главы 6 вы узнали, как создавать пользовательские методы расширения. Для создания методов расширения LINQ вам необходимо лишь расширить тип `IEnumerable<T>`.

Размещайте свои методы расширения в отдельной библиотеке классов, чтобы их можно было легко развернуть в виде сборки или NuGet-пакета.

В качестве примера рассмотрим метод расширения `Average`. Любой школьник скажет вам, что среднее значение может означать одно из трех понятий:

- *среднее арифметическое* — сумма чисел, деленная на их количество;
- *мода* — наиболее часто встречающееся число;
- *медиана* — число в середине упорядоченной последовательности чисел.

Метод расширения `Average` от Microsoft позволяет вычислить среднее арифметическое. Может потребоваться определить собственные методы расширения для вычисления моды и медианы.

1. Добавьте в проект `LinqWithEFCore` файл `MyLinqExtensions.cs`.
2. Отредактируйте класс, чтобы его код выглядел так:

```
using System.Collections.Generic;

namespace System.Linq // расширить пространство имен Microsoft
{
  public static class MyLinqExtensions
  {
    // это цепной метод расширения LINQ
    public static IEnumerable<T> ProcessSequence<T>(
      this IEnumerable<T> sequence)
    {
      // здесь можно выполнить вычисления
      return sequence;
    }

    // это скалярные методы расширения LINQ
    public static int? Median(this IEnumerable<int?> sequence)
    {
      var ordered = sequence.OrderBy(item => item);
      int middlePosition = ordered.Count() / 2;
      return ordered.ElementAt(middlePosition);
    }

    public static int? Median<T>(
      this IEnumerable<T> sequence, Func<T, int?> selector)
    {
      return sequence.Select(selector).Median();
    }

    public static decimal? Median(
      this IEnumerable<decimal?> sequence)
    {
      var ordered = sequence.OrderBy(item => item);
      int middlePosition = ordered.Count() / 2;
      return ordered.ElementAt(middlePosition);
    }

    public static decimal? Median<T>(
      this IEnumerable<T> sequence, Func<T, decimal?> selector)
    {
      return sequence.Select(selector).Median();
    }

    public static int? Mode(this IEnumerable<int?> sequence)
    {
```

```
      var grouped = sequence.GroupBy(item => item);
      var orderedGroups = grouped.OrderByDescending(
        group => group.Count());
      return orderedGroups.FirstOrDefault().Key;
    }

    public static int? Mode<T>(
      this IEnumerable<T> sequence, Func<T, int?> selector)
    {
      return sequence.Select(selector).Mode();
    }

    public static decimal? Mode(
      this IEnumerable<decimal?> sequence)
    {
      var grouped = sequence.GroupBy(item => item);
      var orderedGroups = grouped.OrderByDescending(
        group => group.Count());
      return orderedGroups.FirstOrDefault().Key;
    }

    public static decimal? Mode<T>(
      this IEnumerable<T> sequence, Func<T, decimal?> selector)
    {
      return sequence.Select(selector).Mode();
    }
  }
}
```

Если этот класс находился в отдельной библиотеке классов, то использование ваших методов расширения LINQ потребует лишь сослаться на сборку библиотеки класса, поскольку пространство имен `System.Linq`, как правило, уже импортировано.

3. Для класса `Products` в файле `Program.cs` в методе `FilterAndSort` измените запрос LINQ, чтобы вызвать пользовательский цепной метод расширения, как показано ниже (выделено полужирным шрифтом):

```
var query = db.Products
  // запрос DbSet<Product>
  .ProcessSequence()
  .Where(product => product.UnitPrice < 10M)
  // запрос IQueryable<Product>
  .OrderByDescending(product => product.UnitPrice)
  // запрос IOrderedQueryable<Product>
  .Select(product => new // анонимный тип
  {
    product.ProductID,
    product.ProductName,
    product.UnitPrice
  });
```

4. В методе `Main` закомментируйте метод `FilterAndSort` и вызовы других методов.
5. Запустите консольное приложение и обратите внимание, что увидите тот же самый программный вывод, что и раньше, поскольку ваш метод не изменяет последовательность. Но зато теперь вы знаете, как расширять LINQ с помощью созданной вами функциональности.
6. Добавьте метод, выводящий среднее арифметическое, медиану и моду для `UnitsInStock` и `UnitPrice`, используя собственные методы расширения и встроенный метод расширения `Average`:

```
static void CustomExtensionMethods()
{
  using (var db = new Northwind())
  {
    WriteLine("Mean units in stock: {0:N0}",
      db.Products.Average(p => p.UnitsInStock));
    WriteLine("Mean unit price: {0:$#,##0.00}",
      db.Products.Average(p => p.UnitPrice));
    WriteLine("Median units in stock: {0:N0}",
      db.Products.Median(p => p.UnitsInStock));
    WriteLine("Median unit price: {0:$#,##0.00}",
      db.Products.Median(p => p.UnitPrice));
    WriteLine("Mode units in stock: {0:N0}",
      db.Products.Mode(p => p.UnitsInStock));
    WriteLine("Mode unit price: {0:$#,##0.00}",
      db.Products.Mode(p => p.UnitPrice));
  }
}
```

7. В методе `Main` закомментируйте все предыдущие вызовы методов и вызовите метод `CustomExtensionMethods`.
8. Запустите консольное приложение и проанализируйте результат:

```
Mean units in stock: 41
Mean unit price: $28.87
Median units in stock: 26
Median unit price: $19.50
Mode units in stock: 0
Mode unit price: $18.00
```

В наличии четыре товара с ценой 18 долларов за единицу. Также в наличии пять товаров, количество единиц которых составляет 0 штук.

Работа с LINQ to XML

LINQ to XML — это поставщик данных, позволяющий создавать запросы и управлять XML.

Генерация XML с помощью LINQ to XML

Создадим метод для преобразования таблицы `Products` в XML.

1. В файле `Program.cs` импортируйте пространство имен `System.Xml.Linq`.
2. Создайте метод для вывода товаров в формате XML:

```
static void OutputProductsAsXml()
{
  using (var db = new Northwind())
  {
    var productsForXml = db.Products.ToArray();
    var xml = new XElement("products",
      from p in productsForXml
      select new XElement("product",
        new XAttribute("id", p.ProductID),
        new XAttribute("price", p.UnitPrice),
        new XElement("name", p.ProductName)));

    WriteLine(xml.ToString());
  }
}
```

3. В методе `Main` закомментируйте предыдущий вызов метода и вызовите метод `OutputProductsAsXml`.
4. Запустите консольное приложение и проанализируйте результат. Обратите внимание, что сгенерированный код XML совпадает с элементами и атрибутами, декларативно описанными с помощью выражений LINQ to XML в предыдущем листинге:

```
<products>
  <product id="1" price="18">
    <name>Chai</name>
  </product>
  <product id="2" price="19">
    <name>Chang</name>
  </product>
...
```

Чтение XML с помощью LINQ to XML

Вы можете воспользоваться технологией LINQ to XML, чтобы легко создавать запросы к файлам XML.

1. В проект `LinqWithEFCore` добавьте файл `settings.xml`.
2. Измените его содержимое следующим образом:

```
<?xml version="1.0" encoding="utf-8" ?>
<appSettings>
  <add key="color" value="red" />
  <add key="size" value="large" />
  <add key="price" value="23.99" />
</appSettings>
```

3. Создайте метод для выполнения следующих задач:
   - загрузить файл XML;
   - воспользоваться LINQ to XML для поиска элемента `appSettings` и его потомков с именем `add`;
   - проецировать XML-код в массив анонимного типа со свойствами `Key` и `Value`;
   - перебрать массив для отображения результатов.

```
static void ProcessSettings()
{
  XDocument doc = XDocument.Load("settings.xml");
  var appSettings = doc.Descendants("appSettings")
    .Descendants("add")
    .Select(node => new
    {
      Key = node.Attribute("key").Value,
      Value = node.Attribute("value").Value
    }).ToArray();

  foreach (var item in appSettings)
  {
    WriteLine($"{item.Key}: {item.Value}");
  }
}
```

4. В методе `Main` закомментируйте предыдущий вызов метода и вызовите метод `ProcessSettings`.
5. Запустите консольное приложение и проанализируйте результат:

```
color: red
size: large
price: 23.99
```

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 12.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Каковы две обязательные составные части LINQ?
2. Какой метод расширения LINQ вы использовали бы для возврата из типа подмножества его свойств?
3. Какой метод расширения LINQ вы применили бы для фильтрации последовательности?
4. Перечислите пять методов расширения LINQ, выполняющих агрегацию данных.
5. Чем различаются методы расширения `Select` и `SelectMany`?
6. В чем разница между интерфейсом `IEnumerable<T>` и `IQueryable<T>` и как вы переключаетесь между ними?
7. Что представляет собой последний параметр типа в делегатах-дженериках `Func`?
8. В чем преимущество метода расширения LINQ, который заканчивается оператором `OrDefault`?
9. Почему понятный синтаксис запросов необязателен?
10. Каким образом вы можете создать собственные методы расширения LINQ?

Упражнение 12.2. Создание запросов LINQ

Создайте консольное приложение `Exercise02`, которое запрашивает у пользователя название города, а затем перечисляет название компаний — клиентов Northwind в заданном городе.

```
Enter the name of a city: London
There are 6 customers in London:
Around the Horn
B's Beverages
Consolidated Holdings
Eastern Connection
North/South
Seven Seas Imports
```

Усовершенствуйте приложение так, чтобы оно отображало список всех уникальных городов, в которых уже находятся клиенты, прежде чем пользователь введет название предпочитаемого города:

```
Aachen, Albuquerque, Anchorage, Århus, Barcelona, Barquisimeto, Bergamo, Berlin,
Bern, Boise, Bräcke, Brandenburg, Bruxelles, Buenos Aires, Butte, Campinas,
Caracas, Charleroi, Cork, Cowes, Cunewalde, Elgin, Eugene, Frankfurt a.M.,
```

```
Genève, Graz, Helsinki, I. de Margarita, Kirkland, Kobenhavn, Köln, Lander,
Leipzig, Lille, Lisboa, London, Luleå, Lyon, Madrid, Mannheim, Marseille,
México D.F., Montréal, München, Münster, Nantes, Oulu, Paris, Portland,
Reggio Emilia, Reims, Resende, Rio de Janeiro, Salzburg, San Cristóbal,
San Francisco, Sao Paulo, Seattle, Sevilla, Stavern, Strasbourg, Stuttgart,
Torino, Toulouse, Tsawassen, Vancouver, Versailles, Walla Walla, Warszawa
```

Упражнение 12.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- LINQ в C#: https://docs.microsoft.com/ru-ru/dotnet/csharp/linq/linq-in-csharp;
- 101 пример использования LINQ: https://docs.microsoft.com/ru-ru/samples/dotnet/try-samples/101-linq-samples/;
- параллельный LINQ (PLINQ): https://docs.microsoft.com/ru-ru/dotnet/standard/parallel-programming/introduction-to-plinq;
- LINQ to XML (C#): https://docs.microsoft.com/en-gb/dotnet/csharp/programming-guide/concepts/linq/linq-to-xml-overview;
- LINQPad: https://www.linqpad.net/.

Резюме

Вы изучили, как писать запросы LINQ, предназначенные для выбора, проецирования, фильтрации, сортировки, объединения и группировки данных в различных форматах, в том числе XML, что в общем является каждодневными задачами.

В следующей главе вы будете использовать тип `Task` для улучшения производительности ваших приложений.

13 Улучшение производительности и масштабируемости с помощью многозадачности

Данная глава посвящена способам одновременного выполнения нескольких действий в целях повышения производительности, масштабируемости и эффективности работы конечных пользователей.

В этой главе:

- процессы, потоки и задачи;
- мониторинг производительности и использования ресурсов;
- асинхронное выполнение задач;
- синхронизация доступа к общим ресурсам;
- ключевые слова `async` и `await`.

Процессы, потоки и задачи

Под *процессом* можно принимать ресурсы, такие как память и потоки, выделяемые под каждое из запущенных вами консольных приложений. *Поток* выполняет написанный вами код, оператор за оператором. По умолчанию каждый процесс имеет только один поток, что может привести к проблемам в случае необходимости выполнять несколько *задач* одновременно. Потоки также отвечают за хранение информации об аутентифицированном в данный момент пользователе и правилах интернационализации, которые должны соблюдаться для текущего языка и региона.

Windows и большинство других современных операционных систем используют режим *вытесняющей многозадачности*, которая имитирует параллельное выполнение задач. Данный режим делит процессорное время между потоками, выделяя *«интервал времени»* для каждого потока один за другим. Текущий поток приостанавливается, когда заканчивается его «интервал времени». Затем процессор выделяет другому потоку еще один «интервал времени».

Переключаясь с одного потока на другой, операционная система Windows сохраняет контекст потока и перезагружает ранее сохраненный контекст следующего потока в очереди потоков. На это требуются время и ресурсы.

Потоки имеют свойства `Priority` и `ThreadState`. Кроме того, существует класс `ThreadPool`, предназначенный для управления пулом фоновых рабочих потоков, как показано на следующей схеме (рис. 13.1).

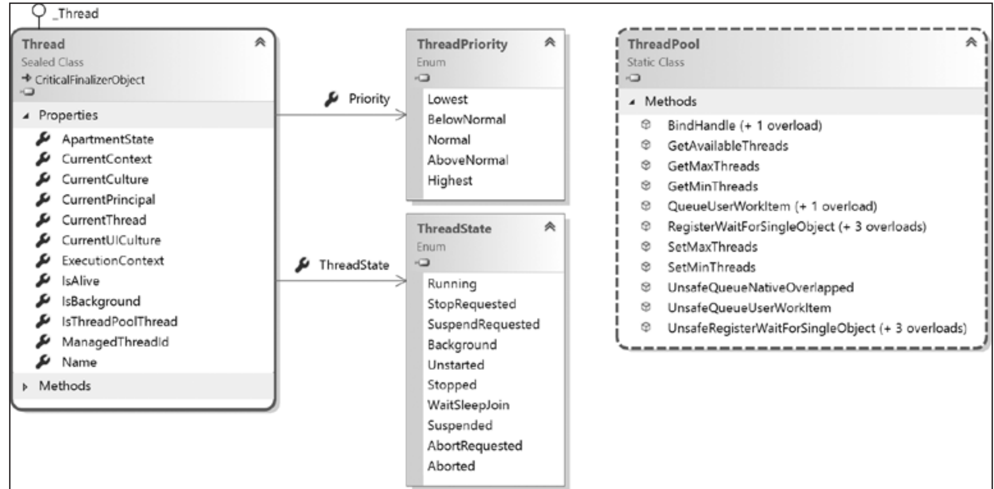


Рис. 13.1. Класс Thread и связанные типы

Если вы как разработчик имеете дело со сложными действиями, которые должны быть выполнены вашим кодом, и хотите получить полный контроль над ними, то можете создавать отдельные экземпляры класса `Thread` и управлять ими. При наличии одного основного потока и нескольких небольших действий, которые можно выполнять в фоновом режиме, вы можете добавить в очередь экземпляры делегатов, указывающие на эти фрагменты, реализованные в виде методов, и они будут автоматически распределены по потокам с помощью пула потоков.

Дополнительную информацию о пуле потоков можно получить на сайте https://docs.microsoft.com/ru-ru/dotnet/standard/threading/the-managed-thread-pool.

Потокам, возможно, придется конкурировать и ждать доступа к общим ресурсам, таким как переменные, файлы и объекты баз данных.

В зависимости от задачи удвоение количества потоков (рабочих) не уменьшает вдвое количество секунд, которое будет затрачено на выполнение задачи. Фактически это может даже *увеличить* длительность выполнения задачи (рис. 13.2).

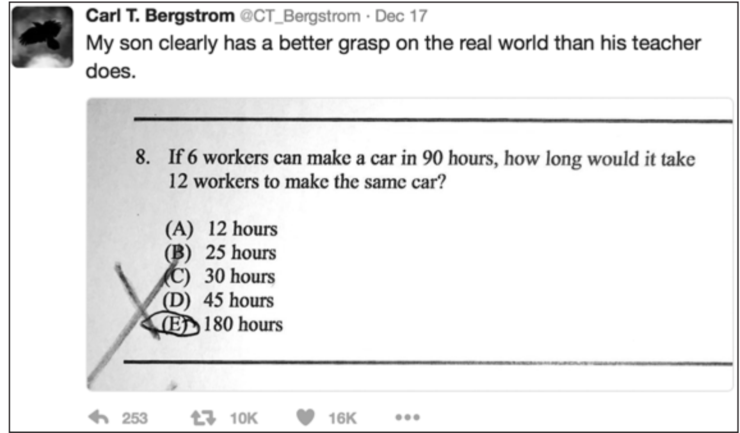


Рис. 13.2. Удвоение количества рабочих не обязательно вдвое сокращает время, необходимое для выполнения задачи

Никогда не полагайтесь на то, что увеличение количества потоков повысит производительность! Выполните тесты производительности сначала базового кода без реализации нескольких потоков, а затем — кода с несколькими потоками. Выполняйте тесты производительности в промежуточной среде, максимально приближенной к рабочей.

Мониторинг производительности и использования ресурсов

Прежде чем мы сможем улучшить производительность любого кода, мы должны иметь возможность проводить мониторинг его скорости и эффективности работы, чтобы получить базовые данные, на основании которых можем измерять улучшения.

Оценка эффективности типов

Какой тип лучше всего использовать для того или иного сценария? Для ответа на этот вопрос нам необходимо тщательно обдумать, что для нас *лучше всего*. Для этого нужно рассмотреть следующие факторы:

- *функциональность* — для принятия решения проверяется, представляет ли данный тип необходимую вам функциональность;
- *объем памяти* — для принятия решения проверяется, сколько байтов памяти потребляет данный тип;

- *производительность* — для принятия решения проверяется, насколько быстро работает данный тип;
- *потребности в будущем* — этот фактор зависит от возможных изменений требований и возможности поддержки.

Будут сценарии, например, при сохранении чисел, когда несколько типов имеют одинаковую функциональность, поэтому, прежде чем выбирать, следует сопоставить расход памяти и производительность.

Для сохранения миллионов чисел наиболее подходящим типом будет тот, который занимает меньше всего места в памяти. Если же необходимо сохранить только несколько чисел, но выполнить с ними большое количество операций, то лучше выбрать тип, который быстрее всего работает на конкретном ЦПУ.

Ранее упоминалась функция `sizeof()`, предназначенная для отображения количества байтов, занимаемых в памяти одним экземпляром конкретного типа. При сохранении большого количества значений в более сложных структурах данных, таких как массивы и списки, измерение использования памяти требует способа получше.

В Интернете и книгах можно прочитать множество рекомендаций, но единственный способ узнать, какой тип лучше подходит для вашей программы, — это сравнить их самостоятельно.

В следующем подразделе вы научитесь писать код для мониторинга действительных требований к памяти и производительности при использовании различных типов.

В настоящее время оптимальным выбором может стать переменная `short`, но, вероятно, будет лучше задействовать переменную `int`, хотя она занимает в два раза больше места в памяти, ведь возможно, что в дальнейшем потребуется хранить более широкий диапазон значений.

Следует принять во внимание еще один показатель — обслуживание. Данная мера показывает, сколько усилий потребуется приложить другому разработчику, чтобы понять и отредактировать ваш код. Если вы прибегаете к неочевидному выбору типа, не объясняя данное действие с помощью комментария, то это может запутать разработчика, открывшего ваш код, чтобы исправить ошибку или добавить какую-либо функциональность.

Мониторинг производительности и использования памяти

В пространстве имен `System.Diagnostics` реализовано большое количество полезных типов для мониторинга вашего кода. В первую очередь следует рассмотреть тип `Stopwatch`.

1. Создайте в папке Code папку Chapter13 с двумя подпапками MonitoringLib и MonitoringApp.
2. В программе Visual Studio Code сохраните рабочую область как `Chapter13.code-workspace`.
3. Добавьте в рабочую область папку MonitoringLib, откройте для нее новую панель TERMINAL (Терминал) и создайте новый проект библиотеки классов, как показано в следующей команде:

```
dotnet new classlib
```

4. Добавьте в рабочую область папку MonitoringApp, откройте для нее новую панель TERMINAL (Терминал) и создайте новый проект консольного приложения, как показано в следующей команде:

```
dotnet new console
```

5. В проекте `MonitoringLib` переименуйте файл `Class1.cs` на `Recorder.cs`.
6. В проекте `MonitoringApp` найдите и откройте файл `MonitoringApp.csproj` и добавьте ссылку на библиотеку `MonitoringLib`, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference
      Include="..\MonitoringLib\MonitoringLib.csproj" />
  </ItemGroup>
</Project>
```

7. На панели TERMINAL (Терминал) скомпилируйте проекты, как показано в следующей команде:

```
dotnet build
```

Реализация класса Recorder

Тип `Stopwatch` содержит несколько полезных членов, как показано в табл. 13.1.

Таблица 13.1

Член	**Описание**
Метод Restart	Обнуляет затраченное время и начинает измерение затраченного времени
Метод Stop	Останавливает измерение затраченного времени

Продолжение ⇗

Таблица 13.1 *(продолжение)*

Член	**Описание**
Свойство Elapsed	Затраченное время, сохраненное в виде структуры TimeSpan (например, часы:минуты:секунды)
Свойство ElapsedMilliseconds	Затраченное время в миллисекундах, хранящееся как целочисленное значение long

Тип `Process` содержит несколько полезных членов, перечисленных в табл. 13.2.

Таблица 13.2

Член	**Описание**
VirtualMemorySize64	Отображает объем виртуальной памяти в байтах, выделенной для процесса
WorkingSet64	Отображает объем физической памяти в байтах, выделенной для процесса

Для реализации класса `Recorder` мы будем использовать классы `Stopwatch` и `Process`.

1. Откройте файл `Recorder.cs` и измените его содержимое, чтобы задействовать экземпляр класса `Stopwatch` в целях записи времени и текущий экземпляр класса `Process` для записи использованной памяти:

```
using System;
using System.Diagnostics;
using static System.Console;
using static System.Diagnostics.Process;

namespace Packt.Shared
{
  public static class Recorder
  {
    static Stopwatch timer = new Stopwatch();

    static long bytesPhysicalBefore = 0;
    static long bytesVirtualBefore = 0;

    public static void Start()
    {
      // очистка памяти, на которую больше нет ссылок,
      // но которая еще не освобождена
      GC.Collect();
      GC.WaitForPendingFinalizers();
      GC.Collect();
      // сохранение текущего использования физической и виртуальной памяти
      bytesPhysicalBefore = GetCurrentProcess().WorkingSet64;
      bytesVirtualBefore = GetCurrentProcess().VirtualMemorySize64;
```

```
      timer.Restart();
    }
    public static void Stop()
    {
      timer.Stop();
      long bytesPhysicalAfter = GetCurrentProcess().WorkingSet64;
      long bytesVirtualAfter =
        GetCurrentProcess().VirtualMemorySize64;
      WriteLine("{0:N0} physical bytes used.",
        bytesPhysicalAfter - bytesPhysicalBefore);
      WriteLine("{0:N0} virtual bytes used.",
        bytesVirtualAfter - bytesVirtualBefore);
      WriteLine("{0} time span ellapsed.", timer.Elapsed);
      WriteLine("{0:N0} total milliseconds ellapsed.",
        timer.ElapsedMilliseconds);
    }
  }
}
```

В методе `Start` класса `Recorder` используется сборщик мусора (garbage collector, класс `GC`), позволяющий нам гарантировать, что вся выделенная в настоящий момент память будет собрана до записи количества использованной памяти. Это сложная техника, и ее стоит избегать при разработке прикладной программы.

2. В классе `Program` в метод `Main` добавьте операторы для запуска и остановки класса `Recorder` при генерации массива из 10 000 целых чисел:

```
using System;
using System.Linq;
using Packt.Shared;
using static System.Console;

namespace MonitoringApp
{
  class Program
  {
    static void Main(string[] args)
    {
      WriteLine("Processing. Please wait...");
      Recorder.Start();

      // моделирование процесса, требующего ресурсов памяти...
      int[] largeArrayOfInts =
        Enumerable.Range(1, 10_000).ToArray();

      // ...и занимает некоторое время, чтобы завершить
      System.Threading.Thread.Sleep(
        new Random().Next(5, 10) * 1000);

      Recorder.Stop();
    }
  }
}
```

3. Запустите консольное приложение и проанализируйте результат:

```
Processing. Please wait...
655,360 physical bytes used.
536,576 virtual bytes used.
00:00:09.0038702 time span ellapsed.
9,003 total milliseconds ellapsed.
```

Измерение эффективности обработки строк

Теперь, когда вы научились использовать типы `Stopwatch` и `Process` для мониторинга своего кода, мы применим их с целью определить наилучший способ обработки строковых переменных.

1. Закомментируйте предыдущий код в методе `Main`, обернув его символами `/*` и `*/`.
2. Добавьте в метод `Main` следующий код. Он создает массив из 50 000 переменных `int`, а затем конкатенирует их, используя в качестве разделителей запятые, с помощью классов `string` и `StringBuilder`:

```
int[] numbers = Enumerable.Range(1, 50_000).ToArray();

WriteLine("Using string with +");
Recorder.Start();
string s = "";
for (int i = 0; i < numbers.Length; i++)
{
  s += numbers[i] + ", ";
}
Recorder.Stop();

WriteLine("Using StringBuilder");
Recorder.Start();
var builder = new System.Text.StringBuilder();
for (int i = 0; i < numbers.Length; i++)
{
  builder.Append(numbers[i]); builder.Append(", ");
}
Recorder.Stop();
```

3. Запустите консольное приложение и проанализируйте результат:

```
Using string with +
10,883,072 physical bytes used.
1,609,728 virtual bytes used.
00:00:02.6220879 time span ellapsed.
2,622 total milliseconds ellapsed.
Using StringBuilder
4,096 physical bytes used.
```

```
0 virtual bytes used.
00:00:00.0014265 time span ellapsed.
1 total milliseconds ellapsed.
```

Исходя из результатов, мы можем сделать следующие выводы:

- класс `string` вместе с оператором + использовал около 11 Мбайт физической памяти, 1,5 Мбайт виртуальной и занял по времени 2,6 с;
- класс `StringBuilder` использовал 4 Кбайт физической памяти, 0 вирутальной и занял менее 1 мс.

В нашем случае при конкатенации текста класс `StringBuilder` выполняется примерно в 1000 раз быстрее и приблизительно в 10 000 раз эффективнее по затратам ресурсов памяти!

Избегайте использования метода String.Concat и оператора + внутри цикла. Вместо этого для конкатенации переменных, особенно в циклах, применяйте класс StringBuilder.

Теперь, когда вы научились измерять производительность и эффективность ресурсов вашего кода, изучим процессы, потоки и задачи.

Асинхронное выполнение задач

Чтобы понять, каким образом несколько задач могут выполняться одновременно, мы напишем простое консольное приложение, выполняющее три метода: первый занимает три секунды, второй — две, а третий — одну. Для имитации такого процесса мы можем применить метод класса `Thread`, который передаст текущему потоку инструкцию заснуть на заданное количество миллисекунд.

Синхронное выполнение нескольких действий

Прежде чем запускать задачи одновременно, мы будем запускать их синхронно, то есть одну за другой.

1. Создайте в папке Chapter13 консольное приложение `WorkingWithTasks` и выберите проект в качестве активного для OmniSharp.
2. В файле `Program.cs` импортируйте пространство имен для работы с потоками и задачами:

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

```
using System.Diagnostics;
using static System.Console;
```

3. В класс `Program` добавьте три метода:

```
static void MethodA()
{
  WriteLine("Starting Method A...");
  Thread.Sleep(3000); // симуляция трех секунд работы
  WriteLine("Finished Method A.");
}

static void MethodB()
{
  WriteLine("Starting Method B...");
  Thread.Sleep(2000); // симуляция двух секунд работы
  WriteLine("Finished Method B.");
}

static void MethodC()
{
  WriteLine("Starting Method C...");
  Thread.Sleep(1000); // симуляция одной секунды работы
  WriteLine("Finished Method C.");
}
```

4. В метод `Main` добавьте следующие операторы для определения секундомера и вывода истекших миллисекунд:

```
static void Main(string[] args)
{
  var timer = Stopwatch.StartNew();
  WriteLine("Running methods synchronously on one thread.");
  MethodA();
  MethodB();
  MethodC();
  WriteLine($"{timer.ElapsedMilliseconds:#,##0}ms elapsed.");
}
```

5. Запустите консольное приложение и проанализируйте результат. Обратите внимание: при использовании только одного потока выполнение программы займет шесть секунд:

```
Running methods synchronously on one thread.
Starting Method A...
Finished Method A.
Starting Method B...
Finished Method B.
Starting Method C...
Finished Method C.
6,015ms elapsed.
```

Асинхронное выполнение нескольких действий с помощью задач

Класс `Thread` был доступен уже в первой версии .NET и может использоваться для создания и управления новыми потоками, но работать с ним напрямую сложно.

В версии .NET Framework 4.0 в 2010 году был добавлен класс `Task`, представляющий собой оболочку для потока, позволившую упростить создание и управление потоками. Обертывание нескольких потоков в задачи даст нашему коду возможность выполняться асинхронно (в одно и то же время).

Каждая задача имеет свойство `Status`, а свойство `CreationOptions` содержит метод `ContinueWith`, который можно настроить, прибегнув к перечислению `TaskContinuationOptions`, и которым можно управлять с помощью класса `TaskFactory`, как показано на рис. 13.3.

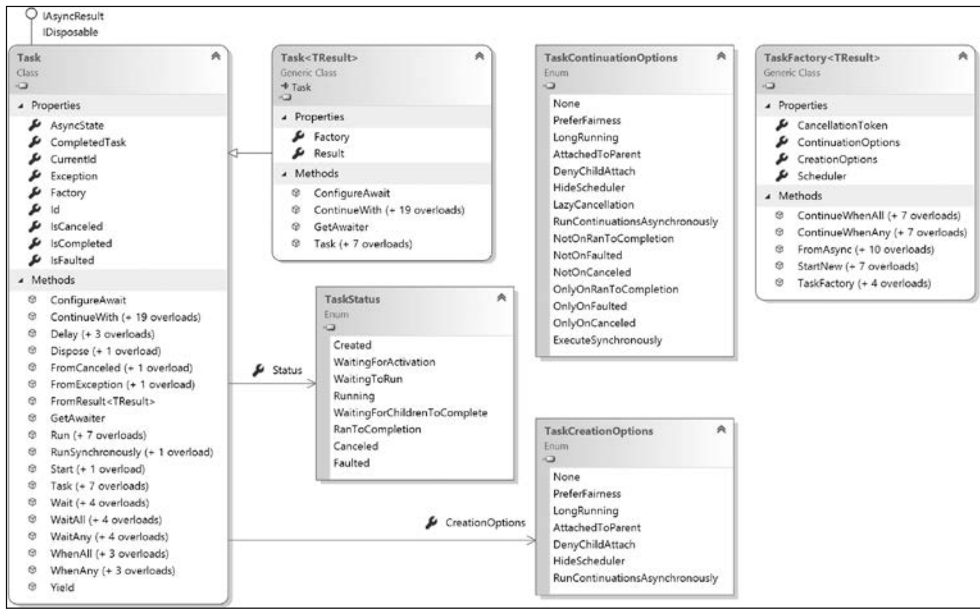


Рис. 13.3. Класс Task и связанные типы

Мы рассмотрим три способа запуска методов с помощью экземпляров класса `Task`. Каждый из них несколько отличается синтаксисом, но все они определяют `Task` и запускают его.

1. Закомментируйте вызовы трех методов и связанного с ними консольного сообщения.

2. Добавьте новые операторы для создания и запуска трех задач, по одной для каждого метода:

```
static void Main(string[] args)
{
  var timer = Stopwatch.StartNew();
  // WriteLine("Running methods synchronously on one thread.");
  // MethodA();
  // MethodB();
  // MethodC();

  WriteLine("Running methods asynchronously on multiple threads.");
  Task taskA = new Task(MethodA);
  taskA.Start();
  Task taskB = Task.Factory.StartNew(MethodB);
  Task taskC = Task.Run(new Action(MethodC));
  WriteLine($"{timer.ElapsedMilliseconds:#,##0}ms elapsed.");
}
```

3. Запустите консольное приложение и проанализируйте результат. Обратите внимание: значение затраченного времени выводится практически сразу, поскольку каждый из трех методов теперь выполняется тремя *новыми* потоками, и, следовательно, исходный поток может записать истекшее время до их завершения:

```
Running methods asynchronously on multiple threads.
Starting Method A...
Starting Method B...
Starting Method C...
3ms elapsed.
```

Возможно даже, что консольное приложение завершится до того, как одна или несколько задач смогут запуститься и записать результат в консоль!

Более подробно о преимуществах и недостатках различных способов запуска задач можно узнать на сайте https://devblogs.microsoft.com/pfxteam/task-factory-startnew-vs-new-task-start/.

Ожидание выполнения задач

Иногда требуется дождаться завершения задачи, прежде чем продолжать работу. Для этого используется метод `Wait` экземпляра класса `Task` или статические методы `WaitAll` и `WaitAny` для массива задач, описанные в табл. 13.3.

Таблица 13.3

Метод	Описание
t.Wait()	Ожидает завершения выполнения задачи с именем t
Task.WaitAny(Task[])	Ожидает завершения выполнения любой задачи в массиве
Task.WaitAll(Task[])	Ожидает завершения выполнения всех задач в массиве

Рассмотрим, каким образом мы можем использовать перечисленные в таблице методы ожидания для решения проблем с нашим консольным приложением.

1. Добавьте операторы в метод `Main` (сразу после кода создания трех задач и перед выводом истекшего времени), чтобы объединить в массив ссылки на три задачи и передать их методу `WaitAll`:

   ```
   Task[] tasks = { taskA, taskB, taskC };
   Task.WaitAll(tasks);
   ```

   Теперь исходный поток будет останавливаться на этом операторе, ожидая завершения всех трех задач, прежде чем вывести значение затраченного времени.

2. Запустите консольное приложение и проанализируйте результат:

   ```
   Running methods asynchronously on multiple threads.
   Starting Method C...
   Starting Method A...
   Starting Method B...
   Finished Method C.
   Finished Method B.
   Finished Method A.
   3,006ms elapsed.
   ```

Три новых потока исполняют свой код одновременно и запускаются в случайном порядке. `MethodC` завершается первым, поскольку длится всего одну секунду, затем `MethodB`, который длится две секунды, и, наконец, `MethodA`, длящийся три секунды.

Тем не менее реально используемый центральный процессор существенно влияет на результат. Именно ЦП выделяет временные интервалы под каждый процесс, чтобы позволить им выполнять свои потоки. Вы не можете контролировать запуск методов.

Задачи продолжения

Если все три задачи могут быть выполнены одновременно, то все, что нам нужно сделать, — это дождаться завершения всех их. Однако часто задача зависит от

результата выполнения другой задачи. Чтобы справиться с данной проблемой, нам нужно определить *задачи продолжения*.

Далее мы создадим методы для имитации вызова веб-сервиса, возвращающего денежную сумму, в дальнейшем применяемую для извлечения из базы данных количества товаров, цена которых превышает данную сумму. Результат, возвращаемый первым методом, должен быть передан в качестве входного параметра второго метода. Чтобы имитировать работу, мы будем использовать класс `Random` для ожидания завершения каждого вызова метода со случайной длительностью в диапазоне от двух до четырех секунд.

1. Добавьте в класс `Program` следующие два метода, которые имитируют вызов веб-сервиса и хранимой процедуры базы данных:

```
static decimal CallWebService()
{
  WriteLine("Starting call to web service...");
  Thread.Sleep((new Random()).Next(2000, 4000));
  WriteLine("Finished call to web service.");
  return 89.99M;
}

static string CallStoredProcedure(decimal amount)
{
  WriteLine("Starting call to stored procedure...");
  Thread.Sleep((new Random()).Next(2000, 4000));
  WriteLine("Finished call to stored procedure.");
  return $"12 products cost more than {amount:C}.";
}
```

2. В методе `Main` закомментируйте предыдущие три задачи, обернув их код символами многострочного комментария `/* */`. Оставьте оператор, который выводит в результате затраченные миллисекунды.
3. Добавьте операторы перед уже существующим оператором, который выводит затраченное время, а затем вызывает метод `ReadLine` для ожидания нажатия пользователем клавиши Enter:

```
WriteLine("Passing the result of one task as an input into another.");
var taskCallWebServiceAndThenStoredProcedure =
  Task.Factory.StartNew(CallWebService)
    .ContinueWith(previousTask =>
      CallStoredProcedure(previousTask.Result));
WriteLine($"Result: {taskCallWebServiceAndThenStoredProcedure.Result}");
```

4. Запустите консольное приложение и проанализируйте результат:

```
Passing the result of one task as an input into another.
Starting call to web service...
Finished call to web service.
```

```
Starting call to stored procedure...
Finished call to stored procedure.
Result: 12 products cost more than £89.99.
5,971ms elapsed
```

Вложенные и дочерние задачи

Помимо определения зависимостей между задачами, выделяют также вложенные и дочерние задачи. *Вложенная задача* — та, которая создается внутри другой задачи. *Дочерняя* — вложенная задача, которая должна быть завершена до завершения ее родительской задачи.

Рассмотрим работу этих задач на примере.

1. Создайте консольное приложение `NestedAndChildTasks`, добавьте его в рабочую область `Chapter13` и выберите проект в качестве активного для OmniSharp.
2. В файле `Program.cs` импортируйте пространства имен для работы с потоками и задачами:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;
using static System.Console;
```

3. Добавьте два метода, один из которых запускает задачу для запуска второго, как показано ниже:

```
static void OuterMethod()
{
  WriteLine("Outer method starting...");
  var inner = Task.Factory.StartNew(InnerMethod);
  WriteLine("Outer method finished.");
}

static void InnerMethod()
{
  WriteLine("Inner method starting...");
  Thread.Sleep(2000);
  WriteLine("Inner method finished.");
}
```

4. В метод `Main` добавьте следующие операторы, чтобы запустить задачу для запуска внешнего метода, и дождитесь ее завершения перед остановкой, как показано ниже:

```
var outer = Task.Factory.StartNew(OuterMethod);
outer.Wait();
WriteLine("Console app is stopping.");
```

5. Запустите консольное приложение и проанализируйте результат:

```
Outer method starting...
Outer method finished.
Console app is stopping.
Inner method starting...
```

Обратите внимание: то, что мы ожидаем завершения внешней задачи, само по себе не заставляет нас ждать завершения внутренней задачи. Связать две задачи необходимо с помощью специального параметра.

6. Измените существующий код, который определяет внутреннюю задачу, чтобы добавить перечисление `TaskCreationOption` с элементом `AttachedToParent` (выделено полужирным шрифтом):

```
var inner = Task.Factory.StartNew(InnerMethod,
  TaskCreationOptions.AttachedToParent);
```

7. Запустите консольное приложение и проанализируйте результат. Обратите внимание, что внутренняя задача должна завершиться раньше внешней:

```
Outer method starting...
Outer method finished.
Inner method starting...
Inner method finished.
Console app is stopping.
```

Метод `OuterMethod` может завершиться раньше метода `InnerMethod`, как показано в результате вывода в консоль, но мы все еще будем ждать завершения обеих задач, что видно по тому, что консольное приложение не закончило работу до завершения как внешней, так и внутренней задачи.

Синхронизация доступа к общим ресурсам

При одновременном выполнении нескольких потоков существует вероятность того, что два и более потока могут одновременно обращаться к одной и той же переменной или другому ресурсу и вызвать тем самым проблему. По этой причине вам следует внимательно изучить, как сделать код *потокобезопасным*.

Простейший механизм обеспечения потоковой безопасности заключается в использовании объектной переменной в качестве *флага* или *светофора*, позволяющего указать, когда на общий ресурс наложена исключительная блокировка.

В книге Уильяма Голдинга *«Повелитель мух»* (William Golding, Lord of the Flies) Хрюша (Piggy) и Ральф (Ralph) находят раковину и используют ее для организа-

ции встречи. Мальчики принимают «правило раковины» и договариваются о том, что говорить имеет право только тот, кто держит в руках рог.

Мне пришлась по душе идея назвать объектную переменную той самой «раковиной» (`conch`). Когда у потока есть «раковина», ни один другой поток не может обращаться к общему ресурсу (-ам), представленному этой раковиной.

Мы рассмотрим ряд типов, которые можно использовать для синхронизации доступа к ресурсам:

- класс `Monitor` предотвращает одновременный доступ нескольких ресурсов к ресурсу в рамках одного процесса;
- класс `Interlocked` управляет простыми числовыми типами на уровне центрального процессора.

Доступ к ресурсу из нескольких потоков

1. Создайте консольное приложение `SynchronizingResourceAccess`, добавьте его в рабочую область `Chapter13` и выберите проект в качестве активного для OmniSharp.
2. Импортируйте пространства имен для работы с потоками и задачами, как показано ниже:

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;
using static System.Console;
```

3. В класс `Program` добавьте операторы для реализации следующих действий:
   - объявите и создайте экземпляр объекта для генерации случайного значения времени ожидания;
   - объявите строковую переменную для хранения сообщения (общий ресурс);
   - объявите два метода, которые добавляют букву `A` или `B` к общей строковой переменной пять раз в цикле и ожидающих каждую итерацию в течение случайного интервала времени в пределах до двух секунд.

```
static Random r = new Random();
static string Message; // общий ресурс

static void MethodA()
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(r.Next(2000));
```

```
      Message += "A";
      Write(".");
    }
  }

  static void MethodB()
  {
    for (int i = 0; i < 5; i++)
    {
      Thread.Sleep(r.Next(2000));
      Message += "B";
      Write(".");
    }
  }
```

4. В методе `Main` выполните оба метода в отдельных потоках с помощью пары задач и дождитесь их завершения, прежде чем выводить значение затраченного времени в миллисекундах:

```
WriteLine("Please wait for the tasks to complete.");
Stopwatch watch = Stopwatch.StartNew();
Task a = Task.Factory.StartNew(MethodA);
Task b = Task.Factory.StartNew(MethodB);
Task.WaitAll(new Task[] { a, b });
WriteLine();
WriteLine($"Results: {Message}.");
WriteLine($"{watch.ElapsedMilliseconds:#,##0} elapsed milliseconds.");
```

5. Запустите консольное приложение и проанализируйте результат:

```
Please wait for the tasks to complete.
..........
Results: BABABAABBA.
5,753 elapsed milliseconds.
```

Обратите внимание: в результате оба потока модифицировали сообщение одновременно. В реальном приложении это чревато проблемой. Мы можем предотвратить одновременный доступ с помощью взаимоисключающей блокировки, что и сделаем в следующем подразделе.

Применение к ресурсу взаимоисключающей блокировки

Теперь воспользуемся «раковиной» для подтверждения того, что только один поток имеет доступ к ресурсу общего доступа одновременно.

1. В код класса `Program` добавьте экземпляр объектной переменной, выступающей в роли «раковины»:

```
static object conch = new object();
```

2. В код обоих методов, MethodA и MethodB, добавьте оператор lock, окружив им оператор for, как показано в коде ниже:

```
lock (conch)
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(r.Next(2000));
    Message += "A";
    Write(".");
  }
}
```

3. Запустите консольное приложение и проанализируйте результат:

```
Please wait for the tasks to complete.
..........
Results: BBBBBAAAAA.
10,345 elapsed milliseconds.
```

Хотя было затрачено довольно много времени, только один метод одновременно мог получить доступ к общему ресурсу. Первым может быть запущен либо MethodA, либо MethodB. Только после того, как метод завершил работу с общим ресурсом, «раковина освобождается» и другой метод получает возможность выполнять свою функцию.

Оператор блокировки и избежание взаимоблокировок

Вы можете задаться вопросом, как работает оператор lock при блокировке объектной переменной?

```
lock (conch)
{
  // работа с общим ресурсом
}
```

Компилятор C# меняет оператор lock на оператор try-finally, который использует класс Monitor для входа и выхода из «раковины» (переменной conch), как показано ниже:

```
try
{
  Monitor.Enter(conch);
  // работа с общим ресурсом
}
finally
{
  Monitor.Exit(conch);
}
```

Понимание принципа работы оператора `lock` важно, поскольку его использование может привести к взаимоблокировке.

Взаимоблокировка возникает в ситуации наличия двух или более общих ресурсов (и, соответственно, «раковин»), после чего происходит следующая последовательность событий:

- поток X захватывает «раковину» A;
- поток Y захватывает «раковину» B;
- поток X пытается захватить «раковину» B, но блокируется, поскольку поток Y уже захватил ее;
- поток Y пытается захватить «раковину» A, но блокируется, поскольку поток X уже захватил ее.

Проверенный способ предотвратить взаимоблокировки — указать тайм-аут при попытке получить блокировку. Чтобы реализовать такое поведение, вам необходимо вручную использовать класс `Monitor` вместо оператора `lock`.

1. Измените код, заменив оператор `lock` командами входа в «раковину» со временем ожидания:

```
try
{
  if (Monitor.TryEnter(conch, TimeSpan.FromSeconds(15)))
  {
    for (int i = 0; i < 5; i++)
    {
      Thread.Sleep(r.Next(2000));
      Message += "A";
      Write(".");
    }
  }
  else
  {
    WriteLine("Method A failed to enter a monitor lock.");
  }
}
finally
{
  Monitor.Exit(conch);
}
```

2. Запустите консольное приложение и проанализируйте результат. Вы должны увидеть те же результаты, что и раньше (с точностью до того, кто — A или B — захватит «раковину» первым). Однако полученный код более хорош, поскольку позволит вам избежать потенциальных взаимоблокировок.

Используйте ключевое слово lock только в том случае, если можете написать свой код таким образом, чтобы избежать потенциальных взаимоблокировок. Используйте всегда метод Monitor.TryEnter в сочетании с оператором try-finally, чтобы настроить время ожидания и позволить одному из методов отступить, если случится взаимоблокировка.

Синхронизация событий

В главе 6 вы изучили, как создавать и обрабатывать события. Однако события .NET не являются потокобезопасными, поэтому вам следует избегать их использования в многопоточных сценариях и следовать стандартному коду создания событий, который я показал вам ранее.

Многие разработчики пытаются использовать эксклюзивные блокировки при добавлении и удалении обработчиков событий или при создании события, что является плохой идеей, что показано ниже в коде:

```
// поле делегата события
public EventHandler Shout;

// раковина
private readonly object eventLock = new object();

// метод
public void Poke()
{
  lock (eventLock)
  {
    // если что-то прослушивается...
    if (Shout != null)
    {
      // ...затем вызовите делегат, чтобы инициировать событие
      Shout(this, EventArgs.Empty);
    }
  }
}
```

Более подробную информацию о событиях и безопасности потоков вы можете найти на сайте https://docs.microsoft.com/ru-ru/archive/blogs/cburrows/field-like-events-considered-harmful.

Это сложно, как объясняет Стивен Клири в следующем сообщении блога: https://blog.stephencleary.com/2009/06/threadsafe-events.html.

Выполнение атомарных операций

«Атомарный» происходит от греческого слова *atomos*, что означает *«неделимый»*. Важно понимать, какие операции являются атомарными при многопоточности, так как, если они не атомарные, они могут быть прерваны другим потоком на полпути в процессе их выполнения. Является ли операция инкремента C# атомарной, как показано в коде ниже?

```
int x = 3;
x++; // данная операция атомарна?
```

Операция инкремента — не атомарна! Инкремент целого числа требует выполнения центральным процессором следующих трех операций.

1. Загрузка значения из экземпляра переменной в регистр.
2. Инкремент (увеличение) значения.
3. Сохранение значения в экземпляре переменной.

Поток может быть выгружен после выполнения первых двух шагов. Затем второй поток может выполнить все три шага. Возобновив выполнение, первый поток перезапишет значение переменной, а результат инкремента или декремента, выполняемый вторым, будет утерян!

Существует тип `Interlocked`, позволяющий выполнять атомарные действия для типов значений, таких как целые числа и числа с плавающей запятой.

1. Объявите другой общий ресурс, который будет подсчитывать количество выполненных операций:

   ```
   static int Counter; // другой общий ресурс
   ```

2. В коде обоих методов в операторе `for` после изменения значения `string` добавьте следующий оператор для безопасного инкремента счетчика:

   ```
   Interlocked.Increment(ref Counter);
   ```

3. После вывода затраченного времени отобразите значение счетчика, запишите текущее значение счетчика в консоль:

   ```
   WriteLine($"{Counter} string modifications.");
   ```

4. Запустите консольное приложение и проанализируйте результат, как показано в следующем фрагменте вывода:

   ```
   10 string modifications.
   ```

Внимательные читатели поймут, что существующая переменная объекта `conch` защищает *все* общие ресурсы, к которым осуществляется доступ в блоке программного кода, заблокированном «раковиной», и потому в этом конкретном примере фактически нет необходимости задействовать класс `Interlocked`. Но если бы мы еще не защитили другой общий ресурс, такой как `Message`, то использовать класс `Interlocked` было бы необходимо.

Использование других типов синхронизации

Классы `Monitor` и `Interlocked` — это взаимоисключающие блокировки, простые и эффективные, но иногда требуются более сложные параметры для синхронизации доступа к общим ресурсам (табл. 13.4).

Таблица 13.4

Тип	Описание
ReaderWriterLock и ReaderWriterLockSlim (рекомендуются)	Позволяют нескольким потокам быть в режиме чтения, еще одному потоку быть в режиме записи с исключительной блокировкой, и еще одному потоку, имеющему доступ для чтения, быть в повышаемом режиме чтения, из которого поток может перейти в режим записи без необходимости отказываться от доступа для чтения к ресурсу
Mutex	Подобно Monitor, класс Mutex обеспечивает эксклюзивный доступ к общему ресурсу, за исключением того, что используется для синхронизации между процессами
Semaphore и SemaphoreSlim	Эти классы ограничивают количество потоков, которые могут одновременно обращаться к ресурсу или пулу ресурсов, определяя слоты
AutoResetEvent и ManualResetEvent	Обработчики ожидания событий позволяют потокам синхронизировать действия, сигнализируя друг другу и ожидая сигналов друг от друга

Ключевые слова async и await

В версии C# 5 были введены два ключевых слова для упрощения работы с типом `Task`. Они особенно полезны для таких ситуаций, как:

- реализация многозадачности для *графического интерфейса пользователя* (graphical user interface, GUI);
- улучшение масштабируемости веб-приложений и веб-сервисов.

В главе 16 мы изучим, как с помощью ключевых слов `async` и `await` улучшить масштабируемость сайтов.

В главе 21 мы рассмотрим, как ключевые слова `async` и `await` позволяют реализовывать многозадачность в GUI-приложениях.

Однако в настоящий момент мы узнаем, почему эти два ключевых слова были введены в язык C#, а затем воспользуемся ими на практике.

Увеличение скорости отклика консольных приложений

Одно из ограничений консольных приложений заключается в том, что слово `await` можно использовать только в методах, помеченных как `async...`, при этом C# 7 и более ранние версии не позволяют методу `Main` быть `async`! К счастью, в C# 7.1 была введена новая функция — поддержка `async` для метода `Main`.

1. Создайте консольное приложение `AsyncConsole`, добавьте его в рабочую область `Chapter13` и выберите проект в качестве активного для OmniSharp.
2. Импортируйте пространства имен для выполнения HTTP-запросов и работы с задачами, а также статически импортируйте класс `Console`:

```
using System.Net.Http;
using System.Threading.Tasks;
using static System.Console;
```

3. Добавьте в метод `Main` операторы для создания экземпляра `HttpClient`, сделайте запрос на главную страницу Apple и выведите ее размер в байтах, как показано ниже:

```
var client = new HttpClient();

HttpResponseMessage response =
  await client.GetAsync("http://www.apple.com/");

WriteLine("Apple's home page has {0:N0} bytes.",
  response.Content.Headers.ContentLength);
```

4. Соберите проект и обратите внимание на сообщение об ошибке, как показано ниже:

```
Program.cs(14,9): error CS4033: The 'await' operator can only be used
within an async method. Consider marking this method with the 'async'
modifier and changing its return type to 'Task'. [/Users/markjprice/Code/
Chapter13/AsyncConsole/AsyncConsole.csproj]
```

5. Добавьте ключевое слово `async` к методу `Main` и измените возвращаемый тип на `Task`.
6. Создайте проект и обратите внимание, что на этот раз все прошло успешно.

7. Запустите консольное приложение и проанализируйте результат, который, вероятно, будет иметь другое количество байтов, поскольку Apple часто меняет свою домашнюю страницу:

```
Apple's home page has 40,252 bytes.
```

Увеличение скорости отклика GUI-приложений

В этой книге мы создавали только консольные приложения. Жизнь разработчика усложняется при создании веб-приложений, веб-сервисов и приложений с графическим интерфейсом, таких как настольные Windows-приложения и мобильные приложения.

Одна из причин — существование для GUI-приложения специального *потока пользовательского интерфейса* (UI).

Существует два правила для работы в GUI:

- не выполняйте задачи с длительным временем выполнения в потоке пользовательского интерфейса;
- не обращайтесь к UI-элементам ни в одном потоке, кроме потока пользовательского интерфейса.

Для соблюдения этих правил разработчикам приходилось писать сложный код, чтобы выполнение длительных задач не производилось UI-потоком, но при этом после завершения такой задачи результаты ее выполнения безопасно передавались UI-потоку для отображения пользователю. Это чревато тем, что все может пойти наперекосяк!

К счастью, в C# 5 и более поздних версиях были введены ключевые слова `async` и `await`. Они позволяют вам продолжать писать код, как если бы он был синхронным, что делает его чистым и легким для понимания. Однако под капотом компилятор C# создает сложную машину состояний и отслеживает запущенные потоки. Это нечто удивительное!

Улучшение масштабируемости клиент-серверных приложений

Ключевые слова `async` и `await` могут применяться на стороне сервера при разработке сайтов, веб-приложений и сервисов. С точки зрения клиентского приложения ничего не меняется (или даже может немного увеличиться время возврата запроса). Таким образом, с точки зрения одного клиента, применение `async` и `await` для реализации многозадачности на стороне сервера негативно влияет на пользовательский опыт!

На стороне сервера, однако, создаются дополнительные, более дешевые рабочие потоки, ожидающие завершения длительных задач, чтобы дорогостоящие потоки ввода-вывода могли обрабатывать запросы других клиентов без блокировки. Это положительно сказывается на общей масштабируемости веб-приложения или сервиса. Одновременно обеспечивается поддержка большего количества клиентов.

Распространенные типы, поддерживающие многозадачность

В табл. 13.5 приведены наиболее распространенные типы, поддерживающие асинхронные методы.

Таблица 13.5

Тип	Методы
DbContext<T>	AddAsync, AddRangeAsync, FindAsync и SaveChangesAsync
DbSet<T>	AddAsync, AddRangeAsync, ForEachAsync, SumAsync, ToListAsync, ToDictionaryAsync, AverageAsync и CountAsync
HttpClient	GetAsync, PostAsync, PutAsync, DeleteAsync и SendAsync
StreamReader	ReadAsync, ReadLineAsync и ReadToEndAsync
StreamWriter	WriteAsync, WriteLineAsync и FlushAsync

Каждый раз, встречая метод, оканчивающийся суффиксом Async, вы должны проверять, возвращает ли он метод Task или Task<T>. Если да, то необходимо использовать этот метод вместо синхронного метода без суффикса Async. Не забудьте, что вызывать его надо с ключевым словом await и добавив к сигнатуре вызывающего метода слово async.

Ключевое слово await в блоках catch

В версии C# 5 ключевое слово `await` можно было использовать только в блоке обработки исключений `try`, но не в `catch`. В C# 6 и более поздней версии вы можете применять ключевое слово `await` в обоих блоках: и `try`, и `catch`.

Работа с асинхронными потоками

До выпуска версии C# 8.0 и .NET Core 3.0 ключевое слово `await` можно было использовать только с задачами, возвращающими скалярные значения. Поддержка

асинхронного потока в .NET Standard 2.1 позволила асинхронному методу возвращать последовательность значений.

Рассмотрим следующий пример.

1. Создайте консольное приложение `AsyncEnumerable`, добавьте его в рабочую область `Chapter13` и выберите проект в качестве активного для OmniSharp.
2. Для работы с задачами импортируйте пространства имен и статически импортируйте класс `Console`:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using static System.Console;
```

3. Создайте метод, который асинхронно возвращает случайную последовательность из трех чисел:

```
async static IAsyncEnumerable<int> GetNumbers()
{
  var r = new Random();

  // имитация работы
  await Task.Run(() => Task.Delay(r.Next(1500, 3000)));
  yield return r.Next(0, 1001);

  await Task.Run(() => Task.Delay(r.Next(1500, 3000)));
  yield return r.Next(0, 1001);

  await Task.Run(() => Task.Delay(r.Next(1500, 3000)));
  yield return r.Next(0, 1001);
}
```

4. Добавьте в метод `Main` операторы для перечисления последовательности чисел, как показано ниже:

```
static async Task Main(string[] args)
{
  await foreach (int number in GetNumbers())
  {
    WriteLine($"Number: {number}");
  }
}
```

5. Запустите консольное приложение и проанализируйте результат:

```
Number: 509
Number: 813
Number: 307
```

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 13.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какую информацию о классе `Process` вы можете найти?
2. Насколько точен класс `Stopwatch`?
3. По соглашению какой суффикс должен быть применен к методу, если он возвращает `Task` или `Task<T>`?
4. Какое ключевое слово следует применить к объявлению метода, чтобы в нем можно было использовать ключевое слово `await`?
5. Как создать дочернюю задачу?
6. Почему не следует использовать ключевое слово `lock`?
7. В каких случаях нужно применять класс `Interlocked`?
8. Когда класс `Mutex` следует задействовать вместо класса `Monitor`?
9. В чем преимущество использования на сайте или в веб-сервисе методов `async` и `await`?
10. Вы можете отменить задачу? Каким образом?

Упражнение 13.2. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- потоки и работа с ними: https://docs.microsoft.com/ru-ru/dotnet/standard/threading/threads-and-threading;
- асинхронное программирование: https://docs.microsoft.com/ru-ru/dotnet/standard/async-in-depth;
- `await` (справочник по C#): https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/await;
- параллельное программирование в .NET: https://docs.microsoft.com/ru-ru/dotnet/standard/parallel-programming/;
- примитивы синхронизации: https://docs.microsoft.com/ru-ru/dotnet/standard/threading/overview-of-synchronization-primitives.

Резюме

Вы узнали, как определять и запускать задачи, дожидаться выполнения одной или нескольких задач и управлять порядком завершения задач. Вы также узнали, как использовать ключевые слова `async` и `await` и синхронизировать доступ к общим ресурсам.

Из оставшихся глав вы узнаете, как создавать приложения для *моделей приложений*, поддерживаемых .NET, таких как сайты, веб-приложения и веб-сервисы. В качестве бонуса вы научитесь создавать настольные Windows-приложения, используя .NET 5, и кросс-платформенные мобильные приложения, задействуя Xamarin.Forms.

14 Практическое применение C# и .NET

Третья часть этой книги посвящена практическому применению C# и .NET. Вы узнаете, как создавать полноценные кросс-платформенные приложения, такие как сайты, веб-сервисы, настольные Windows-приложения и мобильные приложения, и как добавлять к ним интеллектуальные возможности с помощью машинного обучения. Корпорация Microsoft предоставляет платформы для создания приложений на основе *моделей*.

Я рекомендую вам проработать эту и следующие главы последовательно, так как далее будут ссылки на проекты из предыдущих глав, а вы накопите достаточные знания и навыки для решения более сложных вопросов в изложенных далее главах.

В этой главе:

- модели приложений для C# и .NET;
- нововведения ASP.NET Core;
- SignalR;
- Blazor;
- бонус;
- разработка модели данных объекта для Northwind.

Модели приложений для C# и .NET

В оставшихся главах мы узнаем о моделях приложений, построенных на использовании для создания фактических приложений C# 9 и .NET 5, которым и посвящена эта книга.

Корпорация Microsoft предоставляет обширное руководство по внедрению моделей приложений, таких как веб-приложения ASP.NET, мобильные приложения Xamarin и приложения UWP, в документации по архитектуре приложений .NET, которую можно найти на сайте https://www.microsoft.com/net/learn/architecture.

Разработка сайтов с помощью ASP.NET Core

Сайты состоят из нескольких страниц, загружаемых статически из файловой системы или динамически генерируемых технологией разработки кода на стороне сервера, такой как ASP.NET Core. Браузер выполняет запросы `GET`, используя URL, идентифицирующие каждую страницу, и может управлять данными, хранящимися на сервере, с помощью запросов `POST`, `PUT` и `DELETE`.

Для большинства сайтов браузер рассматривается как уровень представления, причем почти вся обработка выполняется на стороне сервера. На стороне клиента для реализации некоторых функций презентации может использоваться небольшое количество кода JavaScript, например карусель.

Платформа ASP.NET Core предоставляет три технологии для создания сайтов:

- *ASP.NET Core Razor Pages* и *библиотеки классов Razor* служат для динамического создания HTML для простых сайтов. Более подробно об этом вы узнаете в главе 15;
- *ASP.NET Core MVC* — реализация шаблона проектирования Model-View-Controller (*MVC*, «Модель — Представление — Контроллер»), популярного при разработке сложных сайтов. Более подробную информацию об этом вы найдете в главе 16;
- *Blazor* позволяет создавать компоненты на стороне сервера или клиента и пользовательские интерфейсы, задействуя язык C# вместо JavaScript.

Разработка сайтов с использованием системы управления веб-контентом (веб-содержимым)

Большинство сайтов содержит много контента, и если бы приходилось каждый раз привлекать разработчиков для изменения какого-либо контента, то сайты не могли бы хорошо масштабироваться. *Система управления веб-контентом* (Content Management System, CMS) позволяет разработчикам определять структуру и шаблоны контента, чтобы обеспечить согласованность и хороший дизайн, а не подкованному в техническом плане владельцу контента дает возможность легко управлять фактическим наполнением сайта. Владелец может создавать новые страницы или блоки контента и обновлять существующий контент, зная, что не потребуется особых усилий, чтобы сайт отлично смотрелся для конечного пользователя.

Существует множество CMS, доступных для всех веб-платформ, таких как WordPress для PHP или Django для Python. Среди CMS корпоративного уровня для .NET Framework можно выделить Episerver и Sitecore, однако ни одна из них еще не доступна для .NET Core или .NET 5 и более поздних версий. Среди CMS, поддерживающих .NET Core, стоит упомянуть Piranha CMS, Squidex и Orchard Core.

Основное преимущество использования CMS состоит в том, что эта система обеспечивает дружественный UI управления контентом. Владельцы контента заходят на сайт и сами им управляют. Затем контент отрисовывается и возвращается пользователям с помощью контроллеров и представлений ASP.NET MVC.

Таким образом, C# и .NET можно использовать для создания как серверного, так и клиентского кода сайтов, как показано на рис. 14.1.

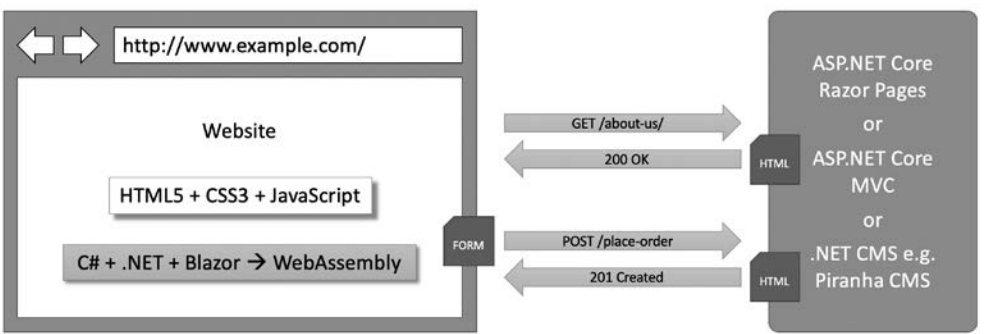


Рис. 14.1. Использование C# и .NET для создания сайтов

Веб-приложения

Веб-приложения, также известные как *одностраничные приложения* (single page application, SPA), состоят из одной веб-страницы, созданной с помощью таких фронтенд-технологий, как Angular, React, Vue, или проприетарной библиотеки JavaScript, которая может отправлять запросы к бэкенд-веб-сервису, чтобы получить при необходимости больше данных и публиковать обновленные данные, задействуя распространенные форматы сериализации, такие как XML и JSON. Классические примеры — это веб-приложения Google: Gmail, «Карты» и «Документы».

В веб-приложении на стороне клиента для реализации сложных взаимодействий с пользователем применяются библиотеки JavaScript, но большая часть важной обработки и доступа к данным все еще происходит на стороне сервера, поскольку браузер имеет ограниченный доступ к ресурсам локальной системы.

JavaScript имеет слабую типизацию и не предназначен для сложных проектов, поэтому большинство библиотек JavaScript в настоящее время используют Microsoft TypeScript, который добавляет строгую типизацию в JavaScript и разработан со множеством современных языковых функций для обработки сложных реализаций. Версия TypeScript 4.0 была выпущена в августе 2020 года. Более подробно о TypeScript вы можете узнать на сайте https://www.typescriptlang.org.

В .NET существуют шаблоны проектов для SPA на основе JavaScript и TypeScript, но мы не будем тратить на это время в нашей книге, хоть они и разрабатываются обычно с ASP.NET Core в роли бэкенда.

Чтобы узнать больше о создании фронтендов для .NET с помощью JavaScript SPA на основе TypeScript, обратитесь к двум интересным книгам издательства Packt, с которыми можно ознакомиться на следующих сайтах:

- ASP.NET Core 2 and Vue.js: https://www.packtpub.com/product/asp-net-core-2-and-vue-js/9781788839464;
- ASP.NET Core 3 and React: https://www.packtpub.com/product/asp-net-core-3-and-react/9781789950229;
- ASP.NET Core 3 and Angular 9: https://www.packtpub.com/product/asp-net-core-3-and-angular-9-third-edition/9781789612165.

Создание и использование веб-сервисов

Даже если мы не будем изучать SPA на основе JavaScript, мы узнаем, как создать веб-сервис с помощью *ASP.NET Core Web API*, затем вызвать его из серверного кода на сайтах ASP.NET Core. Позже мы будем вызывать этот веб-сервис из веб-компонентов Blazor, настольных Windows-приложений и кросс-платформенных мобильных приложений.

Разработка интеллектуальных приложений

В классическом приложении алгоритмы, использующиеся для обработки данных, разрабатываются и реализуются человеком. Люди многое умеют делать хорошо, однако написание сложных алгоритмов не их конек, особенно алгоритмов определения полезных шаблонов в огромных объемах данных.

Добавить интеллектуальность в ваши приложения могут алгоритмы машинного обучения, работающие с пользовательскими моделями, подобные тем, которые предоставляются *ML.NET* от Microsoft для .NET Core. Мы будем применять алгоритмы ML.NET с пользовательскими моделями в целях обработки отслеживаемого поведения посетителей сайта, а затем дадим рекомендации для других страниц, которые могут их заинтересовать. Это будет работать примерно так, как Netflix рекомендует фильмы и телешоу, которые могут вам понравиться, основываясь на вашем предыдущем поведении и поведении пользователей со схожими интересами.

Нововведения ASP.NET Core

За последние несколько лет корпорация Microsoft быстро расширила возможности ASP.NET Core. Платформы .NET, которые поддерживаются на сегодняшний день, перечислены ниже:

- версия ASP.NET 1.0 до 2.2 работает на основе платформы .NET Core или .NET Framework;
- версия ASP.NET Core 3.0 работает только на основе платформы .NET Core 3.0 или более поздних версий.

ASP.NET Core 1.0

Версия ASP.NET Core 1.0 была выпущена в июне 2016 года и ориентирована на реализацию API, подходящего для создания современных кросс-платформенных веб-сервисов и сервисов для операционных систем Windows, macOS и Linux.

Более подробную информацию о версии ASP.NET Core 1.0 можно получить на сайте https://devblogs.microsoft.com/aspnet/announcing-asp-net-core-1-0/.

ASP.NET Core 1.1

Версия ASP.NET Core 1.1 была выпущена в ноябре 2016 года и фокусировалась на исправлении обнаруженных багов и общих улучшениях функционала и производительности.

Более подробную информацию о версии ASP.NET Core 1.1 можно получить на сайте https://devblogs.microsoft.com/aspnet/announcing-asp-net-core-1-1/.

ASP.NET Core 2.0

Версия ASP.NET Core 2.0 была выпущена в августе 2017 года и ориентирована на добавление новых функций, таких как Razor Pages, объединение сборок в метапакет `Microsoft.AspNetCore.All`, поддержку .NET Standard 2.0, предоставление новой модели аутентификации и улучшение производительности. Самая крупная новая функция описана в главе 15.

Более подробную информацию о версии ASP.NET Core 2.0 можно получить на сайте https://devblogs.microsoft.com/aspnet/announcing-asp-net-core-2-0/.

ASP.NET Core 2.1

Версия ASP.NET Core 2.1 была выпущена в мае 2018 года с долгосрочной поддержкой (LTS), что означает, что версия будет поддерживаться до 21 августа 2021 года.

Данная версия нацелена на добавление новых функций, таких как SignalR для связи в реальном времени, библиотеки классов Razor и ASP.NET Core Identity. Кроме того, эта версия фокусировалась на улучшении поддержки HTTPS и *Общего регламента по защите данных в ЕС* (General Data Protection Regulation, GDPR), включая темы, перечисленные в табл. 14.1.

Таблица 14.1

Особенности	Глава	Тема
SignalR	14	Введение в SignalR
Библиотеки классов Razor	15	Использование библиотек классов Razor
Поддержка GDPR	16	Разработка и изучение сайта ASP.NET Core MVC
UI-библиотека Identity и формирование шаблонов	16	Изучение сайта ASP.NET Core MVC
Интеграционное тестирование	16	Тестирование сайта ASP.NET Core MVC
[ApiController], ActionResult<T>	18	Разработка проекта ASP.NET Core Web API
Детали проблемы	18	Спецификация деталей проблемы
IHttpClientFactory	18	Настройка HTTP-клиентов с помощью HttpClientFactory

Более подробную информацию о версии ASP.NET Core 2.1 можно получить на сайте https://devblogs.microsoft.com/aspnet/asp-net-core-2-1-0-now-available/.

ASP.NET Core 2.2

Версия ASP.NET Core 2.2 была выпущена в декабре 2018 года и фокусировалась на улучшении построения RESTful HTTP API, обновлении шаблонов проектов до Bootstrap 4 и Angular 6, оптимизированной конфигурации для хостинга в Azure и повышении производительности, включая темы, перечисленные в табл. 14.2.

Таблица 14.2

Особенности	**Глава**	**Тема**
HTTP/2 в Kestrel	15	Классический ASP.NET против современного ASP.NET Core
Модель внутрипроцессного хостинга	15	Разработка проекта ASP.NET Core
Health Check API	18	Реализация Health Check API
Анализаторы Open API	18	Реализация анализаторов и соглашений Open API
Маршрутизация к конечным точкам	18	Настройки маршрутизации к конечным точкам

Более подробную информацию о версии ASP.NET Core 2.2 можно получить на сайте https://devblogs.microsoft.com/aspnet/asp-net-core-2-2-available-today/.

ASP.NET Core 3.0

Версия ASP.NET Core 3.0 была выпущена в сентябре 2019 года и нацелена на полноценное использование .NET Core 3.0. Это значит, что данная версия больше не может поддерживать .NET Framework и была дополнена полезными обновлениями, включая темы, перечисленные в табл. 14.3.

Таблица 14.3

Особенности	**Глава**	**Тема**
Blazor; на стороне сервера и клиента	14	Ознакомление с Blazor
Статическое содержимое в библиотеках классов Razor	15	Использование библиотек классов Razor
Новые настройки регистрации сервиса MVC	16	Запуск ASP.NET Core MVC

Более подробную информацию о версии ASP.NET Core 3.0 можно получить на сайте https://devblogs.microsoft.com/aspnet/a-first-look-at-changes-coming-in-asp-net-core-3-0/.

ASP.NET Core 3.1

Версия ASP.NET Core 3.1 была выпущена в декабре 2019 года и является выпуском LTS, что означает, что она будет поддерживаться до 3 декабря 2022 года. Данная

версия нацелена на такие улучшения, как поддержка частичного класса для компонентов Razor и создание нового помощника тега компонента.

Информацию, касающуюся версии ASP.NET Core 3.1, можно прочитать на сайте https://devblogs.microsoft.com/aspnet/asp-net-core-updates-in-net-core-3-1/

Blazor WebAssembly 3.2

Версия Blazor WebAssembly 3.2 была выпущена в мае 2020 года и является текущим выпуском, что означает, что проекты должны быть обновлены до версии .NET 5 в течение трех месяцев после выпуска .NET 5, то есть до 10 февраля 2021 года.

Корпорация Microsoft наконец выполнила обещание о полноценной веб-разработке с помощью .NET, Blazor Server и Blazor WebAssembly, которые мы будем изучать в главе 20.

Информацию, касающуюся Blazor WebAssembly, можно прочитать на сайте https://devblogs.microsoft.com/aspnet/blazor-webassembly-3-2-0-now-available/.

ASP.NET Core 5.0

Версия ASP.NET Core 5.0 была выпущена в ноябре 2020 года и нацелена на исправление ошибок, улучшение производительности с использованием кэширования для проверки подлинности сертификата, динамическом сжатии HPack заголовков ответов HTTP/2 в Kestrel, аннотациях с нулевым значением для сборок ASP.NET Core и сокращении контейнера. размеры изображений, включая темы, перечисленные в табл. 14.4.

Таблица 14.4

Особенности	**Глава**	**Тема**
Метод расширения для разрешения анонимного доступа к конечной точке	18	Защита веб-сервисов
Методы расширения JSON для HttpRequest и HttpResponse	18	Получение клиентов в виде JSON в контроллере

Информацию, касающуюся версии ASP.NET Core 5.0, можно найти на сайте https://github.com/markjprice/cs9dotnet5/.

SignalR

В первые годы существования Интернета в 1990-х браузеры должны были отправлять HTTP-запрос `GET` на получение всей страницы веб-серверу, чтобы получить свежую информацию.

В конце 1999 года корпорация Microsoft выпустила Internet Explorer 5.0 с компонентом *XMLHttpRequest*, который мог выполнять асинхронные HTTP-вызовы в фоновом режиме. Это нововведение наряду с *динамическим HTML* (dynamic HTML, DHTML) позволило плавно обновлять новыми данными фрагменты веб-страницы.

Преимущества данной технологии были очевидны, и вскоре все браузеры добавили тот же компонент. Компания Google максимально использовала эту возможность для создания умных веб-приложений, таких как Google Maps и Gmail. Несколько лет спустя технология стала широко известна как *асинхронный JavaScript и XML (AJAX)*.

Однако AJAX использует все тот же протокол HTTP для связи, что накладывает некоторые ограничения. Во-первых, HTTP — это протокол связи «запрос — ответ», означающий, что сервер не может передавать данные клиенту. Он должен ждать, пока клиент сделает запрос. Во-вторых, сообщения HTTP-запроса и ответа содержат заголовки с большим количеством потенциально ненужных служебных данных. В-третьих, HTTP обычно требует, чтобы при каждом запросе создавалось новое базовое TCP-соединение.

WebSocket — протокол дуплексной связи, то есть клиент или сервер могут инициировать передачу новых данных. WebSocket использует одно и то же TCP-соединение в жизненном цикле своего соединения. Кроме того, протокол более эффективен с точки зрения размера отправляемых сообщений, поскольку они дополнены всего двумя байтами технической информации.

Протокол WebSocket работает через порты HTTP `80` и `443`, поэтому совместим с протоколом HTTP, и подтверждение WebSocket использует заголовок HTTP Upgrade для переключения с протокола HTTP на протокол WebSocket.

Более подробную информацию о протоколе WebSocket можно получить на сайте https://en.wikipedia.org/wiki/WebSocket.

Ожидается, что современные веб-приложения будут предоставлять актуальную информацию. Живой чат — классический пример. Однако существует множество потенциальных приложений, от биржевых курсов до игр.

Всякий раз, когда вам нужен сервер для отправки обновлений на веб-страницу, вам требуется веб-совместимая технология связи в реальном времени. Можно использовать протокол WebSocket, но он поддерживается не всеми клиентами.

ASP.NET Core SignalR — это библиотека с открытым исходным кодом, которая упрощает добавление веб-функциональности в реальном времени к приложениям, являясь абстракцией для нескольких базовых коммуникационных технологий. Это позволяет добавлять коммуникационные возможности в реальном времени с помощью кода C#.

Разработчику не нужно понимать или реализовывать используемую базовую технологию, и SignalR автоматически переключается между базовыми технологиями в зависимости от того, что поддерживает браузер пользователя. Например, SignalR будет применять WebSocket, если этот протокол доступен в браузере, и другие технологии, такие как длинный опрос AJAX, — в противном случае, в то время как код вашего приложения остается прежним.

SignalR — это API для *удаленного вызова* сервером *процедур* клиента (remote procedure calls, RPC). Через RPC вызываются JavaScript-функции клиента из серверного кода .NET Core. SignalR применяет концентраторы для определения конвейера доставки сообщений и обрабатывает отправку сообщений автоматически, используя два встроенных протокола концентратора: JSON и двоичный протокол на основе MessagePack.

Более подробную информацию о MessagePack можно получить на сайте https://msgpack.org.

На стороне сервера SignalR работает везде, где работает ASP.NET Core: на серверах Windows, macOS или Linux. Библиотека SignalR поддерживает следующие клиентские платформы:

- клиенты JavaScript для текущих браузеров, включая Google Chrome, Firefox, Safari, Edge и Internet Explorer 11;
- клиенты .NET, включая мобильные приложения Blazor и Xamarin для Android и iOS;
- Java 8 и более поздние версии.

Более подробную информацию о библиотеке SignalR можно получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/signalr/introduction.

Blazor

Фреймворк Blazor позволяет создавать общие компоненты и интерактивные веб-интерфейсы с помощью языка C# вместо JavaScript. В апреле 2019 года корпорация Microsoft объявила, что Blazor «больше не является экспериментальным, и мы

обязуемся поставлять его в качестве поддерживаемого фреймворка разработки веб-интерфейса, включая поддержку запуска на стороне клиента в браузере в WebAssembly».

JavaScript и другие технологии

Традиционно любой код, который необходимо выполнить в браузере, пишется с помощью языка программирования JavaScript или технологии более высокого уровня, которая транспилируется (трансформируется или компилируется) в JavaScript. Это связано с тем, что все браузеры поддерживают JavaScript в течение примерно двух десятилетий, поэтому он стал базовым решением для реализации бизнес-логики на стороне клиента.

Однако при использовании JavaScript могут возникнуть некоторые проблемы. Во-первых, хотя данный язык при первом взгляде во многом схож с языками семейства C, такими как C# и Java, при более пристальном изучении между ними можно обнаружить весьма существенные различия. Во-вторых, это псевдофункциональный язык с динамической типизацией, применяющий прототипы вместо наследования классов для повторного использования объектов.

Разве не было бы замечательно, имей мы возможность использовать в браузере те же язык и библиотеки, что и на стороне сервера?

Silverlight — использование C# и .NET через плагин

Корпорация Microsoft уже делала попытку достичь этой цели с помощью технологии *Silverlight*. Когда в 2008 году была выпущена версия Silverlight 2.0, разработчики на C# и .NET могли применить свои навыки для создания библиотек и визуальных компонентов, которые выполнялись в браузере с помощью плагина Silverlight.

К 2011 году и Silverlight 5.0 успех Apple с iPhone и ненависть Стива Джоба к плагинам браузера, таким как Flash, в конечном итоге привели к тому, что корпорация Microsoft отказалась от плагина Silverlight, поскольку аналогично Flash Silverlight запрещен на iPhone и iPad.

WebAssembly — база для Blazor

Недавнее развитие браузеров дало корпорации Microsoft возможность сделать еще одну попытку. В 2017 году был разработан WebAssembly Consensus, и теперь его поддерживают все основные браузеры: Chromium (Google Chrome, Edge, Opera, Brave), Firefox и WebKit (Safari). WebAssembly не поддерживается браузером Microsoft Internet Explorer, поскольку тот считается устаревшим.

WebAssembly (Wasm) — это двоичный формат инструкций для виртуальной машины, обеспечивающий способ выполнения кода, написанного на нескольких языках в веб-среде, с почти естественной скоростью. Wasm разработан в качестве переносной платформы для компиляции языков высокого уровня, таких как C#.

Более подробную информацию о WebAssembly можно получить на сайте https://webassembly.org.

Blazor — на стороне сервера или клиента

Blazor — это единая модель приложения с двумя моделями *хостинга*:

- серверная часть работает на стороне сервера, используя SignalR для связи со стороной клиента, и поставляется в составе .NET Core 3.0;
- клиентская часть работает на стороне клиента на базе WebAssembly и поставляется как расширение .NET Core 3.1 в составе будущего выпуска .NET Core.

Веб-разработчик может написать компоненты Blazor один раз, а затем запустить их либо на стороне сервера, либо на стороне клиента.

Официальную документацию по Blazor можно найти на сайте https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor.

Дополнительные материалы

Бонус — это последняя глава и приложение Б данной книги:

- глава 21 «Разработка кросс-платформенных мобильных приложений»;
- приложение Б «Разработка настольных Windows-приложений».

Будучи посвященной современной кросс-платформенной разработке с помощью C# 9 и .NET 5, технически эта книга не должна включать в себя описание настольных Windows-приложений, поскольку они предназначены только для операционной системы Windows. Кроме того, книга не должна охватывать кросс-платформенные мобильные приложения, так как для их разработки используется платформа Xamarin вместо .NET 5.

В главах с 1-й по 20-ю для разработки всех приложений мы используем кросс-платформенную среду Visual Studio Code. Настольные Windows-приложения создаются с помощью Visual Studio 2019 для операционной системы Windows 10. Кросс-платформенные мобильные приложения разрабатываются с использованием

среды Visual Studio 2019 для Mac и требуют для компиляции операционной системы macOS.

Однако операционная система Windows и мобильные устройства — важные платформы для разработки текущих и будущих клиентских приложений с помощью C# и .NET, поэтому я не хотел упускать возможность познакомить вас с ними.

Приложение Б можно найти по адресу https://clck.ru/XtWkK.

Разработка кросс-платформенных мобильных и настольных Windows-приложений

Существует две основные мобильные платформы: Apple iOS и Google Android, каждая из которых имеет собственные языки программирования и платформенные API. Существуют также две основные настольные платформы: MacOS от Apple и Windows от Microsoft, каждая со своими собственными языками программирования и API платформы.

Кросс-платформенные мобильные и настольные приложения могут быть разработаны один раз для платформы Xamarin с помощью C#, а затем работать на мобильных платформах. Xamarin.Forms еще более упрощает разработку мобильных приложений, предоставляя возможность создавать общие компоненты пользовательского интерфейса, а не только бизнес-логику. Для определения пользовательского интерфейса большая часть XAML может даже совместно использоваться приложениями Xamarin.Forms, WPF и UWP.

Приложения могут существовать сами по себе, но обычно вызывают веб-сервисы, чтобы обеспечить взаимодействие, охватывающее все ваши вычислительные устройства, от серверов и ноутбуков до телефонов и игровых систем.

После выпуска .NET 6 в ноябре 2021 года вы сможете создавать кросс-платформенные мобильные и настольные приложения, ориентированные на те же API в .NET 6, которые используются консольными приложениями, сайтами, веб-сервисами и настольными Windows-приложениями. Приложения будут выполняться на мобильных устройствах средой выполнения Xamarin. Будет использоваться эволюция Xamarin.Forms, известная как .NET MAUI или многоплатформенный пользовательский интерфейс приложения (пользовательский интерфейс).

Текущая версия Xamarin.Forms требует наличия среды разработки Visual Studio для Mac или Visual Studio 2019 для Windows. MAUI будет поддерживать эти инструменты, а также код Visual Studio через расширение. Шестое издание этой книги наконец сможет использовать только код Visual Studio для 100 % примеров кода.

.NET MAUI будет поддерживать существующие шаблоны MVVM и XAML, а также такие, как Model-View-Update (MVU) с C#, который похож на Apple Swift UI, и Blazor.

Введение в .NET MAUI вы можете прочитать на сайте https://devblogs.microsoft.com/dotnet/introduction-net-multi-platform-app-ui/.

Последняя глава следующего, шестого издания этой книги будет переименована в главу 21 «Создание кросс-платформенных мобильных и настольных приложений» и будет посвящена использованию пользовательских интерфейсов многоплатформенных приложений .NET (MAUI) для создания кросс-платформенных мобильных и настольных приложений.

Репозиторий GitHub для .NET MAUI вы можете просмотреть по следующей ссылке: https://github.com/dotnet/maui.

Разработка настольных Windows-приложений с использованием устаревших технологий

В первой версии C# и .NET Framework, выпущенной в 2002 году, корпорация Microsoft предоставила технологию создания настольных Windows-приложений под названием *Windows Forms*. (Эквивалент на то время для веб-разработки назывался *Web Forms*, отсюда и дополняющие друг друга названия.)

В 2007 году корпорация Microsoft выпустила более эффективную технологию для разработки настольных Windows-приложений под названием *Windows Presentation Foundation (WPF)*. Для определения своего пользовательского интерфейса технология WPF может применять *расширяемый язык разметки приложений* (eXtensible Application Markup Language, XAML) — простой для понимания как человеком, так и программным кодом. Visual Studio 2019 разработана с помощью WPF.

Существует множество корпоративных приложений, созданных с использованием Windows Forms и WPF, которые необходимо поддерживать или расширять новыми функциями, но до сих пор они содержались в платформе .NET Framework, в настоящее время считающейся устаревшей. В .NET 5 и Windows Desktop Pack эти приложения могут использовать все современные возможности .NET.

В 2015 году корпорация Microsoft выпустила операционную систему Windows 10, а вместе с ней и новую технологию под названием *Universal Windows Platform (UWP)*. Приложения UWP могут быть созданы с использованием C++ и DirectX UI, JavaScript и HTML или C# с использованием настроек .NET Core, которая

не является кросс-платформенной, но обеспечивает полный доступ к базовым API WinRT.

Приложения UWP могут выполняться только на платформе Windows 10, но не в более ранних версиях Windows. Кроме того, они могут работать на консоли Xbox и гарнитурах Windows Mixed Reality с контроллерами движения.

Технологии Windows Forms и WPF для создания классических Windows-приложений устаревшие, и очень немногие разработчики разрабатывают UWP-приложения, поэтому данная тема освещена в приложении и только для онлайн-доступа.

Когда станут доступны .NET 6 и .NET MAUI для создания кросс-платформенных приложений, которые могут работать на Рабочем столе Windows, а также на многих других платформах, в следующем издании книги это приложение будет удалено.

Если вас заинтересовала тема разработки WPF-приложений, вам будет полезна книга издательства Packt: https://www.packtpub.com/product/mastering-windows-presentation-foundation/9781785883002.

Разработка модели данных объекта для Northwind

Реальные приложения обычно должны работать с данными в реляционной БД или другом хранилище данных. В этой главе мы определим модель данных для сущностей на основе БД `Northwind`, хранящейся в SQLite. Такая модель будет использоваться в большинстве приложений, которые мы создадим в последующих главах.

Операционная система macOS по умолчанию включает в себя установку SQLite, однако если вы используете операционную систему Windows или Linux, то вам может потребоваться скачать, установить и настроить SQLite для вашей операционной системы. Инструкции по установке описаны в главе 11. В этой главе вы также найдете инструкции по установке инструмента dotnet-ef, который можно будет использовать для построения модели сущности из существующей базы данных.

Вам следует создать отдельный проект библиотеки классов для ваших сущностных моделей данных. Это позволяет упростить обмен между внутренними веб-серверами и клиентскими компьютерами, мобильными устройствами и клиентскими приложениями Blazor.

Разработка библиотеки классов для сущностных моделей Northwind

Теперь вы определите сущностные модели данных в библиотеке классов .NET 5 для их повторного использования в других типах проектов, включая модели приложений на стороне клиента.

Создание моделей сущностей с использованием dotnet-ef

В первую очередь мы автоматически сгенерируем некоторые модели сущностей с помощью инструмента командной строки EF Core.

1. В существующей папке Code создайте подпапку PracticalApps.
2. В программе Visual Studio откройте папку PracticalApps.
3. Создайте файл `Northwind.db`, скопировав файл `Northwind.sql` в папку PracticalApps, а затем на панели TERMINAL (Терминал) введите следующую команду:

```
sqlite3 Northwind.db -init Northwind.sql
```

4. Наберитесь терпения, так как эта команда может занять некоторое время для создания структуры базы данных.

```
-- Loading resources from Northwind.sql
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
sqlite>
```

5. Для выхода из командного режима SQLite нажмите сочетание клавиш Ctrl+D в macOS или Ctrl+C в Windows.
6. В меню File (Файл) закройте папку PracticalApps.
7. В программе Visual Studio Code выберите File ▸ Save Workspace As (Файл ▸ Сохранить рабочую область как), введите имя `PracticalApps`, выберите папку PracticalApps и нажмите кнопку Save (Сохранить).
8. Создайте в папке PracticalApps подпапку NorthwindEntitiesLib.
9. Добавьте в рабочую область папку NorthwindEntitiesLib.
10. Выберите меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку NorthwindEntitiesLib.
11. На панели TERMINAL (Терминал) введите следующую команду: `dotnet new classlib`.

12. Откройте файл NorthwindEntitiesLib.csproj и добавьте две ссылки на пакеты для EF Core для SQLite и поддержки времени разработки.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="5.0.0" />
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Design"
      Version="5.0.0" />
  </ItemGroup>
</Project>
```

13. На панели TERMINAL (Терминал) загрузите указанный пакет и скомпилируйте текущий проект:

```
dotnet build
```

14. Удалите файл Class1.cs.
15. На панели TERMINAL (Терминал) сгенерируйте модели классов сущностей для всех таблиц:

```
dotnet ef dbcontext scaffold "Filename=../Northwind.db" Microsoft.
EntityFrameworkCore.Sqlite --namespace Packt.Shared --data-annotations
--context Northwind
```

Обратите внимание на следующее.

- Команда для выполнения: dbcontext scaffold.
- Строка подключения: "Filename=../Northwind.db".
- Поставщик базы данных: Microsoft.EntityFrameworkCore.Sqlite.
- Пространство имен: --namespace Packt.Shared.
- Чтобы использовать аннотации данных, а также Fluent API: --data-annotations.
- Чтобы переименовать контекст из [*database_name*]Context: --context Northwind.

16. Обратите внимание на сообщения и предупреждения сборки, как показано в следующем выводе:

```
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string,
you should move it out of source code. You can avoid scaffolding the
connection string by using the Name= syntax to read it from configuration
- see https://go.microsoft.com/fwlink/?linkid=2131148. For more
guidance on storing connection strings, see http://go.microsoft.com/
fwlink/?LinkId=723263.
```

Улучшение сопоставления классов и таблиц вручную

Затем, чтобы улучшить отображение модели, сделаем небольшие изменения.

Мы внесем изменения в следующие классы сущностей: `Category.cs`, `Customer.cs`, `Employee.cs`, `Order.cs`, `OrderDetail.cs`, `Product.cs`, `Shipper.cs`, `Supplier.cs` и `Territory.cs`.

1. Измените все свойства первичного или внешнего ключа, чтобы использовать стандартное именование, например измените `CategoryId` на `CategoryID`. Это позволит вам также удалить атрибут `[Column]`, сопоставляющий свойство со столбцом в таблице:

```
// до
[Key]
[Column("CategoryID")]
public long CategoryId { get; set; }

// после
[Key]
public long CategoryID { get; set; }
```

2. Дополните все свойства строки атрибутом `[StringLength]`, чтобы ограничить максимальное количество символов, разрешенное для соответствующего поля, используя информацию в атрибуте `[Column]`:

```
// до
[Required]
[Column(TypeName = "nvarchar (15)")]
public string CategoryName { get; set; }

// после
[Required]
[Column(TypeName = "nvarchar (15)")]
[StringLength(15)]
public string CategoryName { get; set; }
```

3. Откройте файл `Customer.cs` и добавьте регулярное выражение для проверки значения первичного ключа, чтобы разрешить использование только заглавных западных символов:

```
[Key]
[Column(TypeName = "nchar (5)")]
[StringLength(5)]
[RegularExpression("[A-Z]{5}")]
public string CustomerID { get; set; }
```

4. Измените любые свойства даты/времени, например, в файле `Employee.cs`, чтобы вместо массива байтов использовать тип данных `DateTime`, допускающий значение `NULL`:

```
// до
[Column(TypeName = "datetime")]
public byte[] BirthDate { get; set; }

// после
[Column(TypeName = "datetime")]
public DateTime? BirthDate { get; set; }
```

5. Измените любые свойства денежных единиц, например, в файле `Order.cs`, чтобы вместо массива байтов использовать десятичное число, допускающее значение `NULL`:

```
// до
[Column(TypeName = "money")]
public byte[] Freight { get; set; }

// после
[Column(TypeName = "money")]
public decimal? Freight { get; set; }
```

6. Измените любые битовые свойства, например, в файле `Product.cs`, чтобы вместо массива байтов использовать логическое значение:

```
// до
[Required]
[Column(TypeName = "bit")]
public byte[] Discontinued { get; set; }

// после
[Required]
[Column(TypeName = "bit")]
public bool Discontinued { get; set; }
```

Теперь, когда у нас есть библиотека классов для классов сущностей, мы можем создать библиотеку классов для контекста базы данных.

Создание библиотеки классов для контекста базы данных Northwind

Теперь вы определите библиотеку классов контекста базы данных.

1. Создайте в папке PracticalApps подпапку NorthwindContextLib.
2. Добавьте в рабочую область папку NorthwindContextLib.
3. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку NorthwindContextLib.
4. На панели TERMINAL (Терминал) введите следующую команду: `dotnet new classlib`.
5. Выберите View ▸ Command Palette (Вид ▸ Панель команд), а затем OmniSharp: Select Project. Выберите проект `NorthwindContextLib`.

6. Отредактируйте файл `NorthwindContextLib.csproj` и добавьте ссылку на проект `NorthwindEntitiesLib` и базовый пакет Entity Framework для SQLite, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindEntitiesLib\NorthwindEntitiesLib.csproj" />
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.SQLite"
      Version="5.0.0" />
  </ItemGroup>
</Project>
```

7. В проекте `NorthwindContextLib` удалите файл класса `Class1.cs`.
8. Переместите файл `Northwind.cs` из папки NorthwindEntitiesLib в папку NorthwindContextLib.
9. В файле `Northwind.cs` измените операторы, чтобы удалить предупреждение компилятора о строке подключения. Затем удалите операторы Fluent API для проверки и определения ключей и отношений, за исключением `OrderDetail`, поскольку многополевые первичные ключи можно определить только с помощью Fluent API. Это потому, что они дублируют работу, выполняемую атрибутами в классах модели сущностей:

```
using Microsoft.EntityFrameworkCore;

#nullable disable

namespace Packt.Shared
{
  public partial class Northwind : DbContext
  {

  public Northwind()
  {
  }

  public Northwind(DbContextOptions<Northwind> options)
    : base(options)
  {
  }
  public virtual DbSet<Category> Categories { get; set; }
  public virtual DbSet<Customer> Customers { get; set; }
  public virtual DbSet<Employee> Employees { get; set; }
  public virtual DbSet<EmployeeTerritory> EmployeeTerritories
    { get; set; }
  public virtual DbSet<Order> Orders { get; set; }
```

```
    public virtual DbSet<OrderDetail> OrderDetails { get; set; }
    public virtual DbSet<Product> Products { get; set; }
    public virtual DbSet<Shipper> Shippers { get; set; }
    public virtual DbSet<Supplier> Suppliers { get; set; }
    public virtual DbSet<Territory> Territories { get; set; }

    protected override void OnConfiguring(
      DbContextOptionsBuilder optionsBuilder)
    {
      if (!optionsBuilder.IsConfigured)
      {
        optionsBuilder.UseSqlite("Filename=../Northwind.db");
      }
    }

    protected override void OnModelCreating(
      ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<OrderDetail>(entity =>
      {
        entity.HasKey(x => new { x.OrderID, x.ProductID });

        entity.HasOne(d => d.Order)
          .WithMany(p => p.OrderDetails)
          .HasForeignKey(x => x.OrderID)

          .OnDelete(DeleteBehavior.ClientSetNull);

        entity.HasOne(d => d.Product)
                    .WithMany(p => p.OrderDetails)
                    .HasForeignKey(x => x.ProductID)
                    .OnDelete(DeleteBehavior.ClientSetNull);
      });
        modelBuilder.Entity<Product>()
          .Property(product => product.UnitPrice)
          .HasConversion<double>();

        OnModelCreatingPartial(modelBuilder);
      }
      partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
    }
  }
```

10. На панели TERMINAL (Терминал) скомпилируйте библиотеки классов для проверки ошибок с помощью следующей команды: `dotnet build`

Во всех проектах, таких как сайты, которые должны работать с базой данных Northwind, мы установим строку подключения к базе данных (так что это не нужно делать в классе `Northwind`). Однако, чтобы все работало, класс, наследуемый от `DbContext`, должен содержать конструктор с параметром `DbContextOptions`.

Резюме

Вы познакомились с отдельными моделями приложений, которые можете использовать для создания практических приложений с помощью C# и .NET Core, и создали две библиотеки классов для определения сущностных моделей данных для работы с базой данных Northwind.

Из следующих глав вы получите информацию о том, как разработать:

- простые сайты при помощи статических HTML-страниц и динамических веб-страниц Razor;
- сложные веб-приложения, используя паттерн проектирования *Model-View-Controller (MVC)*;
- сложные веб-приложения с контентом, которым могут управлять конечные пользователи с помощью *системы управления контентом* (Content Management System, CMS);
- веб-сервисы, пригодные для вызова любой платформой, которая может создать HTTP-запрос, и клиенты этих сервисов;
- интеллектуальные приложения, использующие машинное обучение;
- компоненты веб-интерфейса пользователя Blazor, которые могут размещаться на сервере или в браузере;
- кросс-платформенные мобильные приложения с Xamarin.Forms.

15 Разработка сайтов с помощью ASP.NET Core Razor Pages

Данная глава посвящена созданию сайтов с современной HTTP-архитектурой на сервере с применением Microsoft ASP.NET Core. Вы научитесь создавать простые сайты, задействуя функцию Razor Pages платформы ASP.NET Core, представленную в .NET Core 2.0, и функцию библиотеки классов Razor, представленную в .NET Core 2.1.

В этой главе:

- веб-разработка;
- обзор ASP.NET Core;
- технология Razor Pages;
- использование Entity Framework Core совместно с ASP.NET Core;
- применение библиотек классов Razor;
- настройка служб и конвейера HTTP-запросов.

Веб-разработка

Разработка для Всемирной паутины — это разработка с *HTTP* (Hypertext Transfer Protocol, протокол передачи гипертекста).

Протокол передачи гипертекста

Для связи с веб-сервером клиент, также известный как *пользовательский агент*, совершает вызовы через Интернет, используя протокол HTTP. Это своего рода технический фундамент *Всемирной паутины*. Таким образом, говоря о веб-приложениях или веб-сервисах, мы имеем в виду, что они используют HTTP для связи между клиентом (обычно браузером) и сервером.

Клиент отправляет HTTP-запрос ресурса, например страницы, идентифицируемого по *URL* (Uniform Resource Locator, единый указатель ресурса), и сервер отправляет HTTP-ответ, как показано на рис. 15.1.

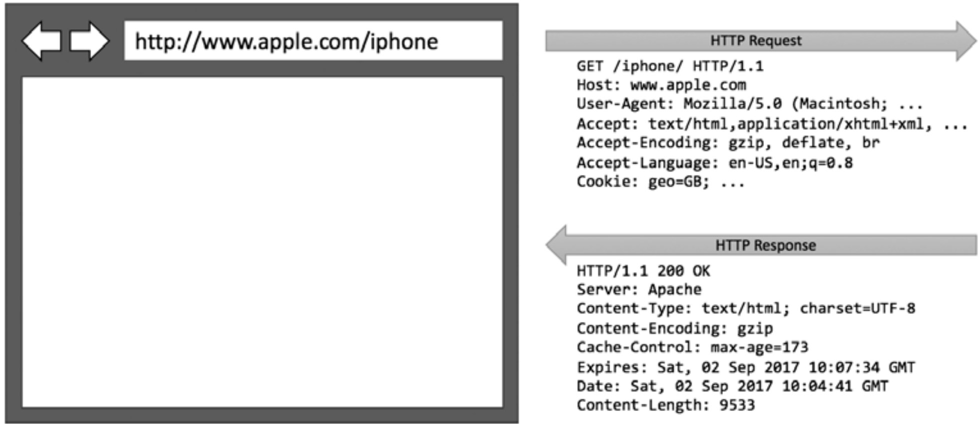


Рис. 15.1. HTTP-запрос и ответ

Для записи запросов и ответов можно использовать Google Chrome и другие браузеры.

Google Chrome доступен для большего количества операционных систем, чем любой другой браузер, и содержит эффективные встроенные инструменты разработчика, поэтому лучше всего подойдет для тестирования веб-приложений. Всегда тестируйте свое веб-приложение в Google Chrome и по крайней мере в двух других браузерах, например Firefox и Safari для macOS и iPhone. Microsoft Edge перейдет от использования собственного механизма визуализации Microsoft к применению Chromium в 2019 году, поэтому не очень важно его тестировать. Если браузер Microsoft Internet Explorer и используется, то обычно только внутри организаций и во внутренней электронной сети.

Рассмотрим применение браузера Google Chrome для выполнения HTTP-запросов.

1. Запустите браузер Google Chrome.
2. Чтобы отобразить инструменты разработчика, выполните следующие действия:
   - в операционной системе macOS нажмите сочетание клавиш Alt+Cmd+I;
   - в операционной системе Windows нажмите клавишу F12 или сочетание клавиш Ctrl+Shift+I.
3. Перейдите на вкладку Network (Сеть). Браузер Google Chrome должен немедленно начать запись сетевого трафика между вашим браузером и любыми веб-серверами (рис. 15.2).

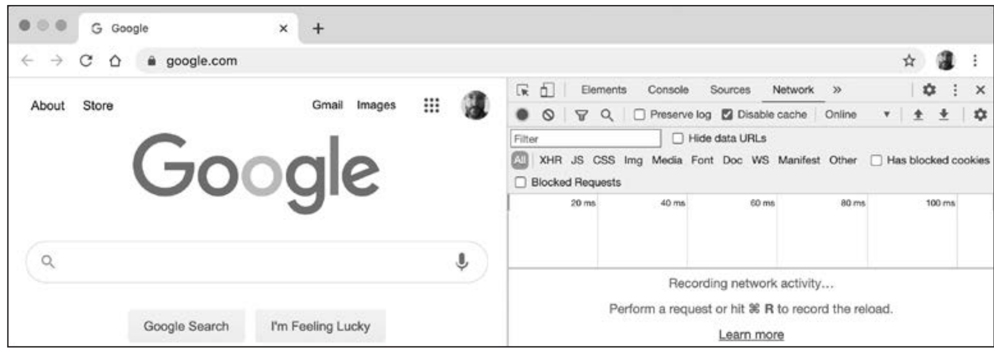


Рис. 15.2. Google Chrome записывает сетевой трафик

4. В адресной строке браузера Google Chrome введите следующий URL: https://dotnet.microsoft.com/learn/aspnet.
5. На панели Developer tools (Инструменты разработчика) прокрутите вверх список записанных запросов и выделите первую запись (рис. 15.3).

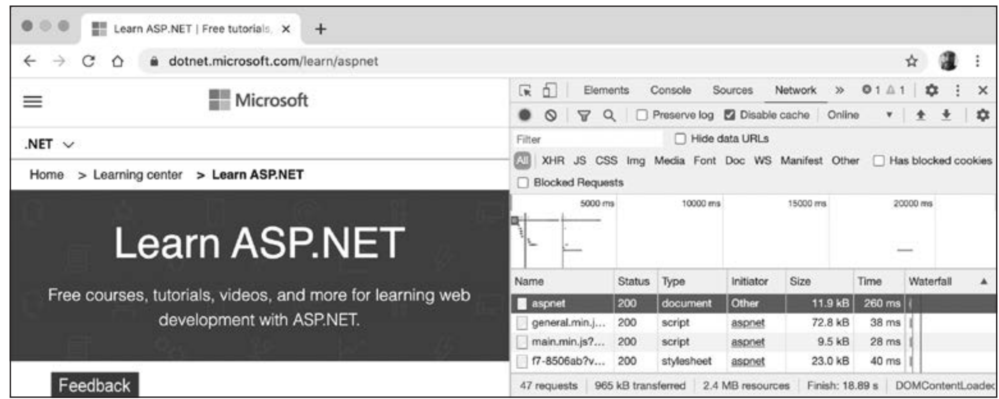


Рис. 15.3. Записанные запросы

6. В правой части панели щелкните кнопкой мыши на вкладке Headers (Заголовки). Вы должны увидеть подробную информацию о запросе и ответе (рис. 15.4).

   Обратите внимание на следующие моменты.

   - Применен *метод запроса* `GET`. Другие методы, определяемые протоколом HTTP, включают `POST`, `PUT`, `DELETE`, `HEAD` и `PATCH`.
   - Получен *код состояния* `200 OK`. Это значит, сервер обнаружил ресурс, запрошенный браузером, и вернул его в ответе. Другие коды состояния, которые вы можете увидеть в ответе на запрос `GET`, включают `301 Moved Permanently`

(перемещено навсегда), `400 Bad Request` (плохой, неверный запрос), `401 Unauthorized` (не авторизован) и `404 Not Found` (не найден).

- *Заголовки запроса*, отправляемые браузером на веб-сервер, включают в себя следующие параметры.
  - Параметр *accept*, перечисляющий форматы ресурсов, принимаемые браузером. В этом случае браузер сообщает, что поддерживает форматы HTML, XHTML, XML и некоторые форматы изображений, но будет принимать все другие файлы `*/*`. Веса, также известные как качественные значения, по умолчанию равны 1,0. XML указан с качественным значением 0,9, поэтому является менее предпочтительным, чем HTML или XHTML. Все другие типы файлов имеют качественное значение 0,8, поэтому наименее предпочтительны.
  - Параметр *accept-encoding*, в котором перечислены алгоритмы сжатия, поддерживаемые браузером. В этом случае браузер поддерживает алгоритмы сжатия GZIP, DEFLATE и Brotli.
  - Параметр *accept-language*, перечисляющий предпочитаемые браузером человеческие языки. В этом случае английский язык США имеет качественное значение по умолчанию 1,0, а любой диалект английского языка имеет явно заданное качественное значение 0,9.
- Файл cookie сервиса Google Analytics с именем `_ga` передается на сервер, чтобы сайт мог отслеживать мои действия, наряду с множеством других файлов cookie.
- Сервер отправил обратно ответ с HTML-страницей, сжатый с помощью алгоритма GZIP, поскольку знает, что клиент способен распаковать этот формат.

7. Закройте браузер Google Chrome.

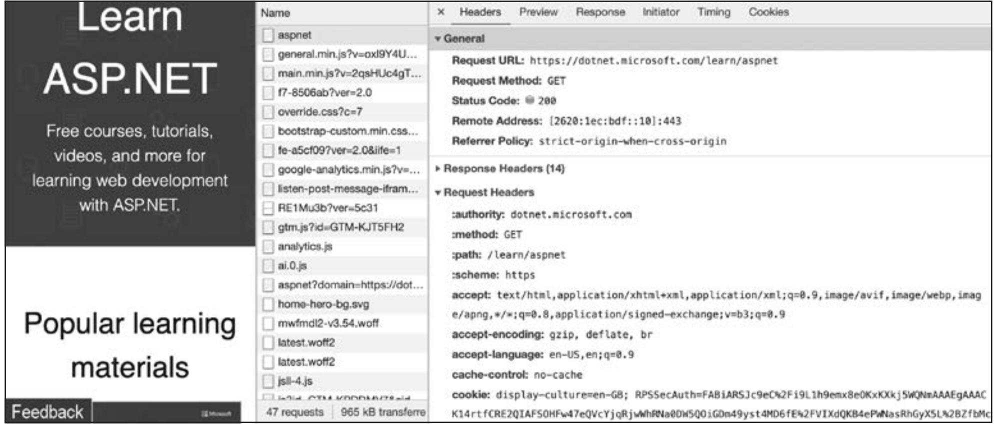


Рис. 15.4. Запрос и ответ

Клиентская веб-разработка

При создании веб-приложений разработчику необходимо знать больше, чем просто C# и .NET Core. На стороне клиента (то есть в браузере) вы будете задействовать комбинацию следующих технологий:

- *HTML5* — используется для определения контента и структуры веб-страницы;
- *CSS3* — служит для настройки стилей, применяемых к элементам на веб-странице;
- *JavaScript* — используется для кодирования любой бизнес-логики, необходимой на веб-странице, например проверки правильности ввода формы или выполнения вызовов веб-сервиса для получения дополнительных данных, необходимых для веб-страницы.

Несмотря на то что HTML5, CSS3 и JavaScript — фундаментальные компоненты веб-разработки, существует множество дополнительных технологий, которые могут повысить ее эффективность, в том числе Bootstap и CSS-препроцессоры, такие как SASS и LESS для стилей, язык TypeScript от Microsoft для написания более надежного кода и библиотеки JavaScript, такие как jQuery, Angular, React и Vue. Все перечисленные высокоуровневые технологии в конечном итоге переводятся или компилируются в три основные технологии, поэтому работают во всех современных браузерах.

В рамках процесса сборки и развертывания вы, вероятно, будете использовать такие технологии, как Node.js; *Node Package Manager* (NPM, стандартный менеджер пакетов) и Bower — менеджеры пакетов на стороне клиента; Webpack — популярный модульный компоновщик, инструмент для компиляции, преобразования и объединения исходных файлов сайтов.

Эта книга посвящена C# и .NET Core, поэтому вы ознакомитесь с некоторыми основами веб-разработки, но получить более подробную информацию можно в книге «HTML5 и CSS3: создание адаптивных сайтов», перейдя по ссылке https://www.packtpub.com/product/html5-and-css3-building-responsive-websites/9781787124813h.

Обзор ASP.NET Core

Microsoft ASP.NET Core — следующий этап развития технологий корпорации Microsoft, используемых для разработки веб-приложений и сервисов, которые эволюционировали на протяжении многих лет.

- *ASP* (Active Server Pages — активные серверные страницы) — технология, реализованная в 1996 году и ставшая первой попыткой корпорации Microsoft

создать платформу для динамического выполнения кода веб-приложений на стороне сервера. ASP-файлы содержат смесь HTML и выполняемого на сервере кода на языке VBScript.

- *ASP.NET Web Forms* — эта технология была представлена в 2002 году вместе с платформой .NET Framework и предназначена для того, чтобы разработчики, незнакомые с веб-программированием, но знающие такие языки, как Visual Basic, могли быстро создавать веб-приложения, перетаскивая и удаляя визуальные компоненты и записывая код событий на Visual Basic или C#. Web Forms поддерживает хостинг только на серверах под управлением операционной системы Windows, но по-прежнему используется в таких продуктах, как Microsoft SharePoint. Применение данной технологии при разработке новых веб-проектов следует избегать в пользу платформы ASP.NET Core.
- *WCF* (Windows Communication Foundation) — платформа, выпущенная в 2006 году и позволяющая разработчикам создавать сервисы SOAP и REST. SOAP — технология мощная, но сложная, поэтому ее следует избегать, если нет необходимости в использовании расширенных функций, таких как распределенные транзакции и сложные топологии обмена сообщениями.
- *ASP.NET MVC* — платформа, выпущенная в 2009 году и реализующая модель четкого разделения задач веб-разработчиков между *моделями*, временно хранящими данные, *представлениями*, которые отображают эти данные в различном виде в пользовательском интерфейсе, и *контроллерами*, осуществляющими выборку модели и передачу ее представлению. Такое разделение положительно сказывается при многократном использовании и модульном тестировании.
- *ASP.NET Web API* — инструмент, выпущенный в 2012 году и позволяющий разработчикам создавать HTTP-сервисы, также известные как REST, которые масштабируются проще и лучше, чем SOAP-сервисы.
- *ASP.NET SignalR* — технология, выпущенная в 2013 году и реализующая методы коммуникации в реальном времени в веб-приложениях путем абстрагирования вспомогательных технологий и методов наподобие веб-сокетов и длинных запросов. Данная технология позволяет разрабатывать такие функции сайта, как чат в режиме реального времени или обновление чувствительных к времени данных (например, биржевых курсов) в самых разных браузерах, даже если они не поддерживают базовую технологию, такую как веб-сокеты.
- *ASP.NET Core* — платформа, выпущенная в 2016 году. Она объединяет технологии MVC, Web-API и SignalR и работает на платформе .NET Core, благодаря чему является кросс-платформенной. В ASP.NET Core существует множество шаблонов проектов, которые помогут вам начать работу с поддерживаемыми технологиями.

Для разработки веб-приложений и сервисов следует выбирать платформу ASP.NET Core, поскольку она включает в себя современные и кросс-платформенные веб-технологии.

Версии ASP.NET Core от 2.0 до 2.2 могут работать на версии .NET Framework 4.6.1 или более поздней (только для операционной системы Windows), а также на .NET Core 3.0 или более поздней версии (кросс-платформенной). ASP.NET Core 5 поддерживает лишь .NET Core 5.

Классический ASP.NET против современного ASP.NET Core

До сих пор ASP.NET была построена на основе крупной сборки `System.Web.dll` на платформе .NET Framework и была тесно связана с веб-сервером Microsoft для операционной системы Windows, известным как *Internet Information Services (IIS)*. За прошедшие годы в сборке скопилось множество функций, не подходящих для современной кросс-платформенной разработки.

ASP.NET Core — попытка глобального пересмотра платформы ASP.NET. Основной принцип заключается в удалении зависимостей от сборки `System.Web.dll` и IIS и реализации платформы в виде модульных легковесных пакетов, как и остальная часть .NET Core.

Вы можете разрабатывать и запускать приложения ASP.NET Core независимо от платформы в операционной системе Windows, macOS и Linux. Корпорация Microsoft даже разработала кросс-платформенный суперпроизводительный веб-сервер *Kestrel*, а исходный код всего стека полностью открыт.

Более подробную информацию о Kestrel, включая его поддержку HTTP/2, можно получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/servers/kestrel.

В версии ASP.NET Core 2.2 или более поздних версиях по умолчанию используется новая модель внутрипроцессного хостинга. Это дает повышение производительности на 400 % при хостинге в Microsoft IIS. Однако корпорация Microsoft по-прежнему рекомендует использовать Kestrel в целях еще большей производительности.

Разработка проекта ASP.NET Core

Мы создадим проект ASP.NET Core, отображающий список поставщиков из базы данных Northwind.

Инструмент `dotnet` распологает множеством шаблонов проектов, выполняющих за вас значительную часть работы. Однако зачастую трудно определить самые подходящие инструменты для данной ситуации, поэтому, чтобы во всем разобраться, мы начнем с самого простого веб-шаблона и постепенно добавим функции.

1. В существующей папке PracticalApps создайте подпапку NorthwindWeb и добавьте ее в рабочую область PracticalApps.
2. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку NorthwindWeb.
3. На панели TERMINAL (Терминал) для создания сайта ASP.NET Core Empty введите следующую команду: dotnet new web.
4. На панели TERMINAL (Терминал) для восстановления пакетов и компиляции сайта введите следующую команду: dotnet build.
5. Отредактируйте файл NorthwindWeb.csproj и обратите внимание, что в качестве SDK используется Microsoft.NET.Sdk.Web:

   ```
   <Project Sdk="Microsoft.NET.Sdk.Web">
     <PropertyGroup>
       <TargetFramework>net5.0</TargetFramework>
     </PropertyGroup>
   </Project>
   ```

   В версии ASP.NET Core 1.0 вам пришлось бы включать большое количество ссылок. В ASP.NET Core 2.0 вам потребуется ссылка на пакет Microsoft.AspNetCore.All. Начиная с версии ASP.NET Core 3.0 и выше, достаточно просто использовать веб-SDK.
6. Откройте файл Program.cs и обратите внимание на следующие моменты:
   - сайт похож на консольное приложение, в качестве точки входа которого используется метод Main;
   - сайт содержит метод CreateHostBuilder, определяющий класс запуска, служащий для настройки сайта, который затем создается и запускается, как показано ниже:

   ```
   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>();
         });
   }
   ```
7. Откройте файл Startup.cs и обратите внимание на следующие два метода:
   - метод ConfigureServices в настоящее время пуст. Мы будем использовать его позже, чтобы добавить такие сервисы, как Razor Pages и контекст базы данных для работы с базой данных Northwind;

- в настоящий момент метод `Configure` выполняет три действия: во-первых, он настраивает то, что при разработке любые необработанные исключения будут отображаться в окне браузера, чтобы разработчик мог увидеть их подробности; во-вторых, применяет маршрутизацию; и в-третьих, подключает конечные точки для ожидания запросов, а затем асинхронно отвечает на каждый HTTP-запрос `GET`, возвращая простой текст `Hello World!`, как показано ниже:

```
public class Startup
{
  // Этот метод вызывается средой выполнения.
  // Используйте его для добавления сервисов в контейнер.
  public void ConfigureServices(IServiceCollection services)
  {
  }

  // Этот метод вызывается средой выполнения.
  // Используйте его для настройки конвейера HTTP-запроса.
  public void Configure(
    IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
    }

    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
      endpoints.MapGet("/", async context =>
      {
        await context.Response.WriteAsync("Hello World!");
      });
    });
  }
}
```

8. Закройте файл класса `Startup.cs`.

Как ASP.NET Core обнаружит, что мы работаем в режиме разработки, чтобы метод `IsDevelopment` возвращал значение `true`? Давайте выясним.

Тестирование и защита сайта

Проверим функциональность проекта пустого сайта ASP.NET Core. Кроме того, в целях обеспечения конфиденциальности мы включим шифрование всего трафика между браузером и веб-сервером путем переключения с HTTP на HTTPS. HTTPS — это безопасная зашифрованная версия HTTP.

1. На панели TERMINAL (Терминал) введите команду `dotnet run` и обратите внимание, что веб-сервер начал прослушивать порты 5000 и 5001, а среда размещения — это разработка:

```
info: Microsoft.Hosting.Lifetime[0]
  Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
  Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
  Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
  Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
  Content root path: /Users/markjprice/Code/PracticalApps/NorthwindWeb
```

2. Запустите браузер Google Chrome.
3. Введите адрес http://localhost:5000/ и обратите внимание, что ответ `Hello World!` — это обычный текст с кросс-платформенного веб-сервера Kestrel (рис. 15.5).

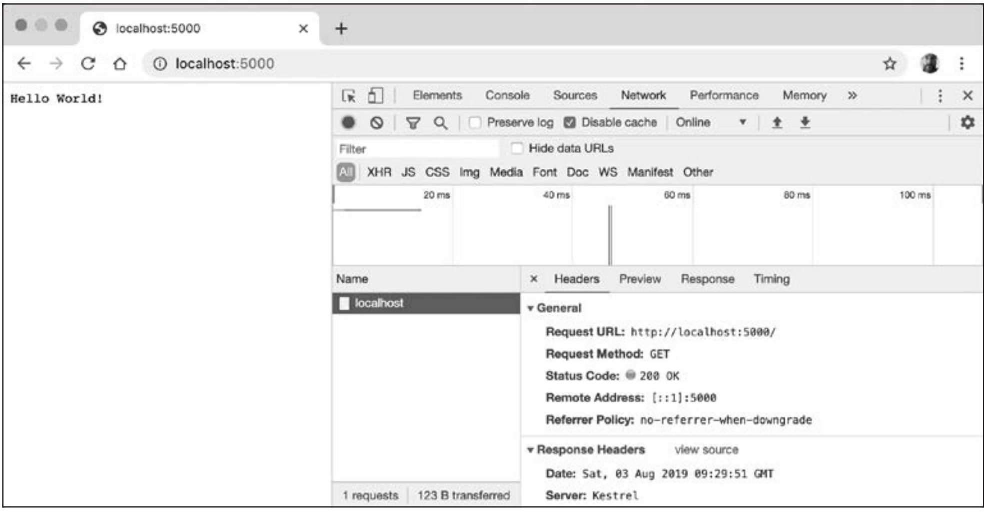


Рис. 15.5. Ответ от http://localhost:5000/

4. Введите адрес https://localhost:5001/ и обратите внимание, что ответ — это ошибка конфиденциальности (рис. 15.6).

   Это связано с тем, что мы не настроили сертификат, которому браузер может доверять, для шифрования и дешифрования HTTP-трафика (отсутствие ошибки означает, что мы уже настроили сертификат). В производственной среде за обеспечение защиты ответственности и техническую поддержку вам пришлось бы заплатить компании, такой как Verisign или другой.

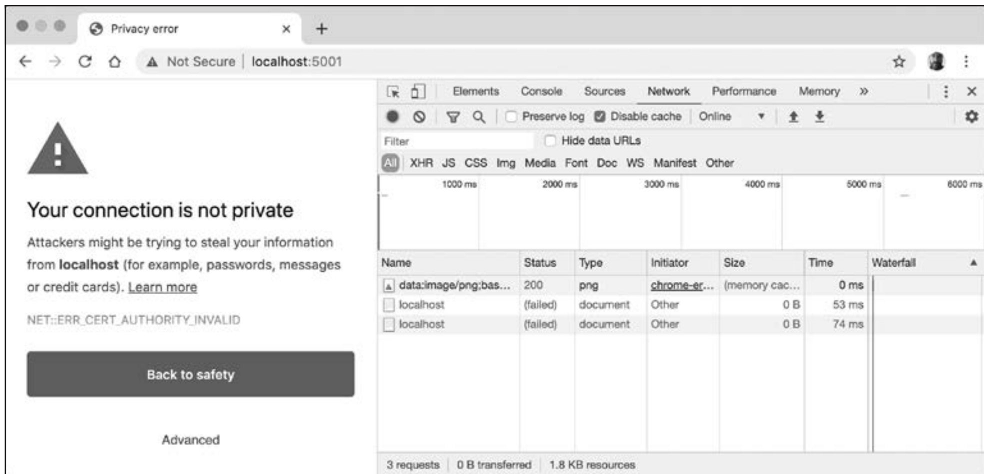


Рис. 15.6. Ошибка конфиденциальности, означающая, что не был включен сертификат SSL

Если вы используете вариант Linux, который не может создавать самозаверяющие сертификаты, и вы готовы каждые 90 дней подавать заявку на новый сертификат, то можете получить бесплатный сертификат по следующей ссылке: https://letsencrypt.org.

В процессе разработки вы можете дать операционной системе указание доверять временному сертификату разработки, предоставленному ASP.NET Core.

5. На панели TERMINAL (Терминал) для остановки веб-сервера нажмите сочетание клавиш Ctrl+C.
6. На панели TERMINAL (Терминал) введите команду `https dev-certs https --trust` и обратите внимание на сообщение Trusting the HTTPS development certificate was requested (Было запрошено доверие HTTPS-сертификату для разработки). Вам могут предложить ввести пароль, а правильный сертификат HTTPS может уже быть установлен.
7. Если Google Chrome все еще работает, то с целью убедиться в том, что он считал новый сертификат, закройте и перезапустите браузер.
8. В файле `Startup.cs` в методе `Configure` добавьте оператор `else`, чтобы включить HSTS (если он не находится в разработке), как показано ниже (выделено полужирным шрифтом):

```
if (env.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
}
```

```
else
{
  app.UseHsts();
}
```

HTTP Strict Transport Security (HSTS) — это механизм дополнительного усовершенствования безопасности. Если сайт указывает его и браузер поддерживает, то механизм принуждает вести все общение через HTTPS и не позволяет пользователю применять недостоверные или недействительные сертификаты.

9. Для перенаправления HTTP-запросов в HTTPS добавьте после вызова `app.UseRouting` следующий оператор:

```
app.UseHttpsRedirection();
```

10. На панели TERMINAL (Терминал) для запуска веб-сервера введите следующую команду: `dotnet run`.
11. В адресной строке браузера Google Chrome введите следующий URL: http://localhost:5000/. Вы должны увидеть, что сервер отвечает `307 Temporary Redirect` (временное перенаправление) на порт 5001 и что сертификат теперь действителен и ему можно доверять (рис. 15.7).

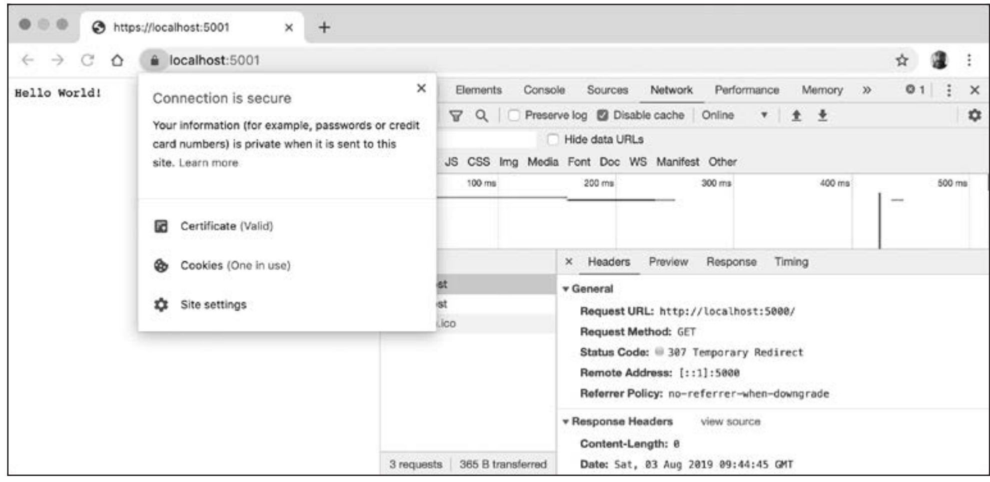


Рис. 15.7. Теперь соединение защищено действующим сертификатом

12. Закройте браузер Google Chrome.
13. На панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C, чтобы остановить веб-сервер.

Не забудьте останавливать веб-сервер Kestrel каждый раз, когда завершаете тестирование сайта.

Управление средой хостинга

ASP.NET Core может считывать переменные среды, чтобы определить, какую среду размещения использовать, например DOTNET_ENVIRONMENT или ASPNETCORE_ENVIRONMENT, когда вызывается метод ConfigureWebHostDefaults, как в файле Program.cs.

Во время локальной разработки эти настройки могут быть переопределены.

1. В папке NorthwindWeb разверните папку Properties, откройте файл launchSettings.json и обратите внимание на профиль NorthwindWeb, который устанавливает среду размещения в Development, как показано ниже (выделено полужирным шрифтом):

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:56111",
      "sslPort": 44329
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "NorthwindWeb": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

2. Измените среду размещения на Production.

3. На панели TERMINAL (Терминал) с помощью команды dotnet run запустите сайт и обратите внимание, что среда размещения — Production, как показано в следующих выходных данных:

```
info: Microsoft.Hosting.Lifetime[0]
  Hosting environment: Production
```

4. На панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C, чтобы остановить веб-сервер.
5. В файле `launchSettings.json` снова измените среду размещения на Development.

Более подробно о работе со средами размещения ASP.NET Core вы можете прочитать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/environments.

Включение статических файлов и файлов по умолчанию

Сайт, возвращающий только одно текстовое сообщение, не очень полезен!

Как минимум он должен возвращать статические HTML-страницы, CSS, которые веб-страницы будут использовать для стилизации, и любые другие статические ресурсы, такие как изображения и видео.

Создадим папку для статических ресурсов сайта и основную страницу индекса, использующую для стилизации Bootstrap.

Веб-технологии, такие как Bootstrap, для эффективной доставки своих исходных файлов по всему миру обычно используют сеть доставки контента (Content Delivery Network, CDN). Более подробную информацию о CDN можно получить на сайте en.wikipedia.org/wiki/Content_delivery_network.

1. Создайте в папке NorthwindWeb папку wwwroot.
2. Добавьте в папку wwwroot новый файл `index.html`.
3. Для стилизации отредактируйте содержимое файла, добавив ссылку на Bootstrap, размещенную на CDN, и задействовав лучшие современные практики, такие как настройка области просмотра:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <!-- Required meta tags -->
  <meta charset="utf-8" />
```

```
    <meta name="viewport" content=
      "width=device-width, initial-scale=1, shrink-to-fit=no" />

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/
    bootstrap/4.5.2/css/bootstrap.min.css" integrity="sha384-JcKb8q3iqJ61gNV9K
    Gb8thSsNjpSL0n8PARn9HuZOnIxN0hoP+VmmDGMN5t9UJ0Z" crossorigin="anonymous">

    <title>Welcome ASP.NET Core!</title>
  </head>

  <body>
    <div class="container">
      <div class="jumbotron">
        <h1 class="display-3">Welcome to Northwind!</h1>
        <p class="lead">We supply products to our customers.</p>
        <hr />
        <h2>This is a static HTML page.</h2>
        <p>Our customers include restaurants, hotels and cruise lines.</p>
        <p>
          <a class="btn btn-primary"
            href="https://www.asp.net/">Learn more</a>
        </p>
      </div>
    </div>
  </body>

  </html>
```

Чтобы получить последний элемент <link> для Bootstrap, скопируйте и вставьте его со страницы Getting Started — Introduction по следующей ссылке: https://getbootstrap.com/.

Если бы вы запустили сайт прямо сейчас и ввели в адресной строке браузера следующий адрес: http://localhost:5000/index.html, то сайт вернул бы ошибку `404 Not Found`, тем самым сообщив, что веб-страница не найдена. Чтобы он мог возвращать статические файлы, такие как `index.html`, необходимо явно настроить данную функцию.

Даже после включения статических файлов, если вы запустите веб-приложение и введете в поле адреса http://localhost:5000/, сайт выдаст ошибку `404 Not Found`, поскольку веб-сервер не знает, что возвращать по умолчанию, если никакой именованный файл не запрашивается.

Включим статические файлы и явно сконфигурируем файлы по умолчанию и изменим зарегистрированный URL-путь, который возвращает `Hello World!`.

1. В файле `Startup.cs` в методе `Configure` закомментируйте оператор, который направляет любой запрос `GET` на возврат ответа `Hello World!` в виде простого текста. Затем добавьте операторы для включения статических файлов и файлов по умолчанию, как показано ниже (выделено полужирным шрифтом):

   ```
   public void Configure(
     IApplicationBuilder app, IWebHostEnvironment env)
   {
     if (env.IsDevelopment())
     {
       app.UseDeveloperExceptionPage();
     }
     else
     {
       app.UseHsts();
     }

     app.UseRouting();

     app.UseHttpsRedirection();
     app.UseDefaultFiles(); // index.html, default.html и т.д.
     app.UseStaticFiles();

     app.UseEndpoints(endpoints =>
     {
       endpoints.MapGet("/hello", async context =>
       {
         await context.Response.WriteAsync("Hello World!");
       });
     });
   }
   ```

   Вызов метода `UseDefaultFiles` должен следовать перед вызовом метода `UseStaticFiles`, в противном случае метод не сработает!

2. Запустите сайт, введя на панели TERMINAL (Терминал) команду `dotnet run`.
3. В адресной строке браузера Google Chrome введите следующий URL: http://localhost:5000/. Обратите внимание: вы перенаправлены на HTTPS-адрес на порте 5001 и файл `index.html` возвращается моментально, поскольку является одним из возможных файлов по умолчанию для этого сайта.

Если все веб-страницы — статические, то есть изменяются только вручную веб-редактором, значит, работа по разработке нашего сайта завершена. Однако почти всем сайтам необходим динамический контент, то есть веб-страница, генерируемая во время работы сайта путем выполнения программного кода.

Проще всего добиться этого результата, задействовав функцию ASP.NET Core под названием *Razor Pages*.

Технология Razor Pages

Технология Razor Pages позволяет разработчику легко смешивать разметку HTML с операторами кода C#. Вот почему они используют расширение файла `.cshtml`.

По умолчанию ASP.NET Core ищет Razor Pages в папке под названием Pages.

Включение Razor Pages

Изменим статическую HTML-страницу на динамическую Razor Page, а затем добавим и включим сервис Razor Pages.

1. В проекте `NorthwindWeb` создайте папку Pages.
2. Скопируйте файл `index.html` в папку Pages.
3. Переименуйте расширение файла из `.html` в `.cshtml`.
4. Удалите элемент `<h2>`, который означает, что это статическая HTML-страница.
5. В файле `Startup.cs` в методе `ConfigureServices` укажите операторы для добавления Razor Pages и связанные с ними сервисы, такие как привязка модели, авторизация, защита от подделки, представления и «тег-хелперы», как показано ниже (выделено полужирным шрифтом):

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddRazorPages();
}
```

6. В файле `Startup.cs` в методе `Configure` в конфигурации использования конечных точек добавьте инструкцию использовать `MapRazorPages`, как показано ниже (выделено полужирным шрифтом):

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapRazorPages();

  endpoints.MapGet("/hello", async context =>
  {
    await context.Response.WriteAsync("Hello World!");
  });
});
```

Определение Razor Pages

Синтаксис Razor в HTML-разметке веб-страницы обозначается символом `@`. Страницу, написанную с помощью Razor Pages, можно описать следующим образом.

- В начале файла требуется директива `@page`.
- Может содержаться раздел `@functions`, определяющий следующие моменты:
  - свойства для хранения значений данных, как в определении класса. Экземпляр этого класса автоматически создается с именем `Model`, его свойства могут быть установлены в специальных методах, и вы можете получить значения свойств в разметке;
  - методы с именами `OnGet`, `OnPost`, `OnDelete` и т. д., которые работают при выполнении HTTP-запросов, таких как `GET`, `POST` и `DELETE`.

Преобразуем статическую HTML-страницу в Razor Page.

1. В программе Visual Studio Code найдите и откройте файл `index.cshtml`.
2. Добавьте в начало файла оператор `@page`.
3. Добавьте блок оператора `@functions` после оператора `@page`.
4. Определите свойства типа `string` для сохранения названия текущего дня как строкового значения.
5. Для установки переменной `DayName` определите метод, который работает при выполнении HTTP-запроса `GET`:

```
@page
@functions
{
  public string DayName { get; set; }

  public void OnGet()
  {
    Model.DayName = DateTime.Now.ToString("dddd");
  }
}
```

6. Выведите название дня в одном из абзацев:

```
<p>It's @Model.DayName! Our customers include restaurants, hotels, and
cruise lines.</p>
```

7. Запустите веб-приложение с помощью браузера Google Chrome, как вы это делали раньше, перейдя по URL-адресу http://localhost:5000/. Вы увидите, что на странице отображается название текущего дня (рис. 15.8).
8. В Google Chrome введите адрес http://localhost:5000/index.html, что точно соответствует имени статического файла, и обратите внимание, что он, как и раньше, возвращает статическую HTML-страницу.
9. На панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C, чтобы остановить веб-сервер.

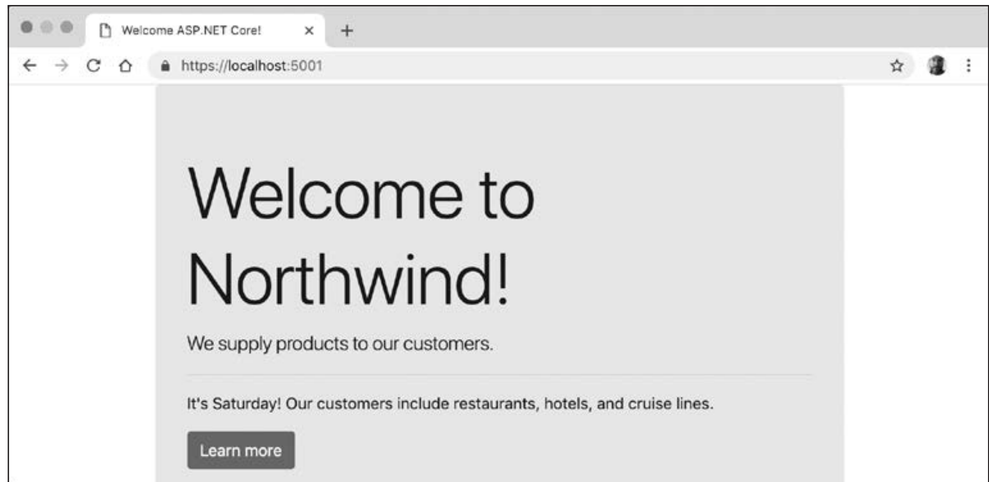


Рис. 15.8. Добро пожаловать а Northwind!

Использование общих макетов в Razor Pages

Большинство сайтов имеют более одной страницы. Если бы каждая из них содержала всю шаблонную разметку, которая в настоящее время находится в файле `index.cshtml`, то управление стало бы проблематичным. Поэтому ASP.NET Core содержит *макеты*.

Чтобы использовать макеты, следует создать Razor-файл, определяющий макет по умолчанию для всех Razor Pages (и всех представлений MVC) и сохранить его в папке Shared, таким образом облегчив его поиск по соглашению. Название этого файла может быть любым, однако хорошей практикой является использование `_Layout.cshtml`. Кроме того, чтобы установить макет по умолчанию для всех Razor Pages (и всех представлений MVC), необходимо создать файл с особым именем. Он должен называться `_ViewStart.cshtml`.

1. В папке Pages создайте файл `_ViewStart.cshtml`.
2. Измените содержимое файла:

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

3. В папке Pages создайте подпапку Shared.
4. В папке Shared создайте файл `_Layout.cshtml`.
5. Отредактируйте содержимое файла `_Layout.cshtml` (оно похоже на содержимое файла `index.cshtml`, поэтому вы можете просто скопировать и вставить его из файла):

```
<!DOCTYPE html>
<html lang="en">

<head>
  <!-- Required meta tags -->
  <meta charset="utf-8" />
  <meta name="viewport" content=
    "width=device-width, initial-scale=1, shrink-to-fit=no" />

  <!-- Bootstrap CSS -->
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/
  bootstrap/4.5.2/css/bootstrap.min.css" integrity="sha384-JcKb8q3iqJ61gNV9
  KGb8thSsNjpSL0n8PARn9HuZOnIxN0hoP+VmmDGMN5t9UJ0Z " crossorigin="anonymous">

  <title>@ViewData["Title"]</title>
</head>

<body>
  <div class="container">
  @RenderBody()
  <hr />
  <footer>
    <p>Copyright &copy; 2020 - @ViewData["Title"]</p>
  </footer>
  </div>

  <!-- JavaScript to enable features like carousel -->
  <!-- jQuery first, then Popper.js, then Bootstrap JS -->
  <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"
  integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+
  OrCXaRkfj" crossorigin="anonymous"></script>

  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/
  umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clH
  TMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>

  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/
  bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEa
  Ff/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>

  @RenderSection("Scripts", required: false)

</body>
</html>
```

Просматривая предыдущую разметку, обратите внимание на следующие моменты.

- Элемент `<title>` устанавливается динамически с помощью серверного кода из словаря `ViewData`. Это простой способ передачи данных между различными элементами сайта на ASP.NET Core. В этом случае данные будут установлены в файле класса Razor Page, а затем выведены в общий макет.

- Метод `@RenderBody()` отмечает точку вставки для запрашиваемой страницы.
- Внизу каждой страницы будут отображаться горизонтальная линейка и нижний колонтитул.
- В нижней части макета находятся несколько сценариев для реализации кое-каких полезных функций Bootstrap, которые мы будем использовать позже, — например, карусель изображений.
- После элементов `<script>` для Bootstrap мы определили раздел под названием `Scripts`, чтобы с помощью технологии Razor Page можно было вставить дополнительные необходимые скрипты.

6. Отредактируйте код файла `index.cshtml`: удалите всю HTML-разметку, за исключением `<div class="jumbotron">` и ее содержимого, и оставьте код C# в добавленном ранее блоке `@functions`.
7. Добавьте в метод `OnGet` оператор для сохранения заголовка страницы в словаре `ViewData` и измените кнопку для перехода на страницу поставщиков (которую мы создадим в следующем разделе), как показано ниже (выделено полужирным шрифтом):

```
@page
@functions
{
  public string DayName { get; set; }

  public void OnGet()
  {
    ViewData["Title"] = "Northwind Website";

    Model.DayName = DateTime.Now.ToString("dddd");
  }
}
<div class="jumbotron">
  <h1 class="display-3">Welcome to Northwind!</h1>
  <p class="lead">We supply products to our customers.</p>
  <hr />
  <p>It's @Model.DayName! Our customers include restaurants, hotels, and
     cruise lines.</p>
  <p>
    <a class="btn btn-primary" href="suppliers">
      Learn more about our suppliers</a>
  </p>
</div>
```

8. Запустите веб-приложение с помощью браузера Google Chrome. Вы увидите, что оно работает аналогично предыдущему, хотя нажатие кнопки для перехода на страницу поставщиков приведет к ошибке `404 Not Found`, поскольку страница еще не создана.

Использование файлов с выделенным кодом в Razor Pages

Иногда лучше отделить HTML-разметку от данных и исполняемого кода, поэтому технология Razor Pages допускает файлы классов *с выделенным кодом*.

Создадим страницу, которая отображает список поставщиков. В этом примере мы сконцентрируемся на изучении файлов с выделенным кодом. В следующей теме мы загрузим список поставщиков из базы данных, но в настоящий момент смоделируем это с помощью жестко запрограммированного массива строковых значений.

1. В папку Pages добавьте два новых файла: `suppliers.cshtml` и `suppliers.cshtml.cs`.
2. В файл `suppliers.cshtml.cs` добавьте операторы:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;

namespace NorthwindWeb.Pages
{
  public class SuppliersModel : PageModel
  {
    public IEnumerable<string> Suppliers { get; set; }

    public void OnGet()
    {
      ViewData["Title"] = "Northwind Web Site - Suppliers";

      Suppliers = new[] {
        "Alpha Co", "Beta Limited", "Gamma Corp"
      };
    }
  }
}
```

Просматривая предыдущую разметку, обратите внимание на следующие моменты.

- Класс `SuppliersModel` наследуется от `PageModel`, поэтому содержит такие элементы, как словарь `ViewData` для совместного использования данных. Вы можете выбрать класс `PageModel` и нажать клавишу F12, чтобы увидеть, что в классе содержится намного больше полезных элементов наподобие объекта `HttpContext` текущего запроса.
- Класс `SuppliersModel` определяет свойство для хранения коллекции строковых значений с именем `Suppliers`.

- Когда для этой Razor-страницы выполняется HTTP-запрос `GET`, свойство `Suppliers` заполняется некоторыми именами поставщиков.

3. Отредактируйте содержимое файла `sources.cshtml`:

```
@page
@model NorthwindWeb.Pages.SuppliersModel
<div class="row">
  <h1 class="display-2">Suppliers</h1>
  <table class="table">
    <thead class="thead-inverse">
      <tr><th>Company Name</th></tr>
    </thead>
    <tbody>
    @foreach(string name in Model.Suppliers)
    {
      <tr><td>@name</td></tr>
    }
    </tbody>
  </table>
</div>
```

Просматривая предыдущую разметку, обратите внимание на следующие моменты.

- Типом модели для этой Razor-страницы выбран класс `SuppliersModel`.
- На странице отображается HTML-таблица со стилями Bootstrap.
- Строки данных в таблице создаются циклически через свойство `Suppliers` класса `Model`.

4. Запустите веб-приложение с помощью браузера Google Chrome, нажмите кнопку, позволяющую узнать больше информации о поставщиках, и обратите внимание на таблицу `Suppliers` (рис. 15.9).

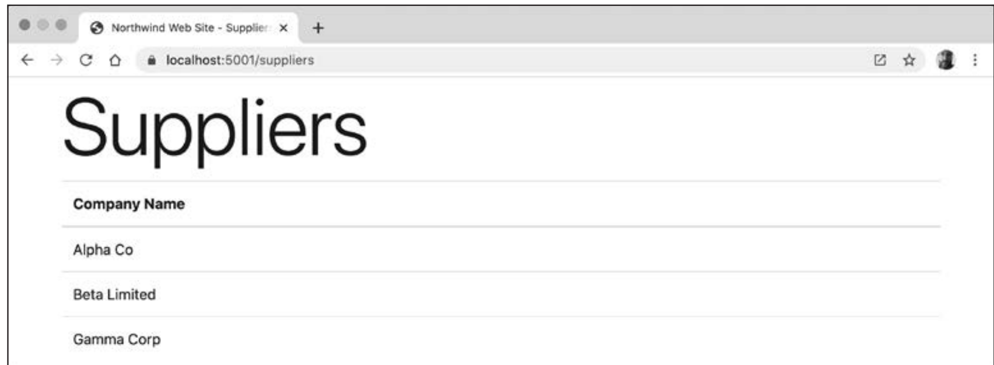


Рис. 15.9. Временный список поставщиков

Использование Entity Framework Core совместно с ASP.NET Core

Использовать Entity Framework Core — естественный способ получить реальные данные на сайте. В главе 14 вы создали две библиотеки классов: для сущностных моделей и для контекста базы данных Northwind.

Настройка Entity Framework Core в виде сервиса

Такие функциональные возможности, как контексты базы данных Entity Framework Core, необходимые для ASP.NET Core, во время запуска должны быть зарегистрированы в виде сервисов.

1. В проекте NorthwindWeb отредактируйте код файла NorthwindWeb.csproj, добавив ссылку на проект NorthwindContextLib, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindContextLib\NorthwindContextLib.csproj" />
  </ItemGroup>
</Project>
```

2. На панели TERMINAL (Терминал) восстановите пакеты и скомпилируйте проект с помощью следующей команды: dotnet build.
3. Откройте файл Startup.cs и импортируйте пространства имен System.IO, Microsoft.EntityFrameworkCore и Packt.Shared:

```
using System.IO;
using Microsoft.EntityFrameworkCore;
using Packt.Shared;
```

4. Добавьте в метод ConfigureServices следующий оператор, чтобы зарегистрировать класс контекста базы данных Northwind для использования SQLite в качестве поставщика БД. Укажите строку подключения к базе, как показано ниже:

```
string databasePath = Path.Combine("..", "Northwind.db");

services.AddDbContext<Northwind>(options =>
  options.UseSqlite($"Data Source={databasePath}"));
```

5. В проекте NorthwindWeb в папке Pages найдите и откройте файл `sources.cshtml.cs` и импортируйте пространства имен `Packt.Shared` и `System.Linq`, как показано ниже:

```
using System.Linq;
using Packt.Shared;
```

6. Чтобы получить контекст базы данных Northwind, в классе `SuppliersModel` добавьте скрытое поле и конструктор:

```
private Northwind db;

public SuppliersModel(Northwind injectedContext)
{
  db = injectedContext;
}
```

7. Чтобы получить имена поставщиков, измените в методе `OnGet` операторы, выбрав названия компаний из свойства `Suppliers` контекста базы данных, как показано ниже (выделено полужирным шрифтом):

```
public void OnGet()
{
  ViewData["Title"] = "Northwind Web Site - Suppliers";

  Suppliers = db.Suppliers.Select(s => s.CompanyName);
}
```

8. На панели TERMINAL (Терминал) введите команду `dotnet run` для запуска сайта. В адресной строке браузера Google Chrome введите следующий URL: http://localhost:5000/. Нажмите кнопку, чтобы перейти на страницу Suppliers. Вы увидите, что таблица поставщиков теперь загружается из базы данных (рис. 15.10).

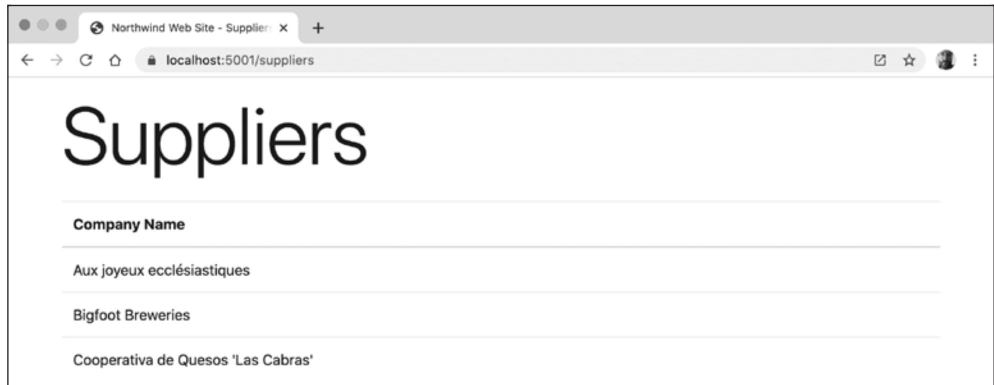


Рис. 15.10. Загрузка таблицы Suppliers из базы данных

Управление данными с помощью страниц Razor

Добавим функции для вставки нового поставщика.

Добавление модели возможности вставки объектов

В первую очередь необходимо изменить модель поставщика, чтобы она отвечала на HTTP-запросы POST, когда пользователь отправляет форму для вставки нового поставщика.

1. В проекте NorthwindWeb в папке Pages найдите и откройте файл sources.cshtml.cs и импортируйте следующее пространство имен:

```
using Microsoft.AspNetCore.Mvc;
```

2. В класс SuppliersModel добавьте свойство для хранения поставщика и метод OnPost, чтобы добавить поставщика, если его модель действительна:

```
[BindProperty]
public Supplier Supplier { get; set; }

public IActionResult OnPost()
{
  if (ModelState.IsValid)
  {
    db.Suppliers.Add(Supplier);
    db.SaveChanges();
    return RedirectToPage("/suppliers");
  }
  return Page();
}
```

Просматривая предыдущий код, обратите внимание на следующие моменты.

- Мы добавили свойство Supplier, которое дополнено атрибутом [BindProperty], чтобы упростить связку HTML-элементов на веб-странице со свойствами в классе Supplier.
- Мы добавили метод, отвечающий на HTTP-запросы POST. Он проверяет все значения свойств на соответствие правилам проверки, а затем добавляет поставщика в существующую таблицу и сохраняет изменения в контексте БД. Это сгенерирует оператор SQL для выполнения вставки в базу данных. Затем запрос перенаправляет на страницу Suppliers для отображения только что добавленного поставщика.

Определение формы для добавления новых поставщиков

Во-вторых, необходимо изменить страницу Razor, чтобы определить форму, которую пользователь может заполнить и отправить для вставки нового поставщика.

1. Откройте файл `suppliers.cshtml` и после объявления `@model` добавьте «тег-хелперы», чтобы мы их могли использовать на этой странице (например, `asp-for`), как показано ниже:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

2. В конце файла добавьте форму для вставки нового поставщика и с помощью «тег-хелпера» `asp-for` подключите к полю ввода свойство `CompanyName` класса `Supplier`:

```
<div class="row">
  <p>Enter a name for a new supplier:</p>
  <form method="POST">
    <div><input asp-for="Supplier.CompanyName" /></div>
    <input type="submit" />
  </form>
</div>
```

Просматривая предыдущий код, обратите внимание на следующие моменты.

- Элемент `<form>` с методом `POST` — обычный HTML-элемент, поэтому внутри него элемент `<input type="submit" />` отправит HTTP-запрос `POST` обратно на текущую страницу вместе со значениями любых других элементов внутри данной формы.
- Элемент `<input>` с «тег-хелпером» `asp-for` обеспечивает привязку данных к модели на странице Razor.

3. Запустите веб-приложение, выберите пункт Learn more about our suppliers, прокрутите вниз таблицу поставщиков. Затем для добавления нового поставщика введите `Bob's Burgers` и нажмите кнопку Submit (Отправить) как показано на рис. 15.11.

Рис. 15.11. Ввод нового поставщика

4. Обратите внимание: вы перенаправлены обратно в список *Suppliers* с добавленным новым поставщиком.
5. Закройте браузер.

Выполните следующее упражнение: добавьте в таблицу Suppliers дополнительные поля, такие как Country (Страна) и Phone (Телефон).

Применение библиотек классов Razor

Все связанное со страницей Razor можно скомпилировать в библиотеку классов, чтобы облегчить повторное использование. В .NET Core 3.0 и более поздних версиях туда же можно включать статические файлы. Сайт может задействовать представление страницы Razor, как определено в библиотеке классов, или переопределить его.

Создание библиотеки классов Razor

Создадим новую библиотеку классов Razor.

1. В папке PracticalApps создайте подпапку NorthwindEmployees.
2. В программе Visual Studio Code добавьте папку NorthwindEmployees в рабочую область `PracticalApps`.
3. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку NorthwindEmployees.
4. На панели TERMINAL (Терминал) введите следующую команду для создания проекта библиотеки классов Razor:

```
dotnet new razorclasslib -s
```

Параметр -s является сокращением от --support-pages-and-views, который позволяет библиотеке классов использовать представления файлов Razor Pages и .cshtml.

Отключение компактных папок

Прежде чем мы реализуем нашу библиотеку классов Razor, я хочу объяснить недавнюю функцию Visual Studio Code, которая смутила некоторых читателей предыдущего издания, поскольку эта функция была добавлена после публикации.

Функция компактных папок означает, что вложенные папки, такие как /Areas/MyFeature/Pages/, отображаются в компактной форме, если промежуточные папки в иерархии не содержат файлов (рис. 15.12).

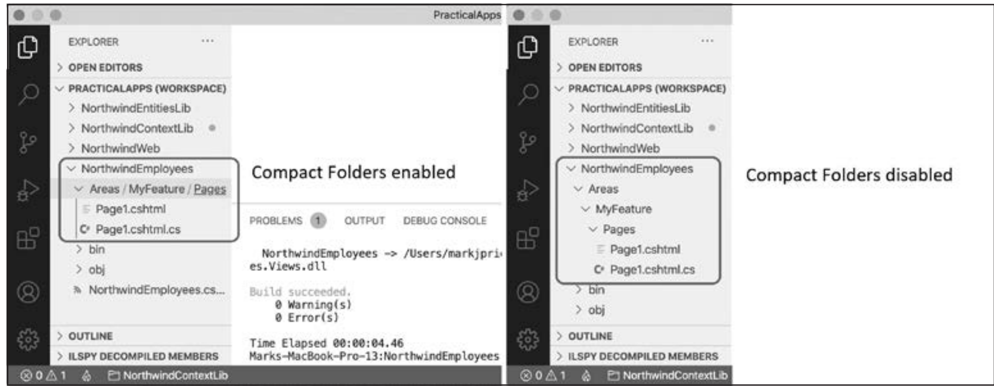


Рис. 15.12. Включено или отключено сжатие папок

Если вы хотите отключить функцию компактных папок Visual Studio Code, выполните следующие действия.

1. В macOS перейдите в Code ▸ Preferences ▸ Settings (Код ▸ Параметры ▸ Настройки) или нажмите сочетание клавиш Cmd+,. В Windows выберите пункт меню File ▸ Preferences ▸ Settings (Файл ▸ Параметры ▸ Настройки) или нажмите сочетание клавиш Ctrl+,.
2. В поле настроек Search введите `compact`.
3. Снимите флажок Explorer: Compact Folders (рис. 15.13).
4. Закройте вкладку Settings (Настройки).

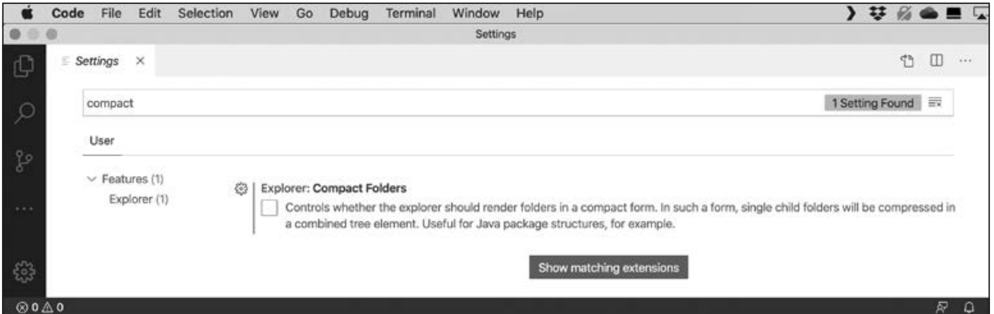


Рис. 15.13. Отключение компактных папок

Информацию, касающуюся функции компактных папок, представленную в Visual Studio Code 1.41 в ноябре 2019 года, вы можете прочитать на сайте https://github.com/microsoft/vscode-docs/blob/vnext/release-notes/v1_41.md#compact-folders-in-explorer.

Реализация функции сотрудников с помощью EF Core

Теперь, чтобы сотрудники отображались в библиотеке классов Razor, мы можем добавить ссылку на наши модели сущностей.

1. Отредактируйте код файла `NorthwindEmployees.csproj` и обратите внимание, что SDK — это Microsoft.NET.Sdk.Razor, и добавьте ссылку на проект NorthwindContextLib.

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>
  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindContextLib\NorthwindContextLib.csproj" />
  </ItemGroup>
</Project>
```

2. На панели TERMINAL (Терминал) введите следующую команду для восстановления пакетов и компиляции проекта: `dotnet build`.
3. На панели EXPLORER (Проводник) разверните папку Areas и переименуйте папку MyFeature в PacktFeatures. Выберите Rename, введите новое имя PacktFeatures и нажмите Enter.
4. На панели EXPLORER (Проводник) разверните папку PacktFeatures и добавьте в подпапку Pages новый файл `_ViewStart.cshtml`.
5. Отредактируйте код файла:

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

6. В подпапке Pages найдите файл `Page1.cshtml` и переименуйте в `employee.cshtml`, а файл `Page1.cshtml.cs` — в `employee.cshtml.cs`.

7. Отредактируйте файл `employee.cshtml.cs`, чтобы определить модель страницы с массивом экземпляров объектов `Employee`, загруженных из базы данных `Northwind`:

```
using Microsoft.AspNetCore.Mvc.RazorPages; // PageModel
using Packt.Shared;                        // Employee
using System.Linq;                         // ToArray()
using System.Collections.Generic;          // IEnumerable<T>

namespace PacktFeatures.Pages
{
  public class EmployeesPageModel : PageModel
  {
    private Northwind db;

    public EmployeesPageModel(Northwind injectedContext)
    {
      db = injectedContext;
    }

    public IEnumerable<Employee> Employees { get; set; }

    public void OnGet()
    {
      Employees = db.Employees.ToArray();
    }
  }
}
```

8. Отредактируйте файл `employee.cshtml`:

```
@page
@using Packt.Shared
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model PacktFeatures.Pages.EmployeesPageModel
<div class="row">
  <h1 class="display-2">Employees</h2>
</div>
<div class="row">
  @foreach(Employee in Model.Employees)
  {
  <div class="col-sm-3">
    <partial name="_Employee" model="employee" />
  </div>
  }
</div>
```

Просматривая предыдущий код, обратите внимание на следующие моменты.

- Мы импортируем пространство имен `Packt.Shared`, чтобы в нем можно было использовать классы, такие как класс `Employee`.

- Мы добавили поддержку для «тег-хелпера», чтобы можно было использовать элемент `<partial>`.
- Мы объявили тип модели для нашей страницы Razor, чтобы задействовать только что определенный класс.
- Мы проходимся по элементам коллекции `Employees`, выводя каждый элемент с помощью частичного представления. Частичные представления похожи на небольшие фрагменты страницы Razor, и их можно создать за несколько следующих шагов.

В ASP.NET Core 2.1 был введен «тег-хелпер» <partial>. Более подробную информацию можно получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/tag-helpers/built-in/partial-tag-helper.

Реализация частичного представления для отображения одного сотрудника

Теперь мы определим частичное представление для визуализации одного сотрудника.

1. В папке Pages создайте подпапку Shared.
2. В папке Shared создайте файл `_Employee.cshtml`.
3. Отредактируйте файл `_Employee.cshtml`:

```
@model Packt.Shared.Employee
<div class="card border-dark mb-3" style="max-width: 18rem;">
  <div class="card-header">@Model.FirstName
    @Model.LastName</div>
  <div class="card-body text-dark">
    <h5 class="card-title">@Model.Country</h5>
    <p class="card-text">@Model.Notes</p>
  </div>
</div>
```

Просматривая предыдущий код, обратите внимание на следующие моменты.

- По соглашению имена частичных представлений начинаются с подчеркивания.
- Если вы поместите частичное представление в папку Shared, то его можно будет найти автоматически.
- Объект `Employee` — тип модели для этого частичного представления.
- Чтобы вывести информацию о каждом сотруднике, в нашем примере используются стили карточек Bootstrap.

Использование библиотек классов Razor

Теперь в проекте сайта вы будете ссылаться на библиотеку классов Razor и использовать ее.

1. Чтобы добавить ссылку на проект `NorthwindEmployees`, отредактируйте файл `NorthwindWeb.csproj`, как показано ниже (выделено полужирным шрифтом):

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindContextLib\NorthwindContextLib.csproj" />
    <ProjectReference Include=
      "..\NorthwindEmployees\NorthwindEmployees.csproj" />
  </ItemGroup>
</Project>
```

2. Чтобы добавить ссылку на страницу сотрудников после ссылки на страницу поставщиков, отредактируйте файл `Pages\index.cshtml`:

```
<p>
  <a class="btn btn-primary" href="packtfeatures/employees">
    Contact our employees
  </a>
</p>
```

3. Запустите веб-приложение с помощью браузера Google Chrome и нажмите кнопку, чтобы просмотреть карточки сотрудников (рис. 15.14).

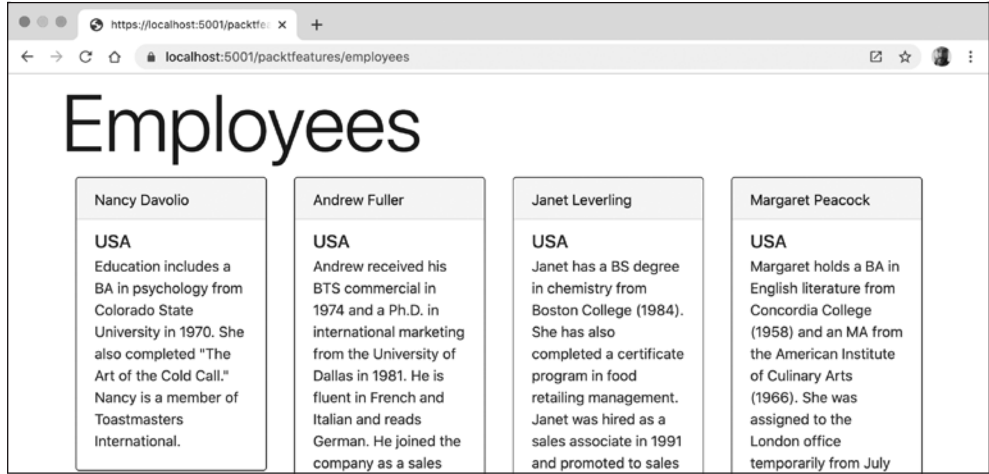


Рис. 15.14. Список сотрудников

Настройка служб и конвейера HTTP-запросов

Теперь, когда мы создали сайт, мы можем вернуться к настройкам и более подробно рассмотреть, как работают службы и конвейер HTTP-запросов.

Проанализируйте файл класса `Startup.cs`:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.EntityFrameworkCore;
using Packt.Shared;
using System;
using System.IO;
using System.Threading.Tasks;

namespace NorthwindWeb
{
  public class Startup
  {
    // Этот метод вызывается средой выполнения.
    // Используйте этот метод для добавления служб в контейнер.
    // Дополнительные сведения о настройках приложения можно изучить
    // на сайте https://go.microsoft.com/fwlink/?LinkID=398940.
    public void ConfigureServices(IServiceCollection services)
    {
      services.AddRazorPages();

      string databasePath = Path.Combine("..", "Northwind.db");

      services.AddDbContext<Northwind>(options =>
        options.UseSqlite($"Data Source={databasePath}"));
    }

    // Этот метод вызывается средой выполнения.
    // Используйте этот метод для настройки конвейера HTTP-запросов.
    public void Configure(
      IApplicationBuilder app, IWebHostEnvironment env)
    {
      if (env.IsDevelopment())
      {
        app.UseDeveloperExceptionPage();
      }
      else
      {
        app.UseHsts();
      }

      app.UseRouting();
```

```
      app.UseHttpsRedirection();

      app.UseDefaultFiles(); // index.html, default.html и т.д.
      app.UseStaticFiles();

      app.UseEndpoints(endpoints =>
      {
        endpoints.MapRazorPages();

        endpoints.MapGet("/hello", async context =>
        {
          await context.Response.WriteAsync("Hello World!");
        });
      });
    }
  }
}
```

Класс Startup содержит два метода, которые автоматически вызываются хостом для настройки сайта. Метод `ConfigureServices` регистрирует службы, которые затем могут быть извлечены при необходимости предоставляемой ими функциональности с помощью внедрения зависимостей. Наш код регистрирует две службы: Razor Pages и контекст базы данных EF Core.

Регистрация служб

В следующей таблице продемонстрированы общие методы, регистрирующие службы зависимостей, включая службы, которые объединяют вызовы других методов, регистрирующие службы.

Метод	**Регистрирующие службы**
AddMvcCore	Минимальный набор сервисов, необходимых для маршрутизации запросов и вызова контроллеров. Большинству сайтов потребуются дополнительные настройки
AddAuthorization	Аутентификация и авторизация
AddDataAnnotations	Служба аннотаций данных MVC
AddCacheTagHelper	Вспомогательная служба тегов кеширования MVC
AddApiExplorer	Служба Razor Pages, включая механизм просмотра Razor. Обычно используется в простых проектах сайтов. Вызывает следующие дополнительные методы: • AddMvcCore; • AddAuthorization; • AddDataAnnotations; • AddCacheTagHelper
AddApiExplorer	Служба обозревателя веб-API

Метод	Регистрирующие службы
AddCors	Поддержка совместного использования ресурсов между источниками (CORS) для повышения безопасности
AddFormatterMappings	Сопоставления между форматом URL-адреса и соответствующим типом мультимедиа
AddControllers	Службы контроллера (но не службы для представлений или страниц). Обычно используется в проектах веб-API ASP.NET Core. Вызывает следующие дополнительные методы: • AddMvcCore; • AddAuthorization; • AddDataAnnotations; • AddCacheTagHelper; • AddApiExplorer; • AddCors; • AddFormatterMappings
AddViews	Поддержка представлений .cshtml, включая соглашения по умолчанию
AddRazorViewEngine	Поддержка механизма просмотра Razor, включая обработку символа @
AddControllersWithViews	Контроллер, представления и службы страниц. Обычно используется в проектах сайтов ASP.NET Core MVC. Вызывает следующие дополнительные методы: • AddMvcCore; • AddAuthorization; • AddDataAnnotations; • AddCacheTagHelper; • AddApiExplorer; • AddCors; • AddFormatterMappings; • AddViews; • AddRazorViewEngine
AddMvc	Аналогично описанному выше, но вы должны использовать его только для обратной совместимости
AddDbContext<T>	Ваш тип DbContext и его необязательный DbContextOptions <TContext>

Более подробно о регистрации контекста базы данных для использования в качестве службы зависимостей вы можете прочитать по адресу https://docs.microsoft.com/ru-ru/ef/core/miscellaneous/configuring-dbcontext#using-dbcontext#using-dbcontext-with-dependency-injection.

В следующих нескольких главах вы увидите больше примеров использования этих методов расширения для регистрации сервисов при работе с сервисами MVC и Web API.

Конфигурация конвейера HTTP-запросов

Метод Configure конфигурирует конвейер HTTP-запросов, состоящий из связанной последовательности делегатов, которые могут выполнять обработку, а затем принимать решение либо вернуть ответ самим, либо передать обработку следующему делегату в конвейере. Возвращающимися ответами также можно манипулировать.

Помните, что делегаты определяют сигнатуру метода, к которой может подключиться реализация делегата. Делегат для конвейера HTTP-запросов выглядит следующим образом:

```
public delegate Task RequestDelegate(HttpContext context);
```

Вы можете видеть, что входным параметром является `HttpContext`. Это обеспечивает доступ ко всему, что вам может понадобиться для обработки входящего HTTP-запроса, включая URL, параметры строки запроса, файлы cookie, пользовательский агент и т. д.

Более подробно о HttpContext вы можете прочитать по адресу https://docs.microsoft.com/ru-ru/dotnet/api/system.web.httpcontext.

Этих делегатов часто называют промежуточным программным обеспечением, так как они находятся между клиентом браузера и сайтом или службой.

Делегаты промежуточного программного обеспечения конфигурируются с использованием одного из следующих методов или специального метода, который вызывает их сам.

- `Run` — добавляет делегат промежуточного программного обеспечения, который завершает конвейер, немедленно возвращая ответ, вместо вызова следующего делегата промежуточного программного обеспечения.
- `Map` — добавляет делегат промежуточного программного обеспечения, который создает ветвь в конвейере при существующем запросе, основанном на пути URL, например `/hello`.
- `Use` — добавляет делегат промежуточного программного обеспечения, который образует часть конвейера, чтобы определить необходимость передачи запроса следующему делегату в конвейере и изменить запрос и ответ до и после следующего делегата.

Простые примеры применения этих методов опубликованы на сайте https://www.vaughanreid.com/2020/05/using-inline-middleware-in-asp-net-core/.

Существует множество методов расширения, которые упрощают построение конвейера, например `UseMiddleware<T>`, где `T` — это класс, содержащий (1) конструктор с параметром `RequestDelegate`, который передается следующему компоненту конвейера, и (2) метод `Invoke` с параметром `HttpContext`, возвращающий `Task`.

Ключевые методы расширения промежуточного программного обеспечения, используемые в нашем коде, включают следующее.

- `UseDeveloperExceptionPage`: захватывает синхронные и асинхронные экземпляры `System.Exception` из конвейера и генерирует ответы об ошибках HTML.
- `UseHsts`: добавляет промежуточное ПО для использования механизма HSTS, добавляющего заголовок `Strict-Transport-Security`.
- `UseRouting`: добавляет промежуточное ПО, которое определяет точку в конвейере, где принимаются решения о маршрутизации, и которое должно сочетаться с вызовом метода `UseEndpoints`, где затем выполняется обработка. Это означает, что для нашего кода любые URL-пути, соответствующие `/` или `/index` или `/suppliers`, будут сопоставлены с Razor Pages, а соответствующие `/hello` будут сопоставлены с анонимным делегатом. Любые другие URL-пути будут переданы следующему делегату для сопоставления, например, статических файлов. Вот почему, хотя кажется, что соответствие для Razor Pages и `/hello` происходит после статических файлов в конвейере, на самом деле они имеют приоритет, так как вызов `UseRouting` происходит до вызова метода `UseStaticFiles`.
- `UseHttpsRedirection`: добавляет промежуточное ПО для перенаправления HTTP-запросов на HTTPS, поэтому в нашем коде запрос `http://localhost:5000` будет изменен на `https://localhost:5001`.
- `UseDefaultFiles`: добавляет промежуточное ПО, позволяющее отображать файлы по умолчанию текущего пути, поэтому в нашем коде он будет идентифицировать файлы, такие как `index.html`.
- `UseStaticFiles`: добавляет промежуточное ПО, которое для возврата в ответе HTTP ищет в wwwroot статические файлы.
- `UseEndpoints`: добавляет промежуточное ПО для генерации ответов от запросов, принятых ранее в конвейере. Добавляются две конечные точки, как показано ниже.
  - `MapRazorPages`: добавляет промежуточное ПО, которое будет сопоставлять URL-пути, такие как `/suppliers`, с файлом `suppliers.cshtml` страницы Razor, расположенным в папке `/Pages`, и возвращать результаты в виде HTTP-ответа.

- `MapGet`: добавляет промежуточное ПО, которое будет сопоставлять URL-пути, такие как `/hello`, со встроенным делегатом, записывающим простой текст непосредственно в HTTP-ответ.

Конвейер HTTP-запроса и ответа может быть визуализирован как последовательность делегатов запроса, вызываемых один за другим (рис. 15.15). Данная диаграмма исключает некоторые делегаты промежуточного программного обеспечения, такие как `UseHsts`.

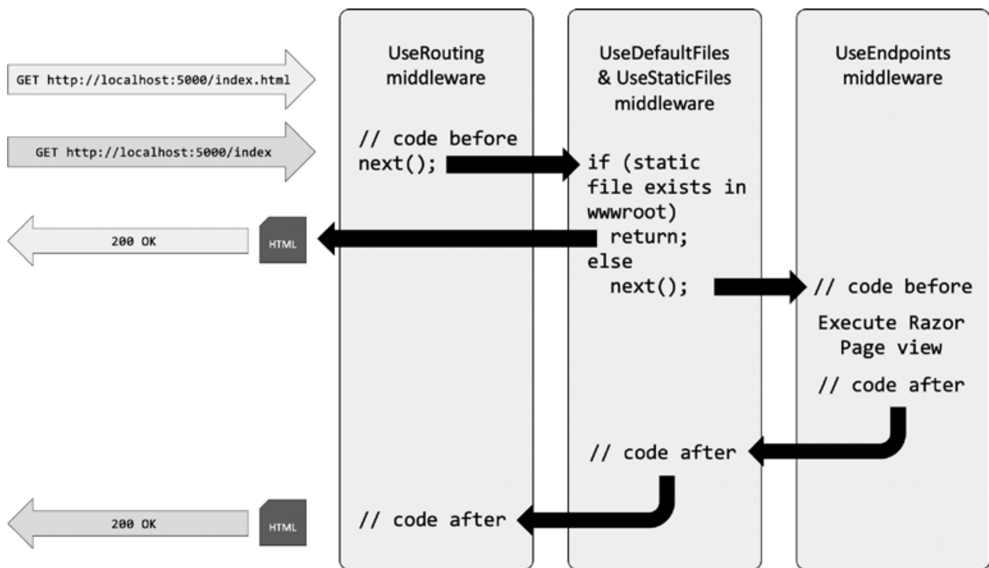


Рис. 15.15. Конвейер HTTP-запросов и ответов

О том, как автоматически визуализировать свои конечные точки, можно узнать на сайте https://andrewlock.net/visualizing-asp-net-core-endpoints-using-graphvizonline-and-the-dot-language/.

Как уже ранее упоминалось, методы `UseRouting` и `UseEndpoints` необходимо использовать вместе. Хотя код для определения сопоставленных путей, например /hello, написан в `UseEndpoints`, решение о том, совпадает ли URL-путь входящего HTTP-запроса и, следовательно, какую конечную точку выполнять, принимается в конвейере в точке `UseRouting`.

Делегат может быть определен в качестве встроенного анонимного метода. Мы зарегистрируем тот, который подключается к конвейеру после того, как будут приняты решения о маршрутизации для конечных точек. Он выведет, какая конечная точка была выбрана, а также обработает один конкретный маршрут: `/bonjour`.

Если маршрут совпадает, в качестве ответа будет обычный текст, без дальнейших вызовов конвейера.

1. Откройте файл класса `Startup.cs`, импортируйте пространство имен `Microsoft.AspNetCore.Routing` и статически импортируйте `Console`.
2. Для использования анонимного метода в качестве делегата промежуточного программного обеспечения перед вызовом метода `UseHttpsRedirection` добавьте операторы:

```
app.Use(async (HttpContext context, Func<Task> next) =>
{
  var rep = context.GetEndpoint() as RouteEndpoint;
  if (rep != null)
  {
    WriteLine($"Endpoint name: {rep.DisplayName}");
    WriteLine($"Endpoint route pattern: {rep.RoutePattern.RawText}");
  }

  if (context.Request.Path == "/bonjour")
  {
// в случае совпадения URL-пути это становится завершающим делегатом,
// который возвращается, поэтому вызов следующего делегата не происходит
    await context.Response.WriteAsync("Bonjour Monde!");
    return;
  }
  // мы могли изменить запрос перед вызовом следующего делегата
  await next();
  // мы могли изменить ответ после вызова следующего делегата
});
```

3. Запустите сайт.
4. В Google Chrome перейдите по адресу https://localhost:5001/ и обратите внимание на панели `TERMINAL` (Терминал), что было совпадение на маршруте к конечной точке /, оно было обработано как `/index`, а страница Razor `Index.cshtml` была выполнена для возврата ответа:

```
Endpoint name: /index
Endpoint route pattern:
```

5. Перейдите по адресу https://localhost:5001/supplier и обратите внимание на панели TERMINAL (Терминал), что было совпадение на маршруте конечной точки `/supplier` и для возврата ответа была выполнена страница Razor `supplier.cshtml`, как показано ниже в выводе:

```
Endpoint name: /suppliers
Endpoint route pattern: suppliers
```

6. Перейдите по адресу https://localhost:5001/index и обратите внимание на панели TERMINAL (Терминал), что было совпадение на маршруте конечной точки /index

и для возврата ответа была выполнена страница Razor `Index.cshtml`, как показано ниже в выводе:

```
Endpoint name: /index
Endpoint route pattern: index
```

7. Перейдите по адресу https://localhost:5001/index.html и обратите внимание на панели TERMINAL (Терминал), что на маршруте конечной точки не было совпадения, но было совпадение для статического файла, поэтому он был возвращен в качестве ответа.
8. Перейдите по адресу https://localhost:5001/hello и обратите внимание на панели TERMINAL (Терминал), что было совпадение на маршруте конечной точки `/hello`, и для возврата простого текстового ответа был выполнен анонимный метод, как показано ниже в выводе:

```
Endpoint name: /hello HTTP: GET
Endpoint route pattern: /hello
```

9. Закройте браузер Google Chrome и остановите работу сайта.

О настройках во время работы конвейера HTTP с промежуточным программным обеспечением вы можете прочитать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/middleware.

Простой проект сайта ASP.NET Core

Создадим простой сайт ASP.NET Core, содержащий код всего из 15 строк. Используем программную функцию верхнего уровня C# 9 и реализацию конвейера запросов, возвращающую один и тот же HTTP-ответ.

1. В существующей папке `PracticalApps` создайте подпапку `SimpleWeb` и добавьте ее в рабочее пространство `PracticalApps`.
2. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку `SimpleWeb`.
3. Для создания консольного приложения на панели TERMINAL (Терминал) введите следующую команду: `dotnet new console`.
4. В качестве активного проекта выберите SimpleWeb.
5. Отредактируйте файл `SimpleWeb.csproj` и пакет SDK на `Microsoft.NET.Sdk.Web`, как показано ниже:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

6. Отредактируйте файл `Program.cs`:

```
using Microsoft.AspNetCore.Hosting; // IWebHostBuilder.Configure
using Microsoft.AspNetCore.Builder; // IApplicationBuilder.Run
using Microsoft.AspNetCore.Http;   // HttpResponse.WriteAsync
using Microsoft.Extensions.Hosting; // Host

Host.CreateDefaultBuilder(args)
  .ConfigureWebHostDefaults(webBuilder =>
  {
    webBuilder.Configure(app =>
    {
      app.Run(context =>
        context.Response.WriteAsync("Hello World Wide Web!"));
    });
  })
  .Build().Run();
```

7. Запустите сайт.
8. В Google Chrome перейдите по адресу http://localhost:5000/ и обратите внимание, что, независимо от используемого пути URL, в качестве ответа всегда будет один и тот же простой текст.
9. Закройте браузер Google Chrome и остановите веб-сервер.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 15.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Перечислите шесть методов, которые могут быть указаны в HTTP-запросе.
2. Перечислите шесть кодов состояния и их описания, которые могут быть возвращены в HTTP-ответе.
3. Для чего в ASP.NET Core используется класс `Startup`?
4. Что такое HSTS и для чего он предназначен?
5. Как включить статические HTML-страницы для сайта?
6. Как вставить код C# в HTML, чтобы создать динамическую страницу?
7. Как определить общие макеты для Razor Pages?
8. Как отделить разметку от кода на странице Razor?

9. Как настроить контекст данных Entity Framework Core для использования с сайтом ASP.NET Core?
10. Как повторно использовать Razor Pages в ASP.NET Core 2.2 или более поздней версии?

Упражнение 15.2. Веб-приложение, управляемое данными

Добавьте на сайт NorthwindWeb страницу Razor, позволяющую пользователю просматривать список клиентов, сгруппированных по странам. Когда тот щелкает на записи клиента, отображается страница, на которой указаны полные контактные данные выбранного клиента и список его заказов.

Упражнение 15.3. Создание веб-страниц для консольных приложений

Реализуйте повторно некоторые консольные приложения, рассмотренные в предыдущих главах как Razor Pages, например из главы 4. Предоставьте веб-интерфейс пользователя для вывода таблиц умножения, расчета налогов и генерации факториалов и последовательности Фибоначчи.

Упражнение 15.4. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- введение в ASP.NET Core: https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/;
- статические файлы в ASP.NET Core: https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/static-files;
- введение в Razor Pages в ASP.NET Core: https://docs.microsoft.com/ru-ru/aspnet/core/razor-pages/;
- синтаксис Razor для ASP.NET Core: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/razor;
- макет в ASP.NET Core: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/layout;
- «тег-хелперы» в ASP.NET Core: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/tag-helpers/intro;
- Razor Pages в ASP.NET Core с EF Core: https://docs.microsoft.com/ru-ru/aspnet/core/data/ef-rp/intro;
- дорожная карта и график поставок ASP.NET Core: https://www.stevejgordon.co.uk/how-is-the-asp-net-core-middleware-pipeline-built.

Резюме

Вы узнали об основах веб-разработки с помощью HTTP, научились создавать простой сайт, возвращающий статические файлы, и использовали Razor Pages в ASP.NET Core совместно с Entity Framework Core для создания веб-страниц, которые динамически генерировались с помощью информации из базы данных.

Мы рассмотрели конвейер HTTP-запросов и ответов, работу вспомогательных методов расширения и варианты добавления собственного промежуточного ПО, влияющего на обработку.

Далее вы научитесь создавать более сложные сайты, используя ASP.NET Core MVC, который разделяет технические аспекты создания сайта на модели, представления и контроллеры, чтобы упростить управление ими.

16 Разработка сайтов с использованием паттерна MVC

Данная глава посвящена разработке сайтов с современной HTTP-архитектурой на стороне сервера с помощью Microsoft ASP.NET Core MVC. Вы узнаете о конфигурации запуска, аутентификации, авторизации, о маршрутах, моделях, представлениях и контроллерах, составляющих ASP.NET Core MVC.

В этой главе:

- настройка сайта ASP.NET Core MVC;
- изучение сайта ASP.NET Core MVC;
- добавление собственного функционала на сайт ASP.NET Core MVC;
- использование других шаблонов проектов.

Настройка сайта ASP.NET Core MVC

Технология Razor Pages в ASP.NET Core отлично подходит для разработки простых сайтов. Более сложным было бы лучше иметь более формальную структуру для управления этой сложностью.

Здесь весьма кстати придется паттерн проектирования *Model-View-Controller* (MVC, «Модель — Представление — Контроллер»). Он использует технологии, аналогичные Razor Pages, но обеспечивает более четкое разделение между техническими компонентами.

- *Модели* — классы, представляющие информацию о сущностях, и модели представления, применяемые при разработке сайта.
- *Представления* — файлы Razor, то есть файлы `.cshtml`, которые отображают данные из моделей представления на HTML-страницах. Фреймворк Blazor использует расширение файла `.razor`, но это не то же, что файлы Razor!
- *Контроллеры* — классы, которые исполняют код при поступлении HTTP-запроса на веб-сервер. Код обычно создает модель представления, которая

может содержать сущностные модели, и передает ее представлению, чтобы генерировать HTTP-ответ для отправки обратно в браузер или другой клиент.

Лучший способ понять технологию MVC — увидеть рабочий пример.

Создание и изучение сайтов ASP.NET Core MVC

Для создания приложения ASP.NET Core MVC с базой данных для аутентификации и авторизации пользователей в нашем примере будет применяться шаблон проекта `mvc`.

1. Создайте в папке PracticalApps подпапку NorthwindMvc.
2. Откройте в программе Visual Studio Code рабочую область `PracticalApps`, а затем добавьте в нее папку NorthwindMvc.
3. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку NorthwindMvc.
4. На панели TERMINAL (Терминал) создайте новый проект сайта MVC с аутентификацией, сохраненной в базе данных SQLite, как показано в следующей команде:

   ```
   dotnet new mvc --auth Individual
   ```

   Чтобы увидеть другие параметры для этого шаблона, вы можете ввести следующую команду:

   ```
   dotnet new mvc --help
   ```

5. Чтобы запустить сайт, на панели TERMINAL (Терминал) введите команду `dotnet run`.
6. Запустите браузер Google Chrome и откройте инструменты разработчика.
7. В адресной строке браузера введите URL http://localhost:5000/. Обратите внимание на следующие моменты.
   - HTTP-запросы автоматически перенаправляются на HTTPS.
   - Панель навигации располагается в верхней части окна со следующими ссылками: Home (Домой), Privacy (Конфиденциальность), Register (Регистрация) и Login (Вход в систему) (рис. 16.1). Если ширина области просмотра составляет 575 пикселов или меньше, навигация сворачивается в меню «гамбургер».
   - Временное сообщение о политике конфиденциальности и использовании файлов cookie, которое можно скрыть, нажав кнопку Accept (Принять).
   - Название сайта, которое отображается в верхнем и нижнем колонтитулах.

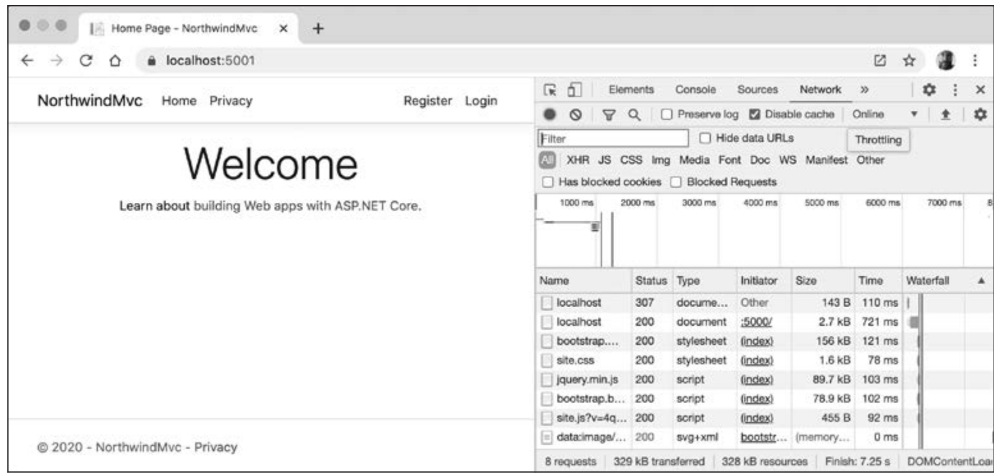


Рис. 16.1. Домашняя страница сайта Northwind MVC

8. Нажмите ссылку Register (Регистрация), введите адрес электронной почты и пароль. Затем нажмите кнопку Register (Регистрация). Для изучения подобных сценариев я использую пароль `Pa$$w0rd`.

   По умолчанию пароли должны содержать хотя бы один не алфавитно-цифровой символ, одну цифру (от 0 до 9) и иметь хотя бы одну прописную букву (A — Z).

   Шаблон проекта MVC соответствует рекомендациям по *двойному согласию* (double-opt-in, DOI). Это значит, что после ввода адреса электронной почты и пароля для регистрации на указанный адрес отправляется письмо. Чтобы подтвердить регистрацию, пользователь должен нажать ссылку в нем.

   Мы еще не настроили поставщик электронной почты для отправки этого письма, поэтому должны симулировать данный шаг.

9. Нажмите ссылку с текстом Click here to confirm your account (Нажмите здесь, чтобы подтвердить свою учетную запись) и обратите внимание, что вы перенаправлены на веб-страницу Confirm email (Подтверждение электронной почты), которую вы можете поменять.

10. В меню навигации в верхней части окна нажмите ссылку Login (Вход в систему), введите свой адрес электронной почты и пароль. (Обратите внимание, что дополнительный флаг служит для сохранения пароля. Здесь же содержатся ссылки на случай, если пользователь забыл пароль или хочет зарегистрироваться в качестве нового пользователя.) Затем нажмите кнопку Log in (Войти в систему).

11. Чтобы перейти на страницу управления учетной записью, нажмите свой адрес электронной почты в меню навигации вверху. Далее вы можете указать номер

телефона, изменить адрес электронной почты, пароль, включить двухуровневую аутентификацию (если добавите приложение для аутентификации). Затем скачайте и удалите свои личные данные (рис. 16.2).

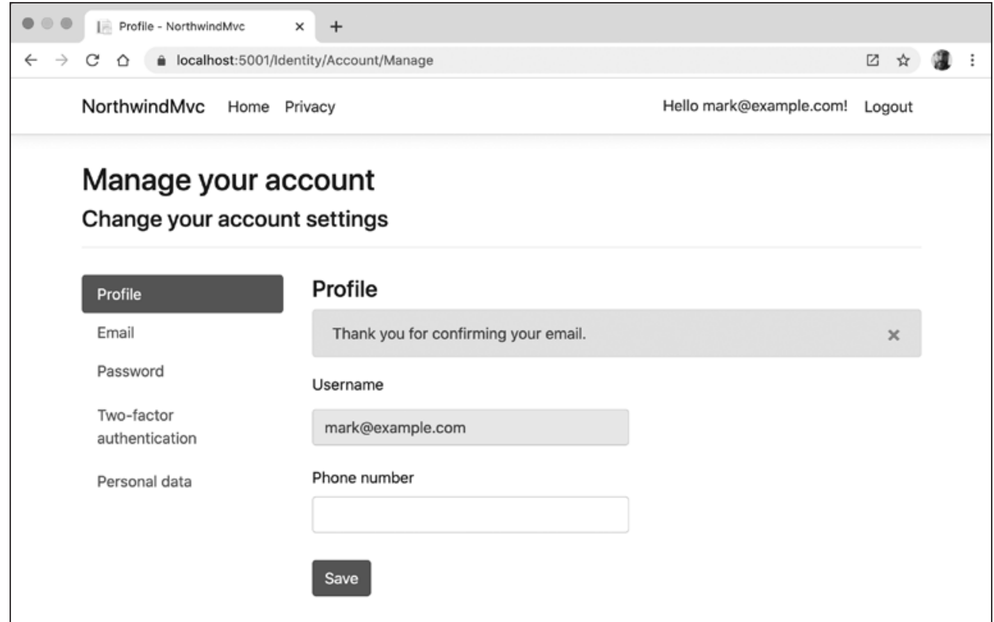


Рис. 16.2. Настройки профиля пользователя

Некоторые из этих встроенных функций базового шаблона проекта MVC облегчают соответствие вашего сайта современным требованиям конфиденциальности, таким как «Общее положение о защите данных» Европейского союза (General Data Protection Regulation, GDPR), которое вступило в силу в мае 2018 года. Более подробную информацию можно получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/security/gdpr.

12. Закройте браузер.
13. На панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C, чтобы остановить консольное приложение и завершить работу веб-сервера Kestrel, на котором размещается сайт ASP.NET Core.

Более подробную информацию о поддержке ASP.NET Core приложений для аутентификации можно получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/security/authentication/identity-enable-qrcodes.

Обзор сайта ASP.NET Core MVC

В программе Visual Studio Code откройте панель EXPLORER (Проводник) (рис. 16.3).

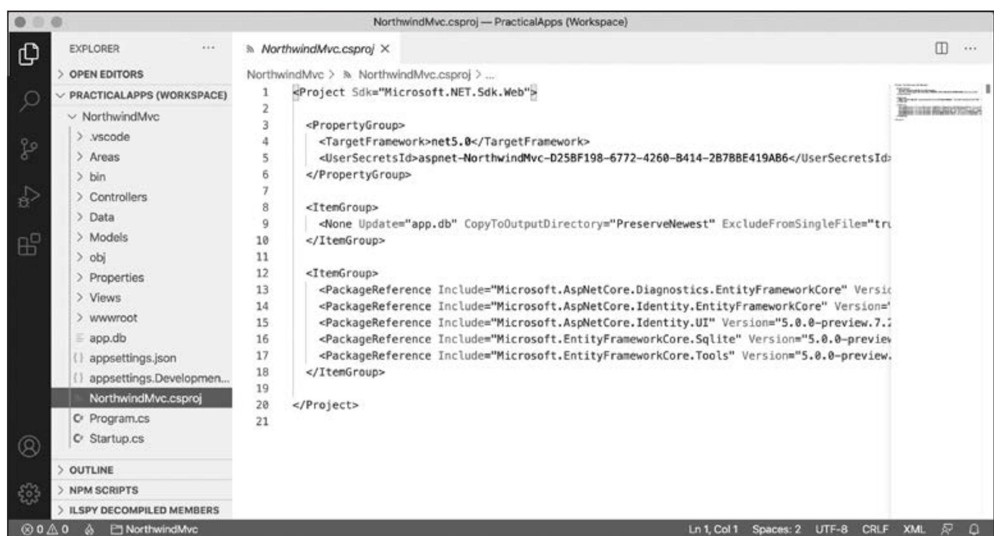


Рис. 16.3. Панель EXPLORER (Проводник), отображающая начальную структуру проекта MVC

Позже мы более подробно рассмотрим некоторые из представленных разделов, однако в настоящий момент необходимо отметить следующее.

- Папка Areas содержит файлы, необходимые для таких функций, как *ASP.NET Core Identity* для проверки подлинности.
- Папки bin и obj вмещают скомпилированные сборки для проекта.
- Папка Controllers содержит классы C# с методами (известными как действия), которые получают *модель* и передают ее *представлению*, например `HomeController.cs`.
- Папка Data вмещает классы миграции Entity Framework Core, используемые системой ASP.NET Core Identity для обеспечения аутентификации и авторизации, например `ApplicationDbContext.cs`.
- Папка Models содержит классы C#, которые представляют данные, собираемые контроллером и передаваемые в представление, например `ErrorViewModel.cs`.
- Папка Properties включает файл конфигурации для IIS или IIS Express и запуска сайта во время разработки — `launchSettings.json`. Этот файл используется только на локальной машине разработки и не может быть развернут на вашем рабочем сайте.

- Папка Views содержит файлы Razor с расширением `.cshtml`, которые комбинируют код HTML и C# для динамического генерирования HTML-ответов. Файл `_ViewStart` устанавливает макет по умолчанию, а `_ViewImports` импортирует общие пространства имен, используемые во всех представлениях, например «тег-хелперы».
  - Подпапка Home содержит файлы Razor для главной страницы и страниц конфиденциальности.
  - Подпапка Shared вмещает файлы Razor для общего макета, страницу ошибок и некоторые частичные представления для входа в систему, принятия политики конфиденциальности и управления файлами cookie.
- Папка wwwroot содержит статический контент, используемый на сайте, такой как CSS-файлы для хранения стилей, изображения, скрипты JavaScript и файл `favicon.ico`. Вы также можете поместить сюда изображения и другие статические файловые ресурсы, такие как документы.
- В базе данных SQLite app.db хранятся зарегистрированные пользователи.
- Файлы `appsettings.json` и `appsettings.Development.json` содержат параметры, которые ваш сайт может загрузить во время выполнения, например строку подключения к базе данных для системы ASP.NET Identity и уровни ведения логов.
- Файл `NorthwindMvc.csproj` содержит параметры проекта, такие как использование пакета SDK для .NET, запись, гарантирующую, что файл `app.db` будет скопирован в выходную папку сайта, и список пакетов NuGet, которые требуются вашему проекту, в том числе:
  - `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore`;
  - `Microsoft.AspNetCore.Identity.EntityFrameworkCore`;
  - `Microsoft.AspNetCore.Identity.UI`;
  - `Microsoft.EntityFrameworkCore.Sqlite`;
  - `Microsoft.EntityFrameworkCore.Tools`.
- Файл `Program.cs` определяет класс, который содержит основную точку входа, создающую конвейер для обработки входящих HTTP-запросов, и определяет параметры по умолчанию для хостинга сайта, такие как настройка веб-сервера Kestrel и загрузка настроек приложений. При создании хоста он вызывает метод `UseStartup<T>()`, чтобы указать другой класс, способный выполнить дополнительную настройку.
- Файл `Startup.cs` добавляет и настраивает сервисы, необходимые вашему сайту, например ASP.NET Identity для проверки подлинности, SQLite для хранения данных и т. д., а также занимается связями между компонентами в вашем приложении.

Более подробную информацию о конфигурации веб-хостов можно получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/host/web-host.

Обзор базы данных ASP.NET Core Identity

Если вы установили инструмент SQLite, такой как *SQLiteStudio*, то можете открыть базу данных и просмотреть таблицы, которые система ASP.NET Core Identity задействует для регистрации пользователей и ролей, включая таблицу `AspNetUsers`, применяемую для хранения зарегистрированных пользователей (рис. 16.4).

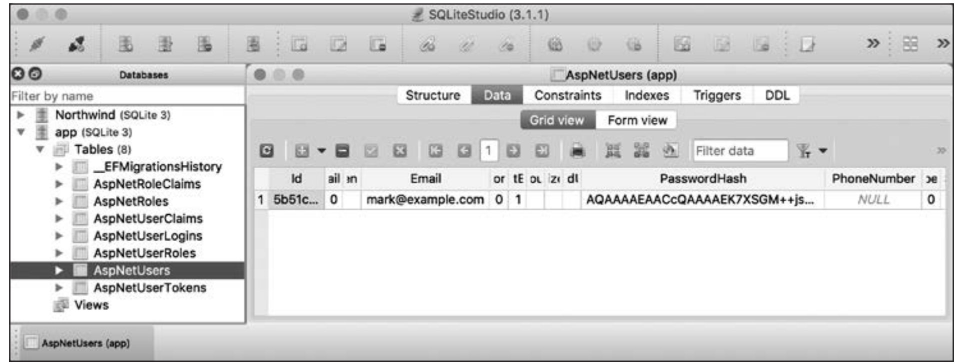


Рис. 16.4. Просмотр зарегистрированных пользователей в базе приложения

Шаблон проекта ASP.NET Core MVC следует обычной практике, храня хеш пароля вместо самого пароля, о чем говорилось в главе 10.

Изучение сайта ASP.NET Core MVC

Рассмотрим компоненты, из которых состоит современный сайт ASP.NET Core MVC.

Запуск ASP.NET Core MVC

Начнем с изучения конфигурации запуска по умолчанию сайта MVC.

1. Откройте файл `Startup.cs`.
2. Обратите внимание, что метод Configuration предназначен только для чтения. Данный метод можно передать и установить в конструкторе класса, как показано ниже:

```
public Startup(IConfiguration configuration)
{
  Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

3. Обратите внимание: метод `ConfigureServices` добавляет в приложение в качестве хранилища данных контекст базы данных, использующий SQLite со строкой подключения к базе, загруженной из файла `appsettings.json`, далее добавляет ASP.NET Identity в целях аутентификации и настраивает его для использования базы данных приложения. Кроме того, этот метод добавляет поддержку контроллеров MVC с представлениями, а также Razor Pages, как показано ниже:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(
      Configuration.GetConnectionString("DefaultConnection")));

  services.AddDatabaseDeveloperPageExceptionFilter();

  services.AddDefaultIdentity<IdentityUser>(options =>
    options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

  services.AddControllersWithViews();
}
```

Мы не будем создавать Razor Pages в этой главе, однако нам необходимо оставить вызов метода, который добавляет поддержку Razor Page, поскольку для аутентификации и авторизации наш сайт MVC применяет ASP.NET Core Identity, а также библиотеку классов Razor для компонентов пользовательского интерфейса, таких как регистрация пользователей и вход в систему.

Дополнительную информацию о библиотеке Identity UI, распространяемой в виде библиотеки классов Razor, можно получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/security/authentication/scaffold-identity?tabs=netcore-cli.

Вызов метода `AddDbContext` — пример регистрации сервиса-зависимости. ASP.NET Core реализует паттерн проектирования *«Внедрение зависимостей»* (dependency injection, DI), вследствие чего контроллеры могут запрашивать необходимые сервисы через свои конструкторы. Разработчики регистрируют эти сервисы в методе `ConfigureServices`.

Более подробно о внедрении зависимостей можно узнать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/dependency-injection.

4. Далее вызывается метод `Configure`, который настраивает подробную страницу исключений и ошибок базы данных, если сайт находится в разработке, или более удобную страницу ошибок и HSTS для производственной среды. Включены HTTPS-перенаправление, статические файлы, маршрутизация и ASP.NET Identity, и настроены маршрут MVC по умолчанию и Razor Pages, как показано ниже:

```
public void Configure(IApplicationBuilder app,
  IWebHostEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
  else
  {
    app.UseExceptionHandler("/Home/Error");
    // значение HSTS по умолчанию составляет 30 дней
    app.UseHsts();
  }
  app.UseHttpsRedirection();
  app.UseStaticFiles();

  app.UseRouting();

  app.UseAuthentication();
  app.UseAuthorization();

  app.UseEndpoints(endpoints =>
  {
    endpoints.MapControllerRoute(
      name: "default",
      pattern: "{controller=Home}/{action=Index}/{id?}");

    endpoints.MapRazorPages();
  });
}
```

В главе 15 мы изучили большинство из этих методов. Помимо методов `UseAuthentication` и `UseAuthorization`, наиболее важный новый метод — `MapControllerRoute`, содержащийся в методе `Configure` и настраивающий маршрут по умолчанию на использование MVC. Этот маршрут очень гибкий, поскольку подходит практически для любого входящего URL, как описано в следующем подразделе.

Более подробную информацию о настройке промежуточного программного обеспечения вы получите на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/middleware/.

Маршрутизация по умолчанию

Маршрутизация заключается в том, чтобы обнаружить имя класса контроллера, который необходимо создать, и метод действия, который нужно выполнить для генерации HTTP-ответа.

Маршрут по умолчанию настроен для MVC:

```
endpoints.MapControllerRoute(
  name: "default",
  pattern: "{controller=Home}/{action=Index}/{id?}");
```

Части маршрута, записанные в фигурных скобках `{}`, называются *сегментами* и подобны именованным параметрам метода. Значение этих сегментов может быть любой строкой.

Этот шаблон предусматривает анализ всех URL, введенных пользователем в адресной строке, и их разбор для извлечения имен контроллера и действия, а также опционального значения `id` (таково оно благодаря символу `?`).

Если пользователь не указал эти данные, то применяются значения по умолчанию: `Home` в качестве контроллера и `Index` в качестве действия (присвоение через `=` устанавливает значение по умолчанию для сегмента с именем).

В табл. 16.1 приведены примеры URL и то, как маршрут по умолчанию будет определять имена контроллера и действия.

Сегменты в URL не чувствительны к регистру.

Таблица 16.1

URL-адрес	**Контроллер**	**Действие**	**ID**
/	Home	Index	—
/Muppet	Muppet	Index	—
/Muppet/Kermit	Muppet	Kermit	—
/Muppet/Kermit/Green	Muppet	Kermit	Green
/Products	Products	Index	—
/Products/Detail	Products	Detail	—
/Products/Detail/3	Products	Detail	3

Контроллеры и действия

В ASP.NET Core MVC символ C обозначает контроллер. По маршруту и входящему URL ASP.NET Core MVC определит имя контроллера, поэтому понадобится класс, помеченный атрибутом [Controller] или наследуемый от класса, помеченного этим атрибутом, — например, от показанного ниже класса ControllerBase:

```
namespace Microsoft.AspNetCore.Mvc
{
  //
  // Summary:
  // A base class for an MVC controller without view support.
  [Controller]
  public abstract class ControllerBase
...
```

Чтобы упростить задачу, корпорация Microsoft предоставила класс Controller, который могут наследовать ваши классы, если нужна поддержка представления.

Обязанности контроллера заключаются в следующем:

- определение в конструкторе (или конструкторах) сервисов, необходимых контроллеру для того, чтобы он находился в допустимом состоянии и правильно функционировал;
- использование имени action для определения метода выполнения;
- извлечение параметров из HTTP-запроса;
- использование параметров в целях получения любых дополнительных данных, необходимых для построения модели представления, и передачи их в соответствующее представление для клиента. Например, если клиент — это браузер, то представление, которое отображает HTML, будет наиболее подходящим. Другие клиенты могут предпочесть альтернативные визуализации, такие как форматы документов, например, PDF- или Excel-файл либо форматы данных, например, JSON или XML;
- возврат результатов из представления клиенту в качестве HTTP-ответа с соответствующим кодом состояния.

Рассмотрим контроллер, который использовался для генерации главной страницы, а также страниц конфиденциальности и ошибок.

1. Разверните папку Controllers.
2. Найдите и откройте файл HomeController.cs.
3. Обратите внимание на следующие моменты, показанные в коде ниже:
   - скрытое поле объявляется в целях хранения ссылки на установленный в конструкторе логер для класса HomeController;

- все три метода действия вызывают метод `View()` и возвращают клиенту результаты в виде `IActionResult`;
- метод действия `Error` передает модель представления в свое представление с идентификатором запроса, используемым для трассировки. Ответ об ошибке не будет кэшироваться.

```
public class HomeController : Controller
{
  private readonly ILogger<HomeController> _logger;

  public HomeController(ILogger<HomeController> logger)
  {
    _logger = logger;
  }

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

  public IActionResult Privacy()
  {
    return View();
  }

  [ResponseCache(Duration = 0,
    Location = ResponseCacheLocation.None, NoStore = true)]
  public IActionResult Error()
  {
    return View(new ErrorViewModel { RequestId =
      Activity.Current?.Id ?? HttpContext.TraceIdentifier });
  }
}
```

Если пользователь вводит символ `/` или `/Home`, то получается эквивалент `/Home/Index`, поскольку это значения по умолчанию.

Соглашение о пути поиска представлений

Методы `Index` и `Privacy` имеют одинаковую реализацию, но в результате возвращают разные веб-страницы. Это из-за соглашения. Вызов метода `View` выполняет поиск файла Razor по разным путям для создания веб-страницы.

1. В проекте NorthwindMvc разверните папку Views, а затем Home.
2. Переименуйте файл `Privacy.cshtml` в `Privacy2.cshtml`.
3. Чтобы запустить сайт, на панели TERMINAL (Терминал) введите команду `dotnet run`.

4. Запустите Google Chrome, перейдите по адресу http://localhost:5000/, нажмите Privacy (Конфиденциальность) и обратите внимание на пути, по которым выполняется поиск представления для визуализации веб-страницы, в том числе в общих папках.

```
InvalidOperationException: The view 'Privacy' was not found. The following
locations were searched:
/Views/Home/Privacy.cshtml
/Views/Shared/Privacy.cshtml
/Pages/Shared/Privacy.cshtml
```

5. Закройте браузер Google Chrome.
6. Переименуйте файл `Privacy2.cshtml` обратно в `Privacy.cshtml`.

Соглашение о пути поиска представлений продемонстрировано в следующем списке:

- конкретное представление Razor: `/Views/{controller}/{action}.cshtml`;
- общее представление Razor: `/Views/Shared/{action}.cshtml`;
- общая страница Razor: `/Pages/Shared/{action}.cshtml`.

Модульное тестирование MVC

В контроллерах выполняется бизнес-логика вашего сайта, поэтому важно проверить ее правильность с помощью модульных тестов, описанных в главе 4.

Более подробно о том, как выполнить модульное тестирование контроллеров, можно прочитать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/mvc/controllers/testing.

Фильтры

Когда вам понадобится добавить некоторые общие функции для нескольких контроллеров и действий, вы можете использовать или определить собственные фильтры, реализованные с помощью атрибутов.

Фильтры могут применяться на следующих уровнях:

- уровень действия путем добавления атрибута к методу. Это повлияет только на один метод;
- уровень контроллера путем добавления атрибута к классу. Это повлияет на все методы данного контроллера;

- глобальный уровень путем добавления экземпляра атрибута в коллекцию `Filters` интерфейса `IServiceCollection` в методе `ConfigureServices` класса `Startup`. Это повлияет на все методы всех контроллеров в проекте.

Более подробно о фильтрах можно прочитать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/mvc/controllers/filters.

Использование фильтра для защиты метода действия

Например, вы можете убедиться, что один конкретный метод контроллера может вызываться только членами определенных ролей безопасности. Это можно сделать, дополнив метод атрибутом `[Authorize]`:

```
[Authorize(Roles = "Sales,Marketing")]
public IActionResult SalesAndMarketingEmployeesOnly()
{
  return View();
}
```

Более подробную информацию об авторизации можно получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/security/authorization/introduction.

Использование фильтра для кэширования ответа

Для кэширования HTTP-ответа, сгенерированного методом действия, к этому методу добавляется атрибут `[ResponseCache]`:

```
[ResponseCache(Duration = 3600, // в секундах = 1 час
  Location = ResponseCacheLocation.Any)]
public IActionResult AboutUs()
{
  return View();
}
```

Вы контролируете, где и насколько долго кэшируется ответ, устанавливая параметры:

- `Duration` — в секундах; устанавливает параметр `max-age` заголовка ответа HTTP;
- `Location` — одно из значений `ResponseCacheLocation`: `Any`, `Client` или `None`; устанавливает заголовок ответа HTTP для управления кэшем;
- `NoStore` — если значение равно `true`, то параметры `Duration` и `Location` игнорируются и задается заголовок ответа HTTP для элемента управления кэшем, равный `no-store`.

Более подробную информацию о кэшировании ответов можно узнать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/performance/caching/response.

Применение фильтра для переопределения маршрута

Возможно, вы захотите определить упрощенный маршрут для метода действия вместо того, чтобы использовать маршрут по умолчанию.

Например, для отображения страницы конфиденциальности в настоящее время требуется следующий URL, указывающий и контроллер, и действие:

```
https://localhost:5001/home/privacy
```

Чтобы упростить маршрут, можно дополнить метод действия:

```
[Route("private")]
public IActionResult Privacy()
{
  return View();
}
```

Теперь мы можем применить следующий URL с переопределенным маршрутом:

```
https://localhost:5001/private
```

Сущности и модели представлений

В ASP.NET Core MVC символ M обозначает модель. Это данные, необходимые для ответа на запрос. *Сущностные модели* представляют собой сущности в хранилище данных, таком как SQLite. На основании запроса может потребоваться извлечь оттуда одну или несколько сущностей. Все данные, которые мы хотим показать в ответ на запрос, — это модель MVC, которую иногда называют *моделью представления*, поскольку это *модель*, передающаяся в *представление* для его последующего перевода в формат ответа, такой как HTML или JSON.

Например, следующий HTTP-запрос `GET` может означать, что браузер запрашивает страницу сведений о товаре под номером 3:

```
http://www.example.com/products/details/3
```

Контроллеру необходимо использовать значение идентификатора 3, чтобы извлечь объект для этого товара и передать его представлению, которое затем превратит модель в HTML, чтобы отобразить в браузере.

Представьте: когда пользователь заходит на наш сайт, мы хотим показать ему карусель категорий, список товаров и количество посетителей за определенный месяц.

Мы будем ссылаться на сущностную модель данных Entity Framework Core для базы данных Northwind, созданной в главе 14.

1. В проекте NorthwindMvc найдите и откройте файл NorthwindMvc.csproj.
2. Добавьте ссылку на библиотеку NorthwindContextLib, как показано в следующем фрагменте кода:

```
<ItemGroup>
  <ProjectReference Include=
    "..\NorthwindContextLib\NorthwindContextLib.csproj" />
</ItemGroup>
```

3. На панели TERMINAL (Терминал) введите следующую команду, чтобы собрать проект:

```
dotnet build
```

4. Отредактируйте файл Startup.cs, чтобы импортировать пространства имен System.IO и Packt.Shared. Для настройки контекста базы данных Northwind добавьте в метод ConfigureServices оператор:

```
string databasePath = Path.Combine("..", "Northwind.db");

services.AddDbContext<Northwind>(options =>
  options.UseSqlite($"Data Source={databasePath}"));
```

5. Добавьте в папку Models файл класса и назовите его HomeIndexViewModel.cs.

Хотя класс ErrorViewModel, созданный шаблоном проекта MVC, не соответствует этому соглашению, я рекомендую использовать соглашение об именовании {Controller}{Action}ViewModel для классов модели представления.

6. Измените определение класса, чтобы создать три свойства для подсчета количества посетителей, получения списков категорий и товаров:

```
using System.Collections.Generic;
using Packt.Shared;

namespace NorthwindMvc.Models
{
  public class HomeIndexViewModel
  {
    public int VisitorCount;
    public IList<Category> Categories { get; set; }
    public IList<Product> Products { get; set; }
  }
}
```

7. Откройте класс HomeController.

8. Импортируйте пространство имен `Packt.Shared`.
9. Добавьте поле для хранения ссылки на экземпляр `Northwind` и инициализируйте его в конструкторе, как показано ниже (выделено полужирным шрифтом):

```
public class HomeController : Controller
{
  private readonly ILogger<HomeController> _logger;

  private Northwind db;

  public HomeController(ILogger<HomeController> logger,
    Northwind injectedContext)
  {
    _logger = logger;
    db = injectedContext;
  }
```

   ASP.NET Core будет использовать инъекцию через параметры конструктора для передачи экземпляра контекста БД `Northwind`, использующего путь к базе данных, указаный вами в классе `Startup`.

10. Измените содержимое метода действия `Index`, чтобы создать для него экземпляр модели представления, имитируя счетчик посетителей с помощью класса `Random` для генерации числа от 1 до 1000 и базы данных `Northwind` для получения списков категорий и товаров:

```
var model = new HomeIndexViewModel
{
  VisitorCount = (new Random()).Next(1, 1001),
  Categories = db.Categories.ToList(),
  Products = db.Products.ToList()
};

return View(model); // передача модели представлению
```

Когда метод `View()` вызывается в методе действия контроллера, ASP.NET Core MVC ищет в папке Views подпапку с тем же именем, что и у текущего контроллера, то есть Home. Затем ищет файл с тем же именем, что и у текущего действия, то есть `Index.cshtml`.

Представления

В ASP.NET Core MVC символ V обозначает представление. Оно несет ответственность за преобразование модели в HTML-код и другие форматы.

Для этого можно применить несколько *обработчиков представления*. Обработчик представления по умолчанию называется *Razor* и использует символ `@`, чтобы обозначить выполнение кода на стороне сервера.

Технология Razor Pages, представленная в ASP.NET Core 2.0, использует тот же обработчик представления и поэтому может применять тот же синтаксис Razor.

Чтобы отобразить списки категорий и товаров, изменим представление главной страницы.

1. Разверните папку Views, а затем папку Home.
2. Откройте файл `Index.cshtml` и обратите внимание на код C#, заключенный в блок `@{}`. Этот блок будет выполнен в первую очередь и может использоваться для хранения данных, которые должны быть переданы в общий файл макета, такой как заголовок веб-страницы:

```
@{
  ViewData["Title"] = "Home Page";
}
```

3. Обратите внимание на статический HTML-контент в элементах `<div>`, для стилизации которых используется библиотека Bootstrap.

При необходимости определить собственные стили старайтесь создавать их на основе универсальной общей библиотеки, такой как Bootstrap, реализующей принципы адаптивного дизайна.

Как и в случае Razor Pages, существует файл `_ViewStart.cshtml`, который запускается из метода `View()`. Он используется для установки значений по умолчанию, применяемых ко всем представлениям.

Например, он настраивает свойство `Layout` всех представлений на отображение общего файла макета:

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

4. В папке Views откройте файл `_ViewImports.cshtml` и обратите внимание, что импортируются некоторые пространства имен, а затем добавляются «тег-хелперы» ASP.NET Core:

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

5. Найдите в папке Shared файл `_Layout.cshtml` и откройте.
6. Обратите внимание: заголовок читается из словаря `ViewData`, который был установлен ранее в представлении `Index.cshtml`.

```
<title>@ViewData["Title"] - NorthwindMvc</title>
```

7. Обратите внимание на ссылки для поддержки Bootstrap и таблиц стилей сайта, где символ ~ обозначает папку wwwroot:

```
<link rel="stylesheet"
      href="~/lib/bootstrap/dist/css/bootstrap.css" />
<link rel="stylesheet" href="~/css/site.css" />
```

8. Обратите внимание на панель навигации в заголовке:

```
<body>
  <header>
    <nav class="navbar ...">
```

9. Обратите внимание на элемент `<div>`, содержащий частичное представление для входа в систему и гиперссылки, позволяющие пользователям перемещаться между страницами:

```
<div class=
  "navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
  <partial name="_LoginPartial" />
  <ul class="navbar-nav flex-grow-1">
    <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="Privacy">Privacy</a>
    </li>
  </ul>
</div>
```

Для элементов `<a>` использованы «тег-хелперы» с именами `asp-controller` и `asp-action`, чтобы указать имена контроллера и действия, которые должны выполниться при переходе по ссылке. Если хотите перейти к объекту в библиотеке классов Razor, то задействуйте `asp-area` для указания имени объекта.

10. Обратите внимание на отрисовку основной части страницы (body) внутри элемента `<main>`:

```
<div class="container">
  <main role="main" class="pb-3">
    @RenderBody()
  </main>
</div>
```

Вызов метода `@RenderBody()` внедряет содержимое определенного представления Razor для страницы, например файла `Index.cshtml`, в общий макет.

О том, почему лучше расположить элементы <script> ниже элемента <body>, можно узнать на сайте https://stackoverflow.com/questions/436411/where-should-i-put-script-tags-in-html-markup.

11. Обратите внимание на элементы `<script>` в нижней части страницы — таким образом не замедляется отображение страницы, а вы можете добавлять собственные блоки скриптов в определенную опциональной секцию `scripts`, как показано в следующем фрагменте кода:

```
<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js">
</script>
<script src="~/js/site.js" asp-append-version="true"></script>
@await RenderSectionAsync("scripts", required: false)
```

Когда «тег-хелпер» `asp-append-version` указывается со значением `true` в любом элементе вместе с атрибутом `src`, вызывается Image Tag Helper (он влияет не только на изображения!).

Его работа заключается в добавлении параметра запроса `v` со значением, которое генерируется из хеша файла, указанного в `src`:

```
<script src="~/js/site.js?
v=Kl_dqr9NVtnMdsM2MUg4qthUnWZm5T1fCEimBPWDNgM"></script>
```

Изменение даже одного байта в файле `site.js` меняет его хеш, и, как следствие, если браузер или CDN кэшируют файл со скриптом, то кэшированная копия обесценнится и будет заменена новой версией.

Каким образом работает очистка кэша с помощью параметров запроса, можно прочитать на сайте https://stackoverflow.com/questions/9692665/cache-busting-via-params.

Добавление собственного функционала на сайт ASP.NET Core MVC

Теперь, проанализировав структуру базового сайта MVC, вы сможете добавить к нему собственный функционал. Вы уже добавили код для извлечения объектов из базы данных `Northwind`, и потому следующая задача — вывести эту информацию на главную страницу.

Чтобы добавить подходящие изображения для восьми категорий, я нашел их на сайте, на котором предоставляются бесплатные стоковые фотографии для коммерческого использования без указания авторства: https://www.pexels.com/.

Определение пользовательских стилей

На главной странице будет отображен список из 77 наименований товаров базы данных Northwind. Чтобы эффективно использовать пространство, мы отобразим список в трех столбцах. Для этого нам необходимо настроить таблицу стилей для сайта.

1. В папке wwwroot\css найдите и откройте файл site.css.
2. В конце файла добавьте новый стиль, который будет применен к элементу с идентификатором newspaper:

```
#newspaper
{
  column-count: 3;
}
```

Настройка категории изображений

База данных Northwind включает в себя таблицу категорий, однако не содержит изображений, а сайты выглядят лучше, если на них есть яркие картинки.

1. Создайте в папке wwwroot папку images.
2. В папку images добавьте восемь файлов изображений: category1.jpeg, category2.jpeg и т. д. до category8.jpeg.

Скачать изображения из репозитория GitHub для этой книги можно, перейдя по следующей ссылке: https://github.com/markjprice/cs9dotnet5/tree/master/Assets/Categories.

Синтаксис Razor

Прежде чем настраивать вид главной страницы, рассмотрим пример файла Razor, имеющего начальный кодовый блок Razor, который создает экземпляр заказа с ценой и количеством, а затем выводит сведения о заказе на веб-странице, как показано ниже:

```
@{
  var order = new Order
  {
    OrderID = 123,
    Product = "Sushi",
    Price = 8.49M,
    Quantity = 3
  };
}
<div>Your order for @order.Quantity of @order.Product has a total cost
of $@order.Price * @order.Quantity</div>
```

Предыдущий Razor-файл приведет к следующему неверному выводу:

```
Your order for 3 of Sushi has a total cost of $8.49 * 3
```

Хотя разметка Razor может включать значение любого отдельного свойства с помощью синтаксиса `@object.property`, выражения необходимо заключать в скобки, как показано ниже:

```
<div>Your order for @order.Quantity of @order.Product has a total cost
of $@(order.Price * @order.Quantity)</div>
```

Предыдущее выражение Razor приведет к следующему правильному выводу:

```
Your order for 3 of Sushi has a total cost of $25.47
```

Определение типизированного представления

Если с помощью директивы `@model` определить ожидаемый представлением тип, то это улучшит работу IntelliSense при написании кода представления.

1. В папке Views\Home найдите и откройте файл `Index.cshtml`.
2. В начале файла добавьте оператор, чтобы установить `HomeIndexViewModel` в качестве используемого типа модели:

   ```
   @model NorthwindMvc.Models.HomeIndexViewModel
   ```

   Каждый раз при вводе `Model` в этом представлении расширение C# для Visual Studio Code будет знать подходящий тип и предоставит для него IntelliSense.

   Вводя код в представление, помните следующее:

   - чтобы объявить тип для модели, следует писать `@model` (со строчной буквой `m`);
   - для взаимодействия с экземпляром модели следует писать `@Model` (с прописной буквой `M`).

   Продолжим переделывать главную страницу.

3. В исходном блоке кода Razor добавьте оператор, объявляющий строковую переменную для текущего элемента, и замените существующий элемент `<div>` новой разметкой, чтобы вывести категории в карусели и товары в виде неупорядоченного списка:

   ```
   @model NorthwindMvc.Models.HomeIndexViewModel
   @{
     ViewData["Title"] = "Home Page";
     string currentItem = "";
   }
   <div id="categories" class="carousel slide" data-ride="carousel"
     data-interval="3000" data-keyboard="true">
   ```

```
<ol class="carousel-indicators">
@for (int c = 0; c < Model.Categories.Count; c++)
{
  if (c == 0)
  {
    currentItem = "active";
  }
  else
  {
    currentItem = "";
  }
  <li data-target="#categories" data-slide-to="@c"
    class="@currentItem"></li>
}
</ol>
<div class="carousel-inner">
@for (int c = 0; c < Model.Categories.Count; c++)
{
  if (c == 0)
  {
    currentItem = "active";
  }
  else
  {
    currentItem = "";
  }
  <div class="carousel-item @currentItem">
    <img class="d-block w-100" src=
      "~/images/category@(Model.Categories[c].CategoryID).jpeg"
      alt="@Model.Categories[c].CategoryName" />
    <div class="carousel-caption d-none d-md-block">
      <h2>@Model.Categories[c].CategoryName</h2>
      <h3>@Model.Categories[c].Description</h3>
      <p>
        <a class="btn btn-primary"
          href="/category/@Model.Categories[c].CategoryID">View</a>
      </p>
    </div>
  </div>
}
</div>
<a class="carousel-control-prev" href="#categories"
  role="button" data-slide="prev">
  <span class="carousel-control-prev-icon"
    aria-hidden="true"></span>
  <span class="sr-only">Previous</span>
</a>
<a class="carousel-control-next" href="#categories"
  role="button" data-slide="next">
  <span class="carousel-control-next-icon"
    aria-hidden="true"></span>
  <span class="sr-only">Next</span>
```

```
  </a>
</div>
<div class="row">
  <div class="col-md-12">
    <h1>Northwind</h1>
    <p class="lead">
      We have had @Model.VisitorCount visitors this month.
    </p>
    <h2>Products</h2>
    <div id="newspaper">
      <ul>
      @foreach (var item in @Model.Products)
      {
        <li>
          <a asp-controller="Home"
            asp-action="ProductDetail"
            asp-route-id="@item.ProductID">
            @item.ProductName costs
            @item.UnitPrice.ToString("C")
          </a>
        </li>
      }
      </ul>
    </div>
  </div>
</div>
```

Просматривая предыдущую разметку Razor, обратите внимание на следующие моменты.

- Статические элементы HTML, такие как `<ul>` и `<li>`, можно легко смешивать с кодом на языке C#, чтобы выводить карусель категорий и список наименований товаров.
- К элементу `<div>` с идентификатором `newspaper` применен пользовательский стиль, определенный нами ранее, поэтому весь контент данного элемента будет выведен в три столбца.
- В элементе `<img>` для каждой категории используются круглые скобки вокруг выражения Razor с целью гарантировать, что компилятор не рассматривает расширение `.jpeg` как часть выражения:

```
"~/images/category@(Model.Categories[c].CategoryID).jpeg"
```

- URL в элементах `<a>` для ссылок на товары создются с помощью «тег-хелперов». Эти гиперссылки будут обрабатываться контроллером `Home` и его методом действия `ProductDetail`. Данный метод действия еще не существует, но мы его добавим в текущей главе позже. Идентификатор товара передается в виде сегмента маршрута с именем `id`, как показано в следующем URL для Ipoh Coffee:

```
https://localhost:5001/Home/ProductDetail/43
```

Тестирование измененной главной страницы

Давайте посмотрим, как в итоге выглядит наша главная страница.

1. Запустите сайт, используя следующую команду: `dotnet run`.
2. Запустите браузер Google Chrome. В адресной строке введите следующий URL: http://localhost:5000.
3. Обратите внимание: на главной странице имеется вращающаяся карусель, которая отображает категории, случайное количество посетителей и список товаров в трех столбцах (рис. 16.5).

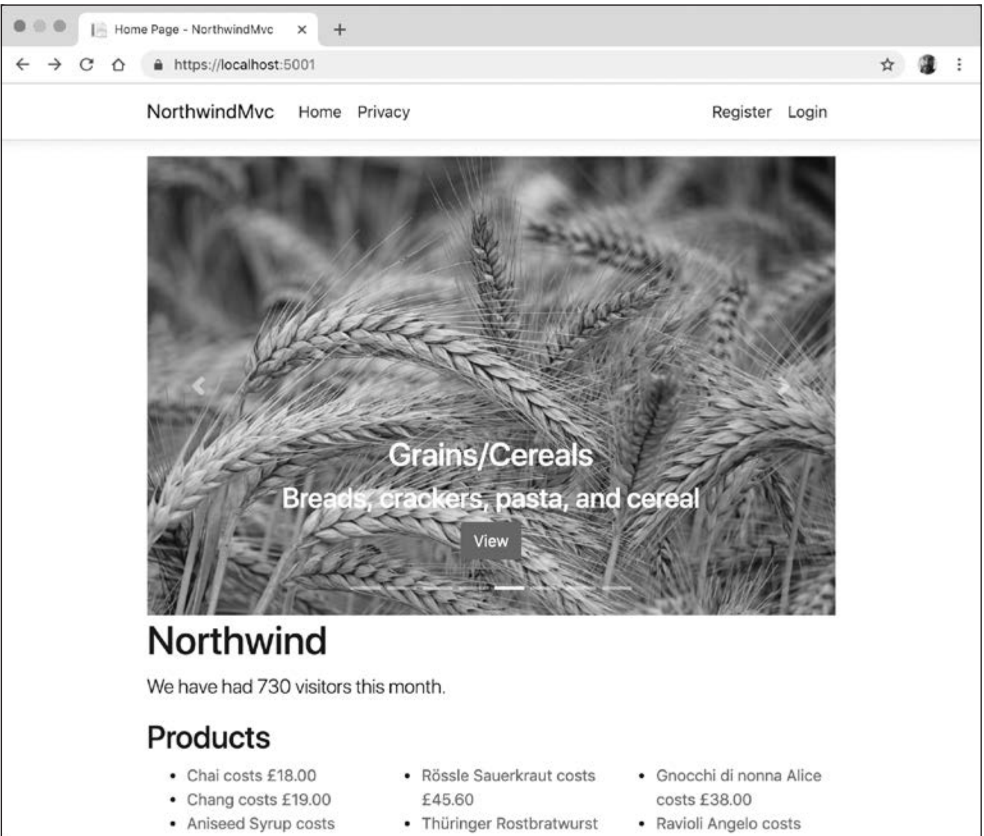


Рис. 16.5. Обновленная домашняя страница сайта Northwind MVC

В настоящий момент нажатие любой из категорий или ссылок на товары приводит к ошибкам `404 Not Found`, поэтому давайте разберемся, как передавать параметры для отображения сведений о товаре или категории.

4. Закройте браузер Google Chrome.

5. Остановите работу сайта, нажав сочетание клавиш Ctrl+C на панели TERMINAL (Терминал).

Передача параметров с помощью значения маршрута

Один из способов передать простой параметр — использовать сегмент `id`, определенный в маршруте по умолчанию.

1. В классе `HomeController` добавьте метод действия `ProductDetail`, как показано ниже:

```
public IActionResult ProductDetail(int? id)
{
  if (!id.HasValue)
  {
    return NotFound("You must pass a product ID in the route, for
example, /Home/ProductDetail/21");
  }

  var model = db.Products
    .SingleOrDefault(p => p.ProductID == id);

  if (model == null)
  {
    return NotFound($"Product with ID of {id} not found.");
  }
  return View(model); // передача модели представлению
                      // с последующим возвратом результата
}
```

   Обратите внимание на следующие моменты.

   - В методе `ProductDetail` используется функция ASP.NET Core, называемая *привязкой модели*, для автоматического сопоставления идентификатора, переданного в маршруте, с идентификатором `id` в методе.
   - В методе проверяется, присвоено ли идентификатору значение `null`, и если да, то мы вызываем метод `NotFound`, чтобы вернуть код состояния `404` и сообщение с правильным форматом URL.
   - В противном случае можно подключиться к базе данных и попытаться запросить товар по переменной `id` (идентификатору).
   - Если товар найден, то передается в представление; в противном случае мы вызываем метод `NotFound`, чтобы вернуть код состояния `404` и сообщение, объясняющее, что продукт с этим ID не был найден в базе данных.

   Если представление названо в соответствии с методом действия и помещено в папку, соответствующую имени контроллера, то соглашения ASP.NET Core MVC найдут его автоматически.

2. Добавьте в папку Views/Home новый файл `ProductDetail.cshtml`.

3. Отредактируйте содержимое созданного файла:

```
@model Packt.Shared.Product
@{
  ViewData["Title"] = "Product Detail - " + Model.ProductName;
}
<h2>Product Detail</h2>
<hr />
<div>
  <dl class="dl-horizontal">
    <dt>Product ID</dt>
    <dd>@Model.ProductID</dd>
    <dt>Product Name</dt>
    <dd>@Model.ProductName</dd>
    <dt>Category ID</dt>
    <dd>@Model.CategoryID</dd>
    <dt>Unit Price</dt>
    <dd>@Model.UnitPrice.Value.ToString("C")</dd>
    <dt>Units In Stock</dt>
    <dd>@Model.UnitsInStock</dd>
  </dl>
</div>
```

4. Запустите сайт, используя следующую команду: `dotnet run`.
5. Запустите браузер Google Chrome. В адресной строке введите следующий URL: http://localhost:5000.
6. Когда появится главная страница со списком товаров, выберите один из них, например товар под номером 2 — Chang.
7. Обратите внимание на URL в адресной строке браузера, заголовок страницы, отображаемый на вкладке браузера, и страницу сведений о товаре (рис. 16.6).

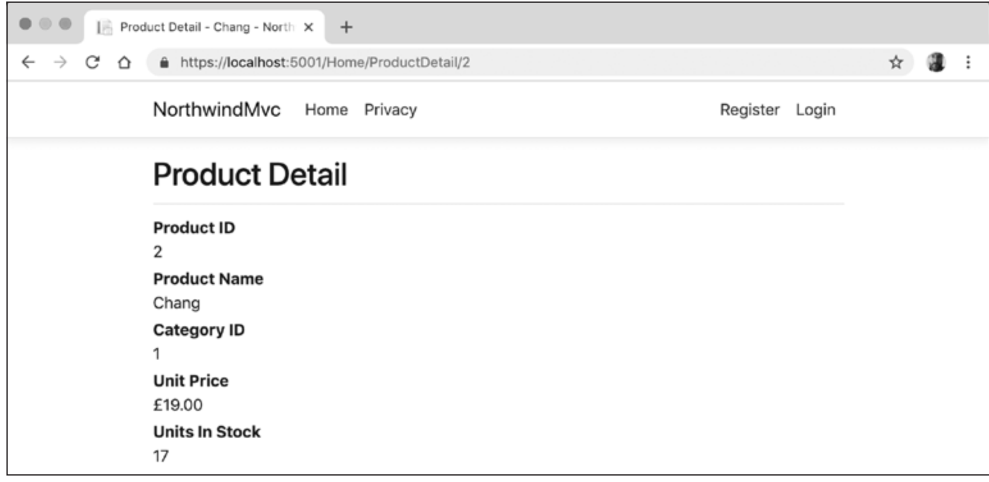


Рис. 16.6. Страница сведений о товаре

8. Переключитесь на панель инструментов разработчика.
9. Измените URL в адресной строке браузера, чтобы запросить идентификатор несуществующего товара, например 99, и обратите внимание на код состояния 404 Not Found и сообщение (рис. 16.7).

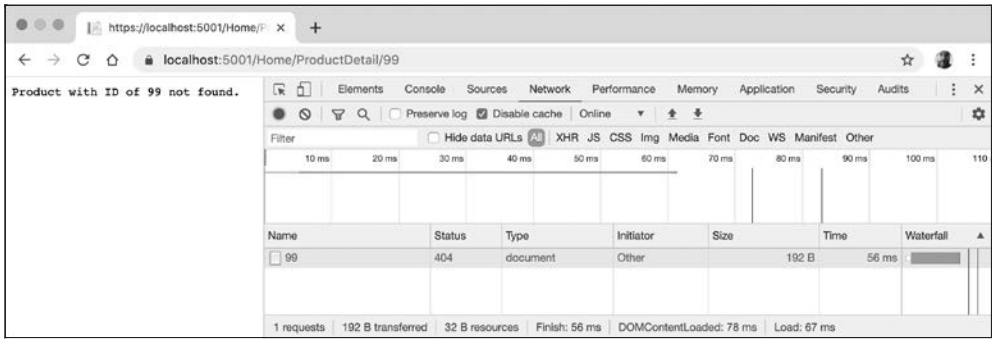


Рис. 16.7. Запрос несуществующего идентификатора товара

Привязка моделей

Привязка моделей (model binders) — очень мощный инструмент и по умолчанию многое способен выполнить автоматически. После того как маршрут по умолчанию идентифицирует класс контроллера для создания экземпляра и метод действия для вызова, при наличии у данного метода параметров, для них надо установить значения.

Для этого привязка моделей проанализирует значения переданных в HTTP-запросе параметров всех описанных ниже типов:

- параметр маршрута, такой как id, созданный в предыдущем разделе, как показано в следующем URL: /Home/ProductDetail/2;
- параметр строки запроса, как показано в следующем URL: /Home/ProductDetail?Id=2;
- параметр формы, как показано ниже:

```
<form action="post" action="/Home/ProductDetail">
  <input type="text" name="id" />
  <input type="submit" />
</form>
```

Привязка моделей может установить значение практически любого типа:

- простые типы, такие как int, string, DateTime и bool;
- сложные типы, определяемые как class или struct;
- типы коллекций, такие как массивы и списки.

Создадим следующий пример для демонстрации того, что может быть достигнуто с помощью привязки моделей по умолчанию.

1. Добавьте в папку Models новый файл `Thing.cs`.
2. Отредактируйте содержимое созданного файла, чтобы определить класс с двумя свойствами: с именем `ID`, допускающим значение `null`, и строковой переменной `color`:

```
namespace NorthwindMvc.Models
{
  public class Thing
  {
    public int? ID { get; set; }
    public string Color { get; set; }
  }
}
```

3. Найдите файл `HomeController.cs`, откройте и добавьте два новых метода действия: один для отображения страницы с формой и один для отображения ее с параметром, используя новый тип модели:

```
public IActionResult ModelBinding()
{
  return View(); // страница с формой для отправки
}

public IActionResult ModelBinding(Thing thing)
{
  return View(thing); // отображение модели, полученной через привязку
}
```

4. В папку Views\Home добавьте новый файл `ModelBinding.cshtml`.
5. Отредактируйте содержимое созданного файла:

```
@model NorthwindMvc.Models.Thing
@{
  ViewData["Title"] = "Model Binding Demo";
}
<h1>@ViewData["Title"]</h1>
<div>
  Enter values for your thing in the following form:
</div>
<form method="POST" action="/home/modelbinding?id=3">
  <input name="color" value="Red" />
  <input type="submit" />
</form>
@if (Model != null)
{
<h2>Submitted Thing</h2>
```

```
<hr />
<div>
  <dl class="dl-horizontal">
    <dt>Model.ID</dt>
    <dd>@Model.ID</dd>
    <dt>Model.Color</dt>
    <dd>@Model.Color</dd>
  </dl>
</div>
}
```

6. Запустите сайт. В адресной строке браузера Google Chrome введите следующий URL: https://localhost:5001/home/modelbinding.
7. Обратите внимание на необработанное исключение о неоднозначном совпадении (рис. 16.8).

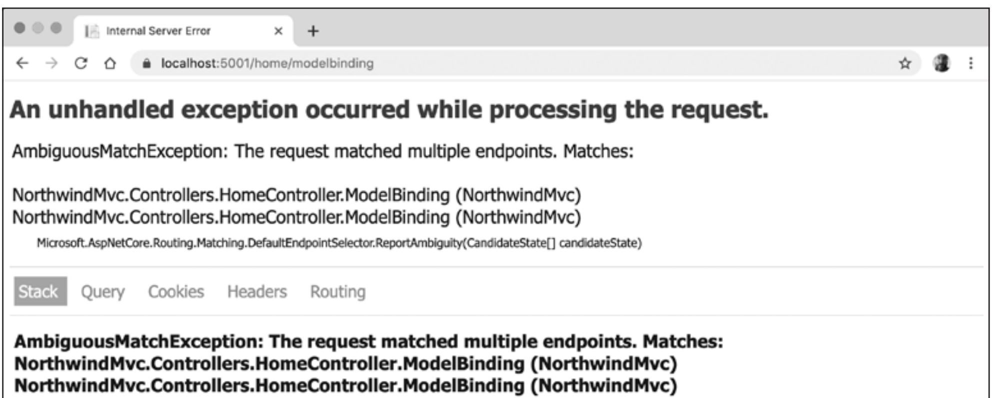


Рис. 16.8. Ошибка необработанного исключения

Компилятор C# может различать два метода, отметив, что сигнатуры различны, однако с точки зрения HTTP оба метода потенциально нам подходят. Нам необходим специфический для HTTP способ устранения неоднозначности методов действия. Мы могли бы сделать так, создав разные имена для действий или указав, что один метод должен использоваться для определенной HTTP-команды, такой как `GET`, `POST` или `DELETE`.

8. Остановите работу сайта.
9. В файле `HomeController.cs` добавьте ко второму методу действия `ModelBinding` индикатор того, что он используется для обработки HTTP-запросов `POST`, то есть при отправке формы, как показано ниже (выделено полужирным шрифтом):

```
[HttpPost]
public IActionResult ModelBinding(Thing thing)
```

10. Запустите сайт. В адресной строке браузера Google Chrome введите следующий URL: https://localhost:5001/home/modelbinding.
11. Нажмите кнопку Submit (Отправить) и обратите внимание, что значение свойства ID задается из параметра строки запроса, а свойства color — из параметра формы (рис. 16.9).

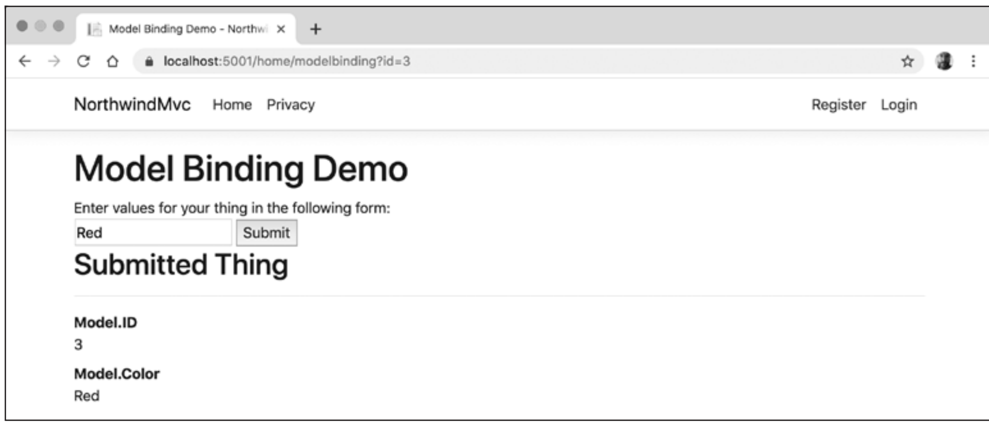


Рис. 16.9. Демонстрационная страница Model Binding Demo

12. Остановите работу сайта.
13. Измените действие для формы, чтобы передать значение 2 в качестве параметра маршрута, как показано ниже (выделено полужирным шрифтом):

```
<form method="POST" action="/home/modelbinding/2?id=3">
```

14. Запустите сайт. В адресной строке браузера Google Chrome введите следующий URL: https://localhost:5001/home/modelbinding.
15. Нажмите кнопку Submit (Отправить) и обратите внимание, что значение свойства ID задается из параметра маршрута, а свойства color — из параметра формы.
16. Остановите работу сайта.
17. Измените действие для формы, чтобы передать значение 1 в качестве параметра формы, как показано ниже (выделено полужирным шрифтом):

```
<form method="POST" action="/home/modelbinding/2?id=3">
  <input name="id" value="1" />
  <input name="color" value="Red" />
  <input type="submit" />
</form>
```

18. Запустите сайт. В адресной строке браузера Google Chrome введите следующий URL: https://localhost:5001/home/modelbinding.

19. Нажмите кнопку Submit (Отправить) и обратите внимание, что значения свойств `ID` и `color` задаются в параметрах формы.
20. При наличии нескольких параметров с одним и тем же именем параметры формы имеют самый высокий приоритет, а параметры строки запроса — самый низкий.

В сложных сценариях вы можете создать, изменить или дополнить механизм привязки моделей, реализовав интерфейс IModelBinder: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/advanced/custom-model-binding.

Проверка модели

Процесс привязки модели может вызвать ошибки, например, преобразования типов данных или ошибки проверки, если модель была дополнена правилами проверки. Привязанные данные, а также все ошибки привязки или проверки сохраняются в `ControllerBase.ModelState`.

Рассмотрим, что мы можем сделать с состоянием модели, применив некоторые правила проверки к привязанной модели и затем показав сообщения о недопустимых данных в представлении.

1. В папке Models найдите и откройте файл `Thing.cs`.
2. Импортируйте пространство имен `System.ComponentModel.DataAnnotations`.
3. Дополните свойство `ID` атрибутом проверки, чтобы ограничить диапазон разрешенных чисел от 1 до 10 и убедиться в предоставлении цвета пользователем, как показано ниже (выделено полужирным шрифтом):

```
public class Thing
{
  [Range(1, 10)]
  public int? ID { get; set; }

  [Required]
  public string Color { get; set; }
}
```

4. Добавьте в папку Models новый файл `HomeModelBindingViewModel.cs`.
5. Отредактируйте содержимое созданного файла, чтобы определить класс с двумя свойствами для связанной модели и всех ошибок:

```
using System.Collections.Generic;

namespace NorthwindMvc.Models
{
  public class HomeModelBindingViewModel
```

```
  {
    public Thing Thing { get; set; }
    public bool HasErrors { get; set; }
    public IEnumerable<string> ValidationErrors { get; set; }
  }
}
```

6. В папке Controllers найдите и откройте файл `HomeController.cs`.

7. Во втором методе `ModelBinding` закомментируйте предыдущий оператор, который передал объект представлению, и вместо этого добавьте операторы для создания экземпляра модели представления. Проверьте модель и сохраните массив сообщений об ошибках, а затем передайте модель представлению в следующем виде:

```
public IActionResult ModelBinding(Thing thing)
{
  // return View(thing); // возвращается параметр, привязанный к модели

  var model = new HomeModelBindingViewModel
  {
    Thing = thing,
    HasErrors = !ModelState.IsValid,
    ValidationErrors = ModelState.Values
      .SelectMany(state => state.Errors)
      .Select(error => error.ErrorMessage)
  };
  return View(model);
}
```

8. В папке Views\Home найдите и откройте файл `ModelBinding.cshtml`.

9. Измените объявление типа модели, чтобы использовать класс модели представления. Добавьте элемент `<div>` для отображения всех ошибок проверки модели. Измените код для вывода свойств класса `Thing`, поскольку модель представления изменилась, как показано ниже (выделено полужирным шрифтом):

```
@model NorthwindMvc.Models.HomeModelBindingViewModel
@{
  ViewData["Title"] = "Model Binding Demo";
}
<h1>@ViewData["Title"]</h1>
<div>
  Enter values for your thing in the following form:
</div>
<form method="POST" action="/home/modelbindingdemo/2?id=3">
  <input name="id" value="1" />
  <input name="color" value="Red" />
  <input type="submit" />
</form>
@if (Model != null)
{
```

```
<h2>Submitted Thing</h2>
<hr />
<div>
  <dl class="dl-horizontal">
    <dt>Model.Thing.ID</dt>
    <dd>@Model.Thing.ID</dd>
    <dt>Model.Thing.Color</dt>
    <dd>@Model.Thing.Color</dd>
  </dl>
</div>
@if (Model.HasErrors)
{
  <div>
  @foreach(string errorMessage in Model.ValidationErrors)
  {
    <div class="alert alert-danger" role="alert">@errorMessage</div>
  }
  </div>
}
}
```

10. Запустите сайт. В адресной строке браузера Google Chrome введите следующий URL: https://localhost:5001/home/modelbinding.
11. Нажмите кнопку Submit (Отправить) и обратите внимание, что значения `1` и `Red` — допустимые.
12. Введите значение ID (идентификатора) равным `99`, очистите цветное текстовое поле, нажмите кнопку Submit (Отправить) и обратите внимание на сообщения об ошибках (рис. 16.10).

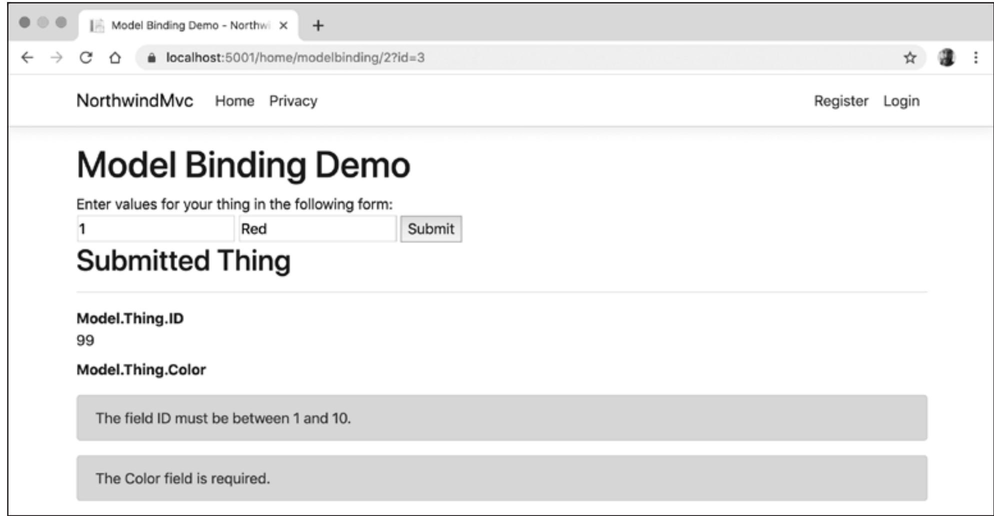


Рис. 16.10. Страница Model Binding Demo модели с полевыми проверками

13. Закройте браузер и остановите работу сайта.

Более подробно с информацией о проверке модели можно ознакомиться на сайте https://docs.microsoft.com/ru-ru/aspnet/core/mvc/models/validation.

Методы класса-помощника для представления

При создании представления для ASP.NET Core MVC вы можете использовать объект класса `Html` и его методы, чтобы создать разметку.

Некоторые полезные методы включают следующее.

- `ActionLink` используется для создания якорного элемента `<a>`, содержащего URL к указанному контроллеру и действию.
- `AntiForgeryToken` применяется внутри элемента `<form>`, чтобы вставить элемент `<hidden>`, содержащий маркер защиты от подделки, который будет проверен при отправке формы.

Более подробно о токенах для защиты от подделки можно ознакомиться на сайте https://docs.microsoft.com/ru-ru/aspnet/core/security/anti-request-forgery.

- `Display` и `DisplayFor` служат для создания HTML-разметки для выражения, относящегося к текущей модели, с помощью шаблона отображения. Существуют встроенные шаблоны отображения для типов .NET, а пользовательские можно создавать в папке DisplayTemplates. Имя папки чувствительно к регистру в чувствительных к регистру файловых системах.
- `DisplayForModel` используется для создания HTML-разметки для всей модели вместо одного выражения.
- `Editor` и `EditorFor` служат для создания HTML-разметки для выражения, относящегося к текущей модели, с помощью шаблона редактора. Существуют встроенные шаблоны редакторов для типов .NET, которые задействуют элементы `<label>` и `<input>`, а пользовательские шаблоны можно создавать в папке EditorTemplates. Имя папки чувствительно к регистру в чувствительных к регистру файловых системах.
- `EditorForModel` используется для создания HTML-разметки для всей модели вместо одного выражения.

- Encode применяется для безопасного кодирования объекта или строки в HTML-коде. Например, строковое значение "<script>" будет закодировано в виде "<script>". Обычно в этом нет необходимости, поскольку символ Razor @ по умолчанию кодирует строковые значения.
- Raw используется для отображения строкового значения *без* кодирования в HTML.
- PartialAsync и RenderPartialAsync применяются для создания HTML-разметки для частичного представления. При желании вы можете передать модель и данные представления (ViewData).

Более подробно о классе HtmlHelper можно прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.aspnetcore.mvc.viewfeatures.htmlhelper.

Отправка запросов в базу данных и использование шаблонов отображения

Создадим новый метод действия, которому будет передаваться параметр из строки запроса, и с его помощью создадим запрос к базе данных Northwind.

1. В классе HomeController импортируйте пространство имен Microsoft.EntityFrameworkCore. Это необходимо для использования метода расширения Include, чтобы мы могли включать связанные объекты, как описано в главе 11.
2. Добавьте новый метод действия:

```
public IActionResult ProductsThatCostMoreThan(decimal? price)
{
  if (!price.HasValue)
  {
    return NotFound("You must pass a product price in the query string,
for example, /Home/ProductsThatCostMoreThan?price=50");
  }

  IEnumerable<Product> model = db.Products
    .Include(p => p.Category)
    .Include(p => p.Supplier)
    .Where(p => p.UnitPrice > price);

  if (model.Count() == 0)
  {
    return NotFound(
      $"No products cost more than {price:C}.");
  }
```

```
  ViewData["MaxPrice"] = price.Value.ToString("C");
  return View(model); // передача модели представлению
}
```

3. В папку Views/Home добавьте новый файл `ProductsThatCostMoreThan.cshtml`.
4. Отредактируйте содержимое:

```
@model IEnumerable<Packt.Shared.Product>
@{
  string title =
    "Products That Cost More Than " + ViewData["MaxPrice"];
  ViewData["Title"] = title;
}
<h2>@title</h2>
<table class="table">
  <thead>
    <tr>
      <th>Category Name</th>
      <th>Supplier's Company Name</th>
      <th>Product Name</th>
      <th>Unit Price</th>
      <th>Units In Stock</th>
    </tr>
  </thead>
  <tbody>
  @foreach (var item in Model)
  {
    <tr>
      <td>
        @Html.DisplayFor(modelItem => item.Category.CategoryName)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Supplier.CompanyName)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.ProductName)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.UnitPrice)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.UnitsInStock)
      </td>
    </tr>
  }
  <tbody>
</table>
```

5. В папке Views/Home найдите и откройте файл `Index.cshtml`.

6. Добавьте элемент `form` после счетчика посетителей и перед заголовком `Products` и списком товаров. Это предоставит пользователю форму для ввода цены. Затем он может нажать кнопку отправки данных Submit (Отправить), чтобы вызвать метод действия, который выведет только товары с ценой, превышающей введенное значение:

```
<h3>Query products by price</h3>
<form asp-action="ProductsThatCostMoreThan" method="get">
  <input name="price" placeholder="Enter a product price" />
  <input type="submit" />
</form>
```

7. Запустите сайт, с помощью Google Chrome перейдите на него и на главной странице введите в форму цену товара, например `50`, а затем нажмите кнопку Submit (Отправить). Вы увидите таблицу товаров, цена которых превышает введенное значение (рис. 16.11).

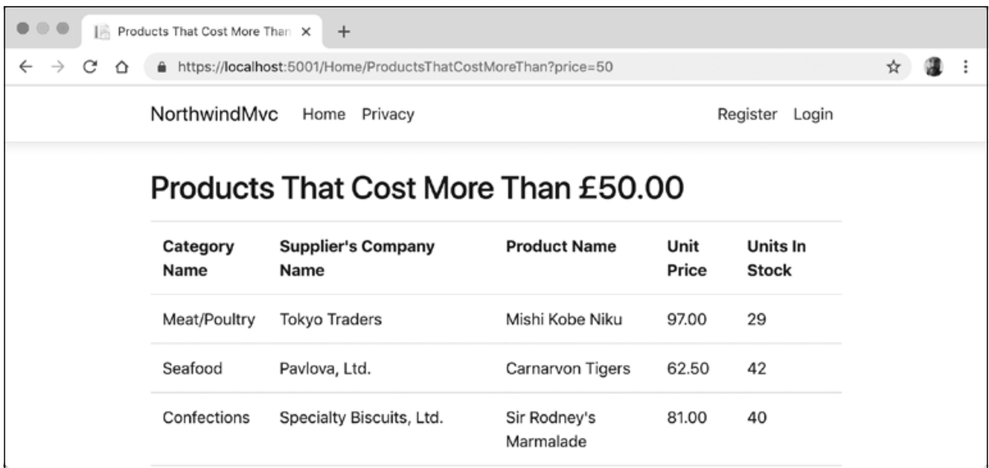


Рис. 16.11. Изменен список товаров стоимостью более 50 фунтов стерлингов

8. Закройте браузер и остановите работу сайта.

Улучшение масштабируемости с помощью асинхронных задач

При создании настольного или мобильного приложения повысить скорость отклика можно с помощью нескольких задач (и потоков, на основе которых они работают), поскольку, пока один поток занят задачей, другой может обрабатывать взаимодействия с пользователем.

Задачи и их потоки также могут быть полезны на стороне сервера, особенно на сайтах, работающих с файлами или запрашивающих данные из хранилища или веб-сервиса, ведь ответ на такой запрос может потребовать какого-то времени. Однако они небезопасны для сложных вычислений, производимых процессором, поэтому выполняйте такие задачи синхронно в обычном режиме.

Когда HTTP-запрос поступает на веб-сервер, поток из его пула выделяется для обработки запроса. Но если этот поток должен ожидать ресурс, то блокируется для обработки всех входящих запросов. Если сайт получает больше одновременных запросов, чем есть потоков в его пуле, то на некоторые из этих запросов будет отправлено сообщение об ошибке тайм-аута сервера: `503 Service Unavailable` (Сервис недоступен).

Блокированные потоки не выполняют полезной работы. Они *могут* обработать один из других запросов, но только если мы внедрим асинхронный код на наших сайтах.

Ожидая нужный ему ресурс, поток может вернуться в пул потоков и обработать другой входящий запрос, что улучшает масштабируемость сайта, то есть увеличивает количество одновременных запросов, которые он может обработать.

Почему бы просто не иметь больший пул потоков? В современных операционных системах каждый поток пула потоков содержит стек емкостью 1 Мбайт. Асинхронный метод использует меньший объем памяти. Благодаря этому также устраняется необходимость создания новых потоков в пуле, что занимает много времени. Скорость, с которой новые потоки добавляются в пул, обычно составляет один раз в каждые две секунды, что, по сравнению с переключением между асинхронными потоками, очень долго.

Превращение методов действия контроллера в асинхронные

Существующий метод действия легко сделать асинхронным.

1. Убедитесь, что пространство имен `System.Threading.Tasks` было импортировано в класс `HomeController`.
2. Измените метод действия `Index` на асинхронный, чтобы он возвращал `Task<T>` и ожидал вызовы асинхронных методов для получения категорий и товаров, как показано ниже (выделено полужирным шрифтом):

```
public async Task<IActionResult> Index()
{
  var model = new HomeIndexViewModel
  {
    VisitorCount = (new Random()).Next(1, 1001),
```

```
    Categories = await db.Categories.ToListAsync(),
    Products = await db.Products.ToListAsync()
  };
  return View(model); // передача модели представлению
}
```

3. Аналогичным образом измените метод действия `ProductDetail`, как показано ниже (выделено полужирным шрифтом):

```
public async Task<IActionResult> ProductDetail(int? id)
{
  if (!id.HasValue)
  {
    return NotFound("You must pass a product ID in the route, for example,
/Home/ProductDetail/21");
  }

  var model = await db.Products
    .SingleOrDefaultAsync(p => p.ProductID == id);

  if (model == null)
  {
    return NotFound($"Product with ID of {id} not found.");
  }
  return View(model); // передача модели представлению,
                      // а затем возврат результата
}
```

4. Запустите сайт, используйте Google Chrome для перехода на него и обратите внимание, что функциональные возможности сайта такие же. Однако поверьте — масштабироваться он теперь будет лучше.
5. Закройте браузер и остановите работу сайта.

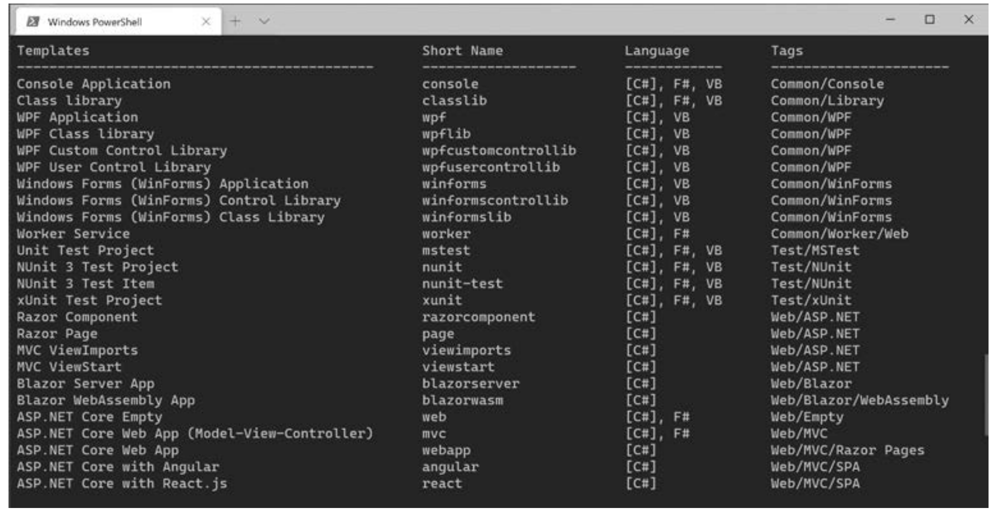
Использование других шаблонов проектов

При установке пакета SDK в .NET Core включается множество шаблонов проектов.

1. На панели TERMINAL (Терминал) введите следующую команду:

```
dotnet new --help
```

2. Вы увидите список установленных шаблонов (рис. 16.12).
3. Обратите внимание на шаблоны веб-проектов, в том числе шаблоны для создания SPA с помощью Blazor.



Templates	Short Name	Language	Tags
Console Application	console	[C#], F#, VB	Common/Console
Class library	classlib	[C#], F#, VB	Common/Library
WPF Application	wpf	[C#], VB	Common/WPF
WPF Class library	wpflib	[C#], VB	Common/WPF
WPF Custom Control Library	wpfcustomcontrollib	[C#], VB	Common/WPF
WPF User Control Library	wpfusercontrollib	[C#], VB	Common/WPF
Windows Forms (WinForms) Application	winforms	[C#], VB	Common/WinForms
Windows Forms (WinForms) Control Library	winformscontrollib	[C#], VB	Common/WinForms
Windows Forms (WinForms) Class Library	winformslib	[C#], VB	Common/WinForms
Worker Service	worker	[C#], F#	Common/Worker/Web
Unit Test Project	mstest	[C#], F#, VB	Test/MSTest
NUnit 3 Test Project	nunit	[C#], F#, VB	Test/NUnit
NUnit 3 Test Item	nunit-test	[C#], F#, VB	Test/NUnit
xUnit Test Project	xunit	[C#], F#, VB	Test/xUnit
Razor Component	razorcomponent	[C#]	Web/ASP.NET
Razor Page	page	[C#]	Web/ASP.NET
MVC ViewImports	viewimports	[C#]	Web/ASP.NET
MVC ViewStart	viewstart	[C#]	Web/ASP.NET
Blazor Server App	blazorserver	[C#]	Web/Blazor
Blazor WebAssembly App	blazorwasm	[C#]	Web/Blazor/WebAssembly
ASP.NET Core Empty	web	[C#], F#	Web/Empty
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#	Web/MVC
ASP.NET Core Web App	webapp	[C#]	Web/MVC/Razor Pages
ASP.NET Core with Angular	angular	[C#]	Web/MVC/SPA
ASP.NET Core with React.js	react	[C#]	Web/MVC/SPA

Рис. 16.12. Список шаблонов проектов

Установка дополнительных шаблонов

Разработчики могут установить множество дополнительных пакетов шаблонов.

1. Запустите браузер и перейдите по адресу dotnetnew.azurewebsites.net.
2. Введите vue в текстовое поле, нажмите кнопку Search templates (Искать шаблоны) и обратите внимание на список доступных шаблонов для `Vue.js`, в том числе опубликованных Microsoft (рис. 16.13).

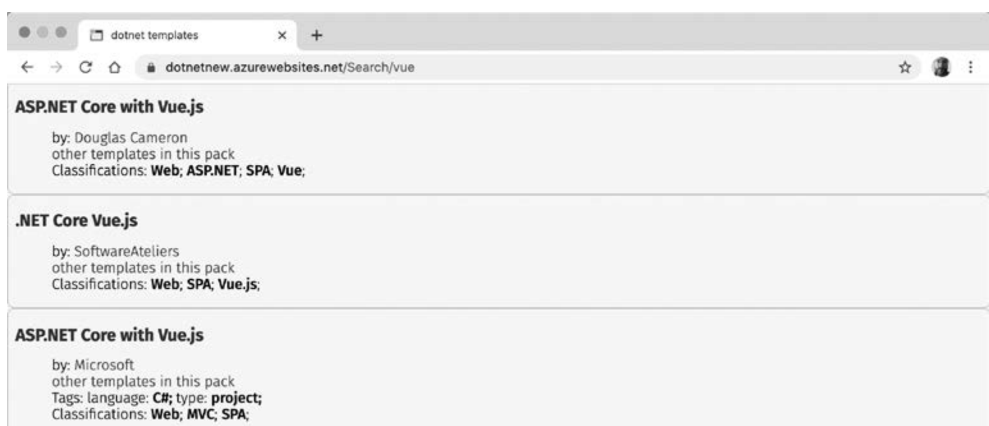


Рис. 16.13. Список шаблонов Vue.js

3. Нажмите ссылку ASP.NET Core with Vue.js и обратите внимание на инструкции по установке и использованию этого шаблона.

Более подробную информацию о шаблонах можно получить на сайте https://github.com/dotnet/templating/wiki/Available-templates-for-dotnet-new.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 16.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Зачем нужны файлы со специальными именами `_ViewStart` и `_ViewImports`, созданные в папке Views?
2. Что представляют собой имена трех сегментов, определенных по умолчанию в ASP. NET Core MVC, и какие из них необязательны?
3. Что такое привязка моделей и какие типы данных она может обрабатывать?
4. Как выводится содержимое текущего представления в общем файле макета, таком как `_Layout.cshtml`?
5. Как в общем файле макета, таком как `_Layout.cshtml`, задать секцию, для которой содержимое может быть предоставлено текущим представлением, и как это представление выдает содержимое для этой секции?
6. Какие пути ищутся для представления по соглашению при вызове метода `View` внутри метода действия контроллера?
7. Как вы можете указать браузеру пользователя кэшировать ответ в течение 24 часов?
8. Почему вы можете решить подключить Razor Pages, даже если сами их не создаете?
9. Как ASP.NET Core MVC идентифицирует классы, которые могут действовать как контроллеры?
10. Каким образом ASP.NET Core MVC облегчает тестирование сайта?

Упражнение 16.2. Реализация MVC для страницы, содержащей сведения о категориях

В проекте NorthwindMvc есть главная страница, на которой отображаются категории, но при нажатии кнопки View (Вид) сайт возвращает ошибку 404 Not Found, например, для следующего URL: https://localhost:5001/category/1.

Дополните проект NorthwindMvc, добавив возможность отображения страницы, содержащей подробные сведения о категории.

Упражнение 16.3. Улучшение масштабируемости за счет понимания и реализации асинхронных методов действий

Несколько лет назад Стивен Клири написал отличную статью для MSDN Magazine, объясняющую преимущества в масштабируемости от реализации асинхронных методов действия для ASP.NET. Для ASP.NET Core те же принципы не только верны, но и имеют еще большее влияние, поскольку в отличие от устаревшего ASP.NET, как описано в статье, ASP.NET Core поддерживает асинхронные фильтры и другие компоненты.

Прочитайте статью по следующей ссылке и измените приложение, созданное в этой главе, применив асинхронные методы действия в контроллере: https://docs.microsoft.com/ru-ru/archive/msdn-magazine/2014/october/async-programming-introduction-to-async-await-on-asp-net.

Упражнение 16.4. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- обзор ASP.NET Core MVC: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/overview;
- учебное пособие: начало работы с EF Core в веб-приложении ASP.NET MVC: https://docs.microsoft.com/ru-ru/aspnet/core/data/ef-mvc/intro;
- обработка запросов с помощью контроллеров в ASP.NET Core MVC: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/controllers/actions;
- привязка моделей в ASP.NET Core: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/models/model-binding;
- представления в ASP.NET Core MVC: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/overview.

Резюме

Вы научились создавать большие и сложные сайты таким образом, чтобы можно было легко их тестировать и управлять ими с помощью команд разработчиков, использующих ASP.NET Core. Вы изучили конфигурацию запуска, аутентификацию, маршруты, модели, представления и контроллеры в ASP.NET Core MVC.

Далее вы узнаете, как создавать сайты с помощью кросс-платформенной *системы управления контентом* (Content Management System, CMS). Это позволяет разработчикам передать ответственность за контент тем, кому он принадлежит, — пользователям.

17 Разработка сайтов с помощью системы управления контентом (CMS)

Данная глава посвящена разработке сайтов с помощью современной кросс-платформенной *системы управления контентом* (Content Management System, CMS).

Существует множество вариантов CMS для большинства платформ веб-разработки. Для изучения основных принципов работы с CMS в контексте кросс-платформенной разработки на C# и .NET на сегодняшний момент лучше всего выбрать Piranha CMS. С выпуском 1 декабря 2017 года своей версии Piranha CMS 4.0 она стала первой CMS, поддерживающей .NET Core.

В этой главе:

- преимущества CMS;
- Piranha CMS;
- определение компонентов, типов контента и шаблонов;
- тестирование сайта CMS.

Преимущества CMS

В предыдущих главах вы научились создавать статические веб-страницы в формате HTML и настраивать ASP.NET Core для выдачи их по запросу браузера пользователя.

Вы также научились с помощью ASP.NET Core Razor Pages добавлять код на языке C#, который выполняется на стороне сервера, для динамической генерации HTML, в том числе из информации, загружаемой в реальном времени из базы данных. Кроме того, вы узнали, как ASP.NET Core MVC обеспечивает разделение технических задач, чтобы сделать создание более сложных сайтов более управляемым.

Платформа ASP.NET Core сама по себе не решает проблему управления контентом. Специалист, создающий контент и управляющий им, должен обладать навыками программирования и редактирования HTML или возможностью редактирования данных в базе данных `Northwind`, чтобы изменить информацию, доступную пользователям на сайте.

Тут-то и становится полезной CMS. Система отделяет контент (значения данных) от шаблонов (макет, формат и стиль). Большинство CMS генерируют веб-ответы, такие как HTML, для пользователей, просматривающих сайт с помощью браузера.

Некоторые CMS генерируют открытые форматы данных, такие как JSON и XML, для обработки веб-сервисом или визуализации в браузере с помощью технологий на стороне клиента, наподобие Angular, React или Vue. Подобную систему часто называют *headless CMS* (система управления контентом, которая функционирует без собственного пользовательского интерфейса).

Разработчики определяют структуру данных, хранящихся в CMS, используя классы типов контента для различных целей (например, страница для отображения товара) с шаблонами контента, которые визуализируют данные контента в HTML, JSON или других форматах.

Владельцы контента, не обладающие специфическими техническими знаниями, могут входить в CMS и с помощью простого пользовательского интерфейса создавать, редактировать, удалять и публиковать контент, который будет соответствовать структуре, определяемой классами типов контента. Для этого им не требуются ни разработчики, ни такие инструменты, как Visual Studio Code.

Основные функции CMS

Любая базовая CMS будет включать в себя следующие основные функции:

- пользовательский интерфейс, позволяющий владельцам контента входить и управлять своим контентом;
- совместное и повторное использование фрагментов контента, обычно называемых *блоками*;
- сохраненные черновики контента, скрытые от пользователей сайта (пока они не будут опубликованы);
- *оптимизированные для поисковых систем* (Search Engine Optimized, SEO) URL, заголовки страниц и соответствующие метаданные, карты сайтов и т. д.;
- аутентификацию и авторизацию, включая управление пользователями, группами и их правами доступа к контенту;

- систему доставки контента, которая преобразует контент из простых данных в один или несколько форматов, таких как HTML и JSON.

Возможности корпоративной CMS

Любая коммерческая CMS корпоративного уровня добавит следующие дополнительные функции:

- дизайнер форм для получения данных от пользователей;
- маркетинговые инструменты, такие как отслеживание поведения пользователей и A/B-тестирование контента;
- персонализацию контента на основе, например, географического положения или обработки с помощью машинного обучения данных о поведении пользователей;
- сохраненный в черновики контент, который скрыт до публикации от пользователей сайта;
- сохранение нескольких версий контента и возможность повторной публикации устаревших версий;
- перевод контента на несколько языков, например английский и немецкий.

Платформы CMS

Для большинства платформ и языков разработки существуют свои CMS, как показано в табл. 17.1.

Таблица 17.1

Платформа разработки	Система управления контентом
PHP	WordPress, Drupal, Joomla!, Magento
Python	django CMS
Java	Adobe Experience Manager, Bloomreach Experience Manager (ранее Hippo CMS)
.NET Framework	Episerver CMS, Sitecore, Umbraco, Kentico CMS
.NET Core and .NET 5	Piranha CMS, Orchard Core CMS

Orchard Core CMS версии 1.0 планировалось выпустить в сентябре 2020 года, но на момент публикации она все еще была последней выпущенной версией, поэтому для данного издания я решил использовать Piranha CMS, поскольку эта среда поддерживает .NET Core, начиная с версии 4.0 и до текущей версии 8.4. О системе Orchard Core CMS можно прочитать на сайте https://orchardcore.readthedocs.io/en/dev/.

Piranha CMS

Piranha CMS — отличный выбор для изучения разработки под CMS, поскольку это система с открытым исходным кодом, простая и гибкая.

По словам ее ведущего разработчика и создателя Хакана Эдлинга, «Piranha CMS — это легкая, ненавязчивая и кросс-платформенная библиотека CMS для .NET Standard 2.0. Она позволяет добавлять функциональность CMS в уже существующее приложение, создавать новый сайт с нуля или даже использовать ее в качестве бэкенда для мобильного приложения».

Piranha CMS имеет три принципа проектирования:

- открытая и расширяемая платформа;
- простая и интуитивно понятная для администраторов контента;
- быстрая, эффективная и нескучная для разработчиков.

Вместо того чтобы добавлять все больше и больше сложных функций для больших коммерческих клиентов, Piranha CMS фокусируется на предоставлении платформы для небольших и средних компаний, которым необходимо, чтобы нетехнические пользователи могли редактировать контент.

Piranha не WordPress, а это значит, что у нее никогда не будет тысяч предустановленных тем и плагинов. По словам создателей Piranha, «сердце и душа Piranha — структурирование контента и его редактирование самым интуитивным способом; остальное мы оставляем на ваше усмотрение».

Официальную документацию по Piranha CMS можно прочитать на сайте https://piranhacms.org/.

Библиотеки с открытым исходным кодом и лицензирование

Система Piranha CMS построена с помощью некоторых библиотек с открытым исходным кодом, включая следующие:

- *Font Awesome* — самый популярный набор веб-значков и инструментария: https://fontawesome.com;
- *AutoMapper* — основанное на конвенциях средство отображения объектов: https://automapper.org;
- *Markdig* — быстрый, мощный, совместимый с CommonMark, расширяемый процессор Markdown для .NET: https://github.com/lunet-io/markdig;
- *Newtonsoft Json.NET* — популярный высокопроизводительный фреймворк для работы с JSON в .NET: https://www.newtonsoft.com/json.

Piranha CMS выпускается под лицензией MIT, что означает следующее: система разрешает повторное использование в проприетарном программном обеспечении при условии, что все копии лицензионного программного обеспечения включают копию условий лицензии MIT и уведомление об авторских правах.

Создание веб-приложения с помощью Piranha CMS

Система Piranha CMS содержит четыре шаблона проекта: *Empty, MVC, Razor Pages и Module.*. MVC и Razor Pages включают модели, контроллеры и представления для базовых страниц, а также архив блога с сообщениями блога. Но уже вам решать, предпочитаете ли вы MVC или Razor Pages для реализации своего сайта.

Нам необходимо будет установить данные шаблоны, прежде чем мы сможем создать проект веб-приложения Piranha CMS.

Вы будете применять шаблон проекта `piranha.blog`, чтобы создать сайт Piranha CMS с базой данных SQLite для хранения контента, включая записи в блогах, а также имена пользователей и пароли для аутентификации.

1. В папке PracticalApps создайте подпапку NorthwindCms.
2. В программе Visual Studio Code откройте рабочее пространство `PracticalApps`.
3. Добавьте в рабочую область папку NorthwindCms.
4. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и выберите папку NorthwindCms.
5. На панели TERMINAL (Терминал) установите шаблоны проектов Piranha:

```
dotnet new -i Piranha.Templates
```

6. На панели TERMINAL (Терминал) введите следующую команду для вывода списка шаблонов Piranha CMS:

```
dotnet new "piranha cms" --list
```

7. Обратите внимание на шаблоны:

```
Templates                                     Short Name  Language
--------------------------------------------------------------------
ASP.NET Core Empty with Piranha CMS           piranha.empty   [C#]
ASP.NET Core MVC Web with Piranha CMS         piranha.mvc     [C#]
ASP.NET Core Razor Pages Web with Piranha CMS piranha.razor   [C#]
ASP.NET Core Razor pages Piranha CMS Module   piranha.module  [C#]
```

8. На панели TERMINAL (Терминал) введите команды для создания сайта Piranha CMS, используя MVC:

```
dotnet new piranha.mvc
```

9. На панели EXPLORER (Проводник) найдите и откройте файл `NorthwindCms.csproj` и обратите внимание, что на момент написания книги Piranha CMS имеет версию 8.4 и ориентирована на ASP.NET Core 3.1, версию с долгосрочной поддержкой (до 3 декабря 2022 года):

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Piranha" Version="8.4.2" />
    <PackageReference Include="Piranha.AspNetCore" Version="8.4.1" />
    <PackageReference Include="Piranha.AspNetCore.Identity.SQLite"
      Version="8.4.0" />
    <PackageReference Include="Piranha.AttributeBuilder"
      Version="8.4.0" />
    <PackageReference
      Include="Piranha.Data.EF.SQLite" Version="8.4.0" />
    <PackageReference Include="Piranha.ImageSharp" Version="8.4.0" />
    <PackageReference Include="Piranha.Local.FileStorage"
      Version="8.4.0" />
    <PackageReference Include="Piranha.Manager" Version="8.4.0" />
    <PackageReference Include="Piranha.Manager.TinyMCE"
      Version="8.4.0" />
  </ItemGroup>
</Project>
```

Изучение сайта Piranha CMS

1. На панели TERMINAL (Терминал) создайте и запустите сайт, используя следующую команду:

```
dotnet run
```

2. Запустите браузер Google Chrome. В адресной строке введите следующий URL: https://localhost:5001/.
3. Обратите внимание на главную страницу по умолчанию для сайта, созданного с помощью Piranha CMS, на которой приветствие, означающее, что «вы можете войти на панель администратора по адресу ~/manager с учетными данными по умолчанию admin/password. При необходимости для вас мы можем ввести некоторые данные. Пример демонстрирует создание и структурирование с помощью Piranha CMS. После того как вы закончите настройку, вы можете удалить `SetupController` и папку ~/seed из вашего проекта».
4. Внизу страницы нажмите ссылку Seed some data и обратите внимание на следующие моменты.

- На главной странице сайта находятся блоки контента, что позволяет пользователю более детально изучить сайт с помощью ссылок, которые могут отображать часть содержимого страницы, например основное изображение и описание.
- Страница по умолчанию для сайта — это архив блога, в котором отображаются последние записи. Каждая запись в блоге содержит изображение, заголовок и дату публикации, сводный текст с кнопкой Read more (Читать далее) и может иметь категорию и хештег (рис. 17.1).

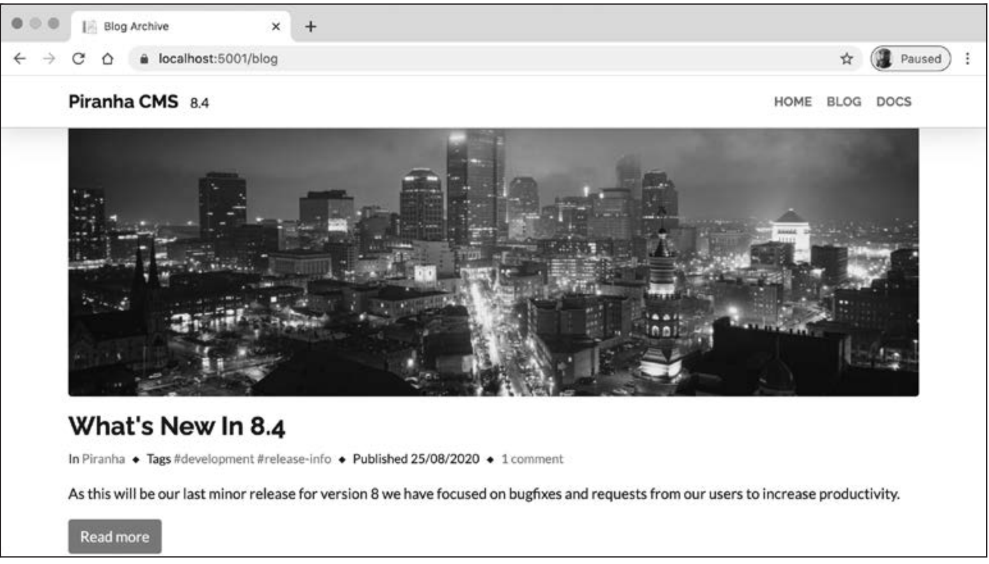


Рис. 17.1. Сайт Piranha CMS

Редактирование содержимого сайта и страницы

Войдем в систему от имени владельца контента для сайта и будем управлять неким контентом.

1. В адресной строке браузера введите следующий URL: https://localhost:5001/manager.
2. Введите имя пользователя `admin` и пароль `password`.
3. В менеджере обратите внимание на существующие страницы с именами Home (Главная) (пример типа «стандартная страница»), Blog (Блог) (пример типа «архив блога») и Docs (Документы) (пример типа «стандартная страница»), а затем нажмите ссылку Default Site (Сайт по умолчанию) (рис. 17.2).

Рис. 17.2. Default Site и его страницы

4. В окне Edit site (Редактировать сайт) на вкладке Settings (Параметры) измените Title (Заголовок) на `Northwind CMS` и Description (Описание) на `Providing fresh tasty food to restaurants for three generations` и нажмите кнопку Save (Сохранить) (рис. 17.3).

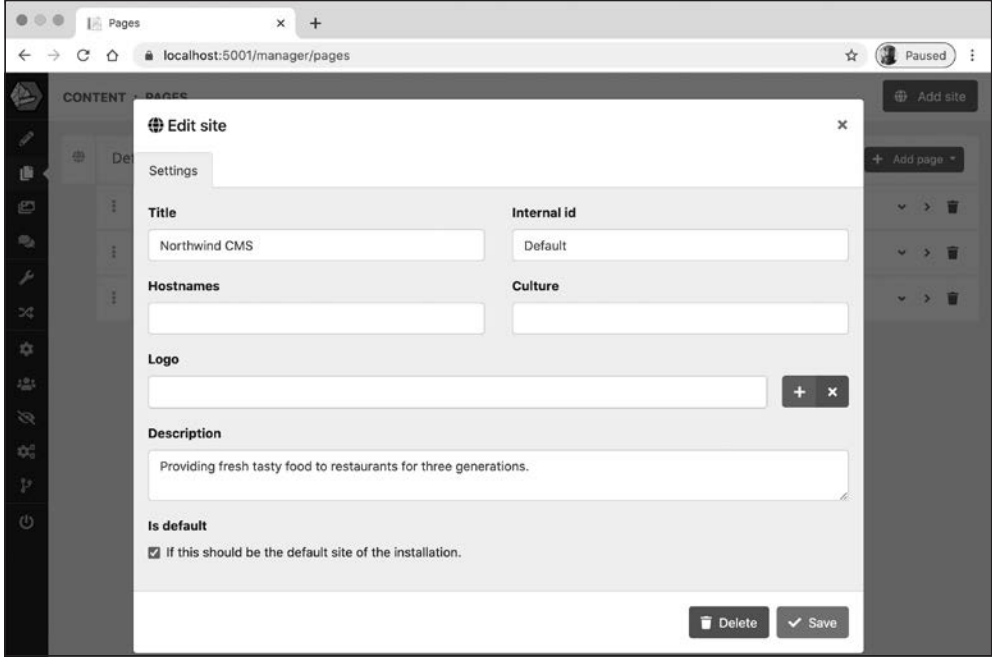


Рис. 17.3. Редактирование настроек сайта

5. Если вы еще не находитесь в разделе CONTENT : PAGES (Содержание: Страницы), то для редактирования страницы на панели навигации в меню слева нажмите ссылку Pages (Страницы), а затем Home (Главная).

6. Измените заголовок страницы на Welcome To Northwind CMS и обратите внимание, что под ним владелец контента может установить основное изображение и некоторый текст, а затем добавить на страницу любое количество блоков, как показано на рис. 17.4, в том числе:
   - горизонтальные разделители между каждым блоком с круглой кнопкой для вставки нового блока;
   - появляющиеся при фокусировке на блок кнопки, позволяющие свернуть, развернуть и удалить блок, а также меню дополнительных действий;
   - блок CONTENT с одним столбцом форматированного текста, изображений и ссылок, который легко оформляется с помощью панели инструментов;
   - блок COLUMNS с несколькими столбцами форматированного текста, с кнопкой + для добавления столбцов и кнопками значка корзины для удаления столбцов или всего блока;
   - блок GALLERY, который может содержать несколько изображений и вращать их в анимации в стиле карусель.

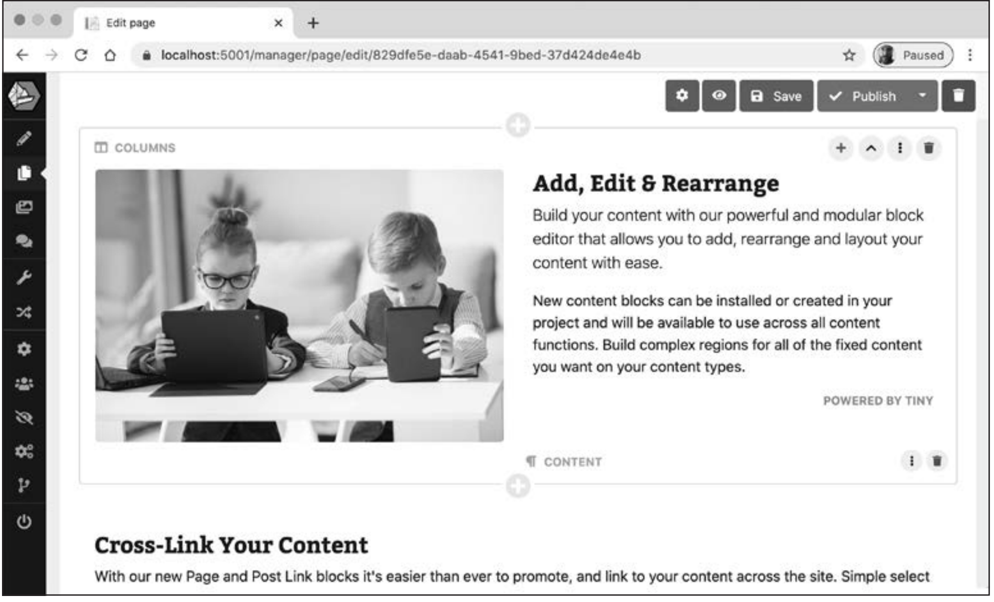


Рис. 17.4. Редактирование страницы

7. Вверху страницы CONTENT : PAGES/EDIT (Содержание: страницы/редактирование) нажмите значок шестеренки, чтобы открыть диалоговое окно Settings

(Настройки), где владелец контента может установить заголовок навигации, метаключевые слова, метаописание и строку, используемую в пути URL, дату публикации, заголовок навигации, если страница должна быть скрыта в карте сайта, перенаправление и включение комментариев в течение определенного количества дней (рис. 17.5).

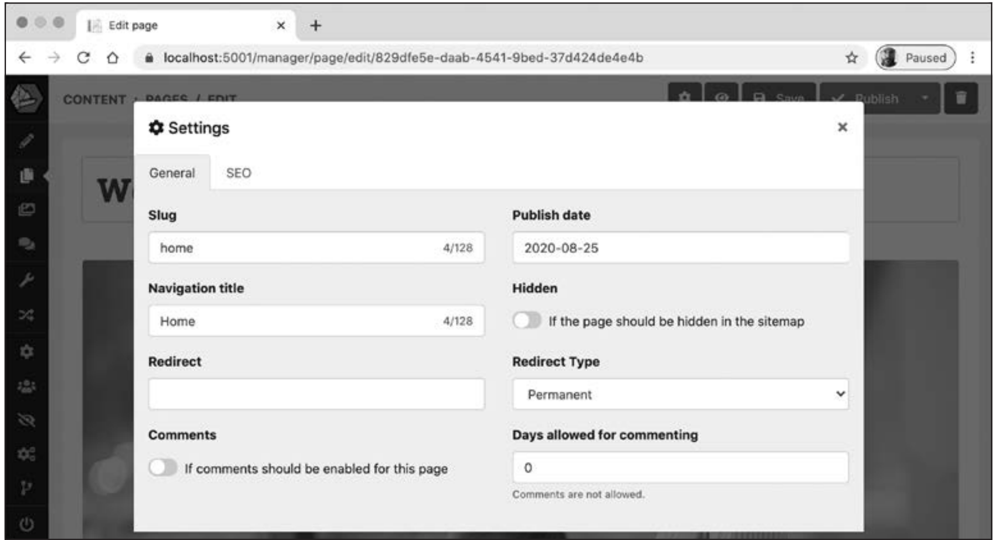


Рис. 17.5. Диалоговое окно General Settings (Настройки)

8. Щелкните на вкладке SEO, где владелец контента может установить метазаголовок, ключевые метаслова, метаописание и метаданные OpenGraph (OG).

Более подробно об установке метаданных OpenGraph для своих веб-страниц вы можете прочитать на сайте https://ogp.me/.

9. Измените Meta title (Метазаголовок) на `CMS`.
10. Измените Meta keywords (Ключевые метаслова) на `beverages`, `condiments`, `meat`, `seafood`.
11. Измените Meta description (Метаописание) на `Providing fresh tasty food to restaurants for three generations`.
12. Обратите внимание, что метаданные заголовка и описания OpenGraph установлены и вам нужно выбрать изображение самостоятельно, как показано на рис. 17.6.

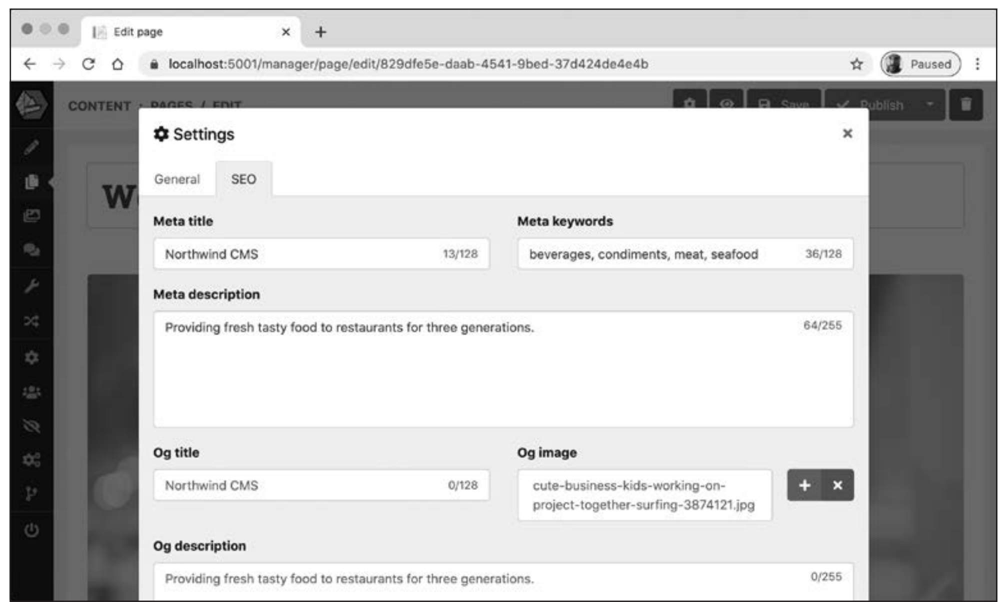


Рис. 17.6. Диалоговое окно SEO Settings (Настройки SEO)

13. Закройте диалоговое окно.
14. Вверху страницы CONTENT : PAGES/EDIT (Содержание: страницы/редактирование) нажмите кнопку Save (Сохранить) и обратите внимание, что изменения были сохранены в черновике (рис. 17.7).

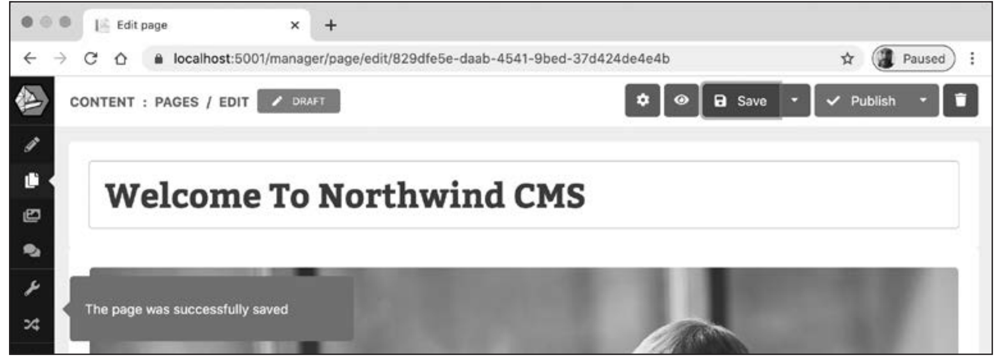


Рис. 17.7. Сохранение страницы

15. Чтобы сделать изменения видимыми для пользователей сайта, нажмите кнопку Publish (Опубликовать).

Создание новой страницы верхнего уровня

Создадим новую страницу с несколькими блоками в качестве содержимого.

1. В меню навигации слева нажмите ссылку Pages (Страницы), кнопку + Add (+ Добавить) и затем выберите пункт Standard page (Стандартная страница).
2. В поле Your page title (Название вашей страницы) введите `Contact Us`.
3. Выберите разнообразные изображения.
4. Для текста страницы введите We welcome your contact (Мы приветствует вас!).
5. Нажмите круглую кнопку + и выберите блок Columns (Столбцы) (рис. 17.8).

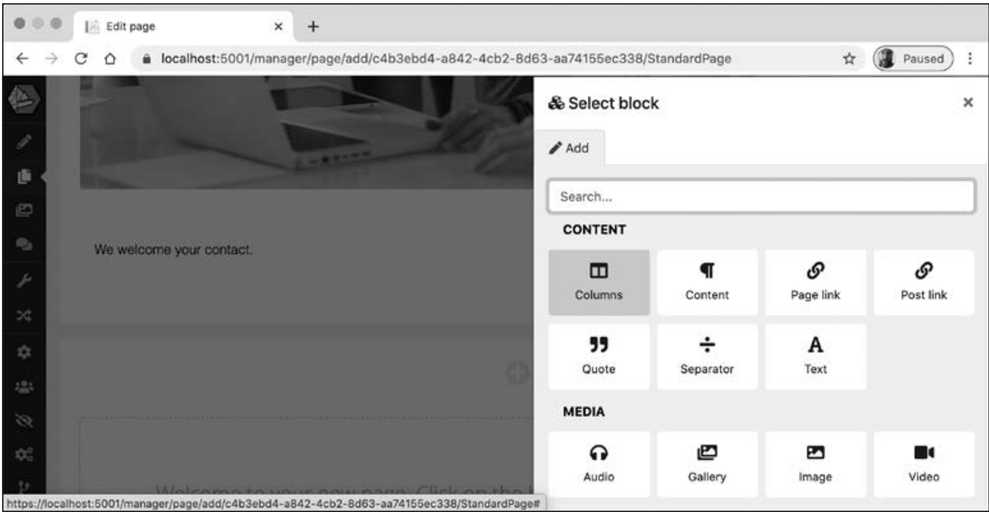


Рис. 17.8. Добавление блока контента на страницу

6. Вверху блока Columns (Столбцы) нажмите кнопку +, ссылку Content (Содержание) и введите вымышленный почтовый адрес, например 123 Main Street, New York City, United States, а также поддельный адрес электронной почты и номер телефона, например admin@northwind.com и (123) 555-1234.
7. Вверху блока Columns (Столбцы) нажмите кнопку +, ссылку Quote (Цитата) и введите We love our customers by the Customer Success Team (рис. 17.9).
8. Вверху страницы CONTENT : PAGES/EDIT (Содержание: страницы/редактирование) нажмите кнопку Publish (Опубликовать).

Если вы просто нажмете кнопку Save (Сохранить), то новая страница будет сохранена в базе данных CMS, но пока не будет видна пользователям сайта.

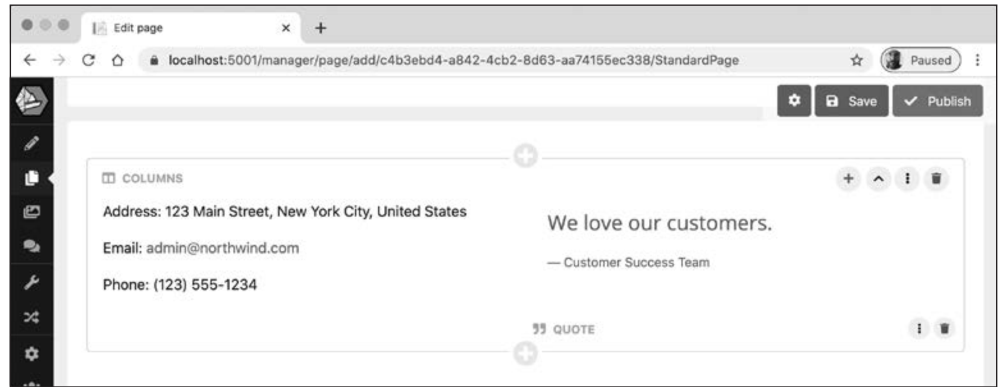


Рис. 17.9. Добавление блока цитаты

Создание новой дочерней страницы

При создании новых страниц в Piranha CMS необходимо учитывать иерархию страниц.

Для создания страницы Contact Us (Свяжитесь с нами) мы нажали кнопку + Add page (+ Добавить страницу) вверху списка Pages (Страницы). Но чтобы вставить новую страницу в существующую структуру, необходимо нажать значки со стрелками вниз или вправо.

- *Стрелка вниз* создает новую страницу *после* текущей, то есть на том же уровне.
- *Стрелка вправо* создает новую страницу *под* текущей, то есть дочернюю.

Если вы создадите страницу не там, то ее легко будет перетащить в правильное место в древовидной структуре страницы. Многие владельцы контента, вероятно, так и будут делать: сначала создадут новые страницы внизу списка, а затем перетащат их в нужное место в дереве страниц.

Создадим новую дочернюю страницу для страницы Contact Us (Свяжитесь с нами).

1. Слева в меню на панели навигации нажмите ссылку Pages (Страницы) и в строке Contact Us (Свяжитесь с нами) нажмите значок со стрелкой вправо и выберите пункт Standard page (Стандартная страница). Обратите внимание, что вы можете также скопировать содержимое существующей страницы, но мы начнем с создания пустой стандартной страницы.
2. Введите `Our Location` в качестве заголовка страницы и нажмите кнопку Publish (Опубликовать).
3. Слева в меню на панели навигации выберите ссылку Pages (Страницы) и обратите внимание, что страница Our Location (Наше местоположение) — дочерняя по отношению к странице Contact Us (Свяжитесь с нами) (рис. 17.10).

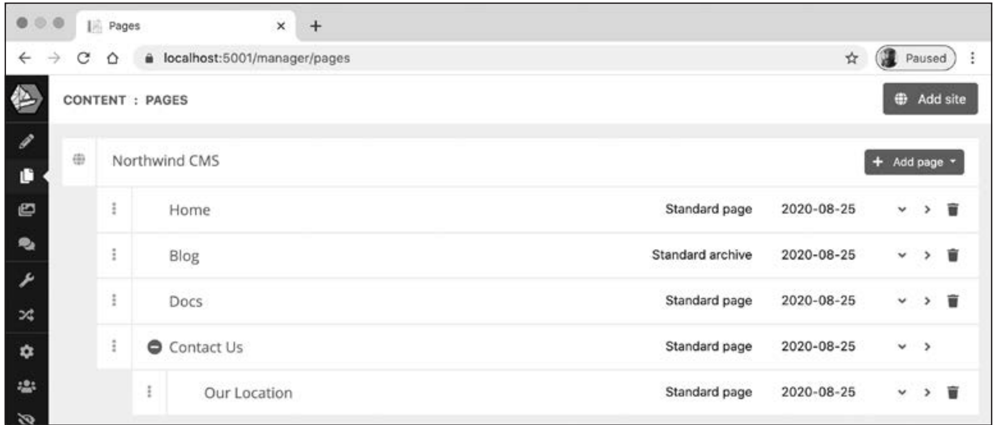


Рис. 17.10. Структура страниц сайта Northwind CMS

4. Щелкните на странице Our Location (Наше местоположение), чтобы отредактировать ее.
5. Щелкните на значке шестеренки, чтобы открыть диалоговое окно Settings (Параметры), и обратите внимание, что параметр Slug (Слаг) был установлен автоматически в зависимости от того, где страница была создана изначально (рис. 17.11).

Рис. 17.11. Параметр Slug (Слаг) страницы Our Location

Слаг для страницы не изменится автоматически, если страница перетаскивается и перемещается в другое место в иерархии дерева страниц, и придется менять его вручную.

Обзор архива блога

Рассмотрим архив блога.

1. Слева в меню на панели навигации нажмите ссылки Pages (Страницы), затем ссылку Blog (Блог). Обратите внимание: в типе «архив блога» на странице

содержится вкладка для основного содержимого с изображением, текстом и блоками, как и на Standard page (Стандартная страница).

2. Щелкните на вкладке Archive (Архив). Обратите внимание, что в ней таблица элементов блога, каждая строка которой включает название элемента, дату публикации, тип элемента (в данном случае есть один элемент — запись в блоге), категорию (в данном случае Piranha, Tristique и Magna) и возможность удалять элементы (рис. 17.12).

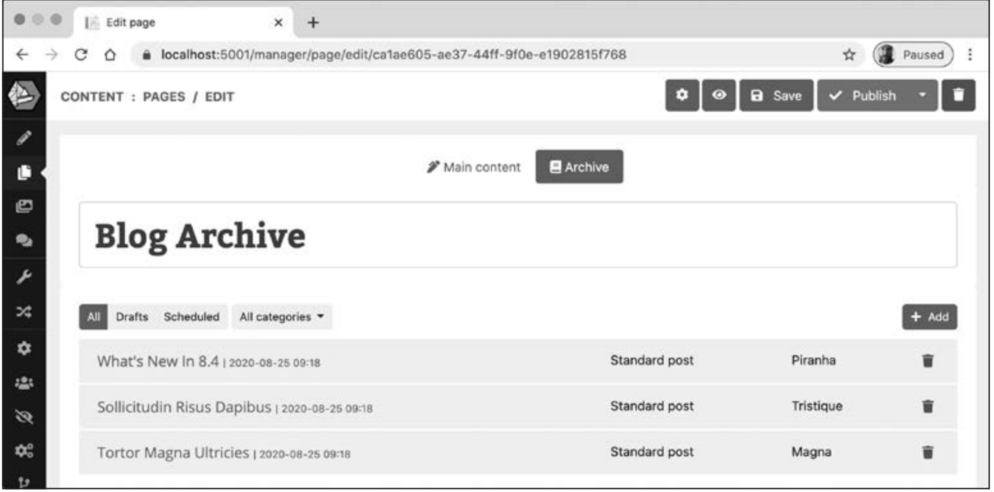


Рис. 17.12. Blog Archive (Архив блога) со списком опубликованных сообщений

3. Нажмите кнопку + Add (+ Добавить) и затем ссылку Blog post (Запись в блоге).
4. Введите сообщение, например, `Northwind has some cool new fish to sell you!`, введите `Fish` для категории, добавьте несколько тегов, таких как `seafood` и `cool`, а затем добавьте хотя бы один блок, например цитату: `"This fish is the tastiest ever!"`.
5. Нажмите кнопку Publish (Опубликовать).
6. Нажмите стрелку справа от кнопки Save (Сохранить) и затем кнопку Preview (Предварительный просмотр).
7. На вкладке браузера Live Preview (Динамический просмотр) нажмите кнопку MOBILE (Мобильные устройства), чтобы увидеть вид записи в блоге на мобильных устройствах (рис. 17.13).

Комментирование постов и страниц

Одна из последних новых функций Piranha CMS — встроенная поддержка комментариев к сообщениям и страницам.

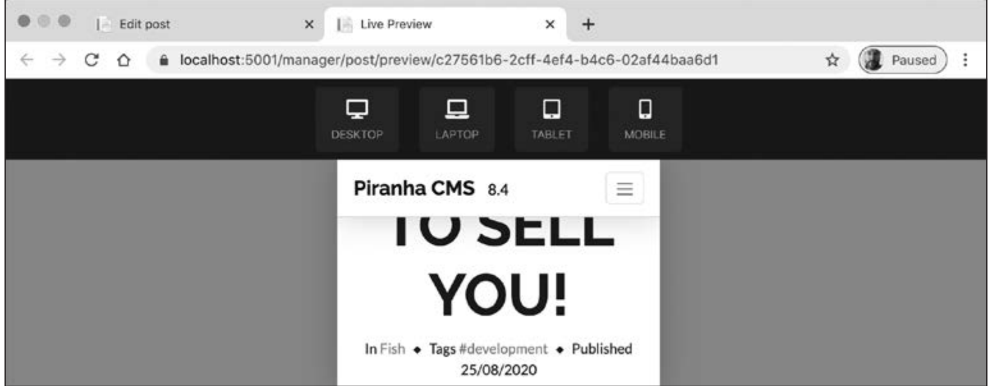


Рис. 17.13. Вкладка Live Preview (Динамический просмотр)

1. Выберите TABLET (Планшет) и прокрутите сообщение вниз. Обратите внимание, что пользователь может оставить комментарий (рис. 17.14).

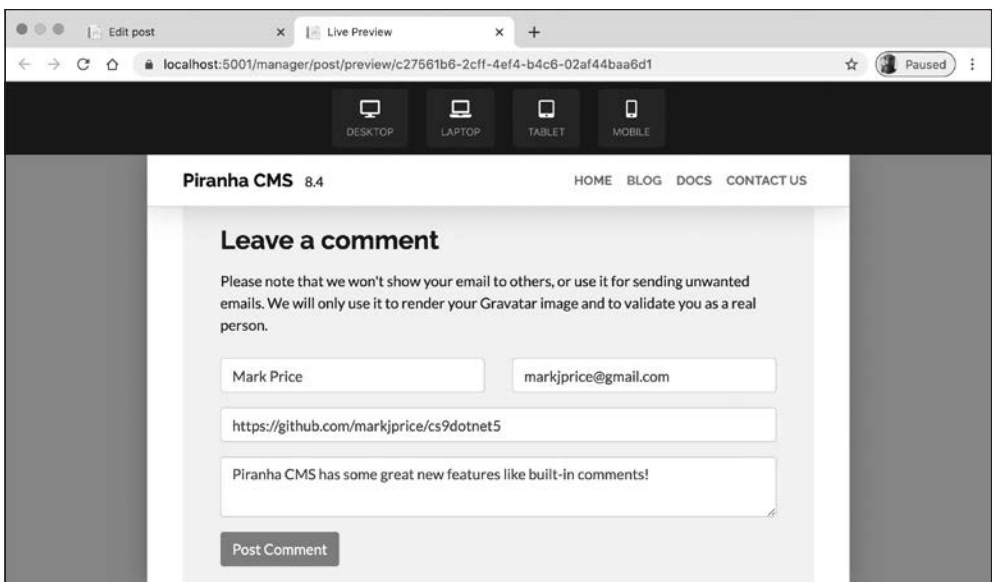


Рис. 17.14. Предварительный просмотр страницы TABLET (Планшет)

2. Оставьте свой комментарий.
3. Закройте вкладку браузера Live Preview (Динамический просмотр).
4. На панели навигации меню слева выберите Comments (Комментарии). Обратите внимание на таблицу комментариев для всех страниц (рис. 17.15).

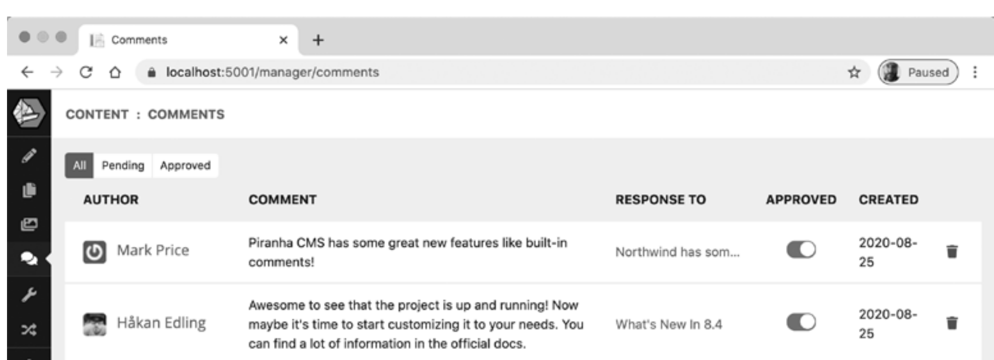


Рис. 17.15. Страница комментариев

5. Чтобы отменить свой комментарий, нажмите кнопку-переключатель в столбце APPROVED (Утверждено).
6. Чтобы быстро вернуться к сообщению, выберите заголовок сообщения в столбце RESPONSE TO (Ответ на).
7. Обратите внимание на предупреждение, отображающее количество комментариев, находящихся на рассмотрении (рис. 17.16).

Рис. 17.16. Один комментарий к уведомлению

8. Выберите вкладку Comments (Комментарии).
9. Выберите вкладку Pending (На рассмотрении).
10. Щелкните на переключателе, чтобы одобрить свой комментарий.
11. Закройте вкладку Comments (Комментарии).

Аутентификация и авторизация

Рассмотрим настройки системы, которые доступны для защиты контента.

1. Слева в меню на панели навигации нажмите ссылку Users (Пользователи) и обратите внимание, что admin (пользователь с правами администратора) — член роли `SysAdmin` (рис. 17.17).

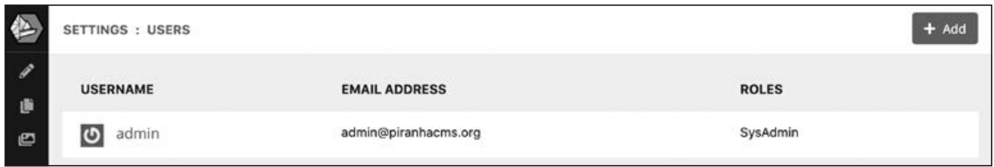


Рис. 17.17. Роли администратора

2. Слева в меню на панели навигации выберите ссылку Roles (Роли), выберите роль `SysAdmin` и обратите внимание, что вы можете назначить десятки разрешений для роли (рис. 17.18).

Рис.17.18. Управление разрешениями

3. Слева в меню на панели навигации нажмите ссылку Roles (Роли), затем нажмите кнопку + Add (+ Добавить).
4. В разделе GENERAL (Общие) введите имя `Editors`.
5. В разделе CORE PERMISSIONS (Основные разрешения) выберите оба разрешения.
6. В разделе MANAGER PERMISSIONS (Разрешения менеджера) выберите следующие разрешения:
   - Comments (Комментарии): List Comments (Список комментариев);
   - Media (Мультимедиа) — Add Media (Добавить мультимедиа), Add Media Folders (Добавить папку мультимедиа), Edit Media (Редактировать мультимедиа), List Media (Список мультимедиа);
   - Pages (Страницы) — Add Pages (Добавить страницы), Edit Pages (Редактировать страницы), List Pages (Список страниц), Pages — Save (Страницы — сохранить);
   - Posts (Сообщения) — Add Posts (Добавить сообщения), Edit Posts (Редактировать сообщения), List Posts (Список сообщений), Save Posts (Сохранить сообщения).
7. Нажмите кнопку Save (Сохранить).

8. Слева в меню на панели навигации нажмите ссылку Users (Пользователи), затем кнопку + Add (+ Добавить).
9. Введите имя `Eve`, адрес электронной почты `eve@northwind.com`, назначьте пользователю роль `Editors` и установите пароль `Pa$$w0rd` (рис. 17.19).

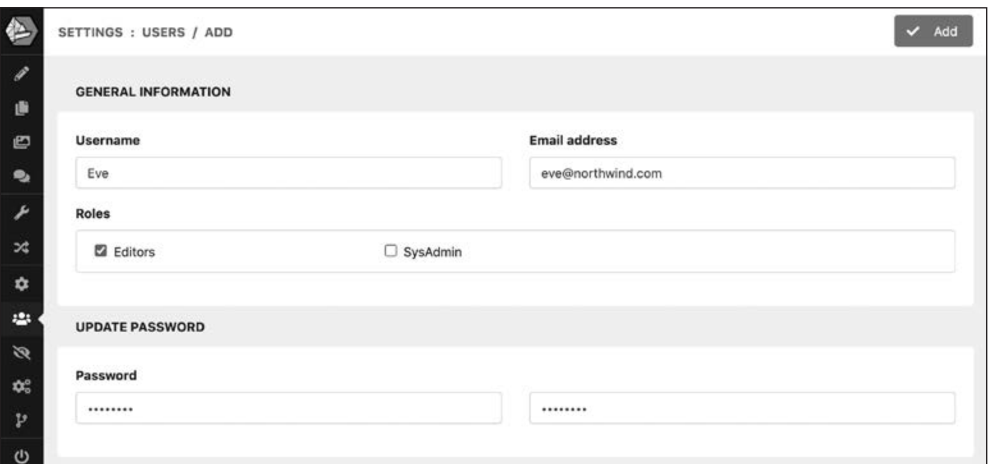


Рис. 17.19. Добавление пользователя

10. Нажмите кнопку Add Добавить).

Вам необходимо продумать, какие разрешения нужны для разных ролей. Роль SysAdmin (Администратор) должна иметь все разрешения. Если вы создадите роль Editor (Редактор), то можете дать ей разрешение добавлять, удалять, редактировать и сохранять страницы, публикации и мультимедиа. Но, возможно, только роли Publisher (Издатель) будет разрешено публиковать контент и контролировать, когда разрешать пользователям сайта увидеть содержимое.

Изучение конфигурации

Рассмотрим, каким образом можно контролировать пути URL, количество версий каждой страницы и публикации, которые сохраняются в базе данных CMS, и кэширование для повышения масштабируемости и производительности.

1. Слева в меню на панели навигации нажмите ссылку Config (Конфигурация). Обратите внимание на некоторые общие настройки конфигурации, в том числе приведенные в следующем списке.

- Иерархические ярлыки страниц — по умолчанию если владелец контента создает иерархическое дерево страниц, то путь URL-адреса будет использовать иерархические ярлыки, как показано в следующем примере пути URL: /about-us/news-and-events/northwind-wins-award.
- Закрыть комментарии после — значение по умолчанию: 0 дней, поэтому комментарии всегда открыты.
- Включить комментарии к сообщениям/страницам — по умолчанию сообщения разрешают комментарии, а страницы — нет.
- Утверждать комментарии — по умолчанию комментарии утверждаются автоматически. Было бы безопаснее отключить этот параметр, поэтому отключите его сейчас, но тогда вам нужно будет модерировать комментарии для отдельных страниц и сообщений.
- История — по умолчанию сохраняется десять ревизий каждого сообщения и страницы.
- Кэширование — по умолчанию страницы и сообщения не кэшируются, поэтому каждый запрос посетителя обслуживается путем загрузки содержимого из базы данных. Стандартные значения: 120 минут (два часа), 720 минут (12 часов) и 1440 минут (24 часа).

В процессе разработки отключайте кэширование страниц, устанавливая минуты в 0, поскольку это может вызвать путаницу, когда внесенные в код изменения не отображаются на сайте! После развертывания в рабочей среде войдите в систему и установите для этих параметров подходящие значения в зависимости от того, как часто владельцы контента обновляют страницы и насколько быстро ожидают, что пользователь сайта увидит эти изменения. Кэширование — это баланс.

2. Нажмите кнопку Save (Сохранить).

Тестирование нового контента

Рассмотрим, публикуются ли страницы Contact Us (Свяжитесь с нами) и Our Location (Наше местоположение) и, следовательно, видны ли они пользователям сайта.

1. Слева в меню на панели навигации нажмите ссылку Logout (Выйти).
2. Перейдите на сайт по адресу https://localhost:5001/ и обратите внимание, что теперь отображаются страница Contact Us (Свяжитесь с нами) и запись в блоге (рис. 17.20).

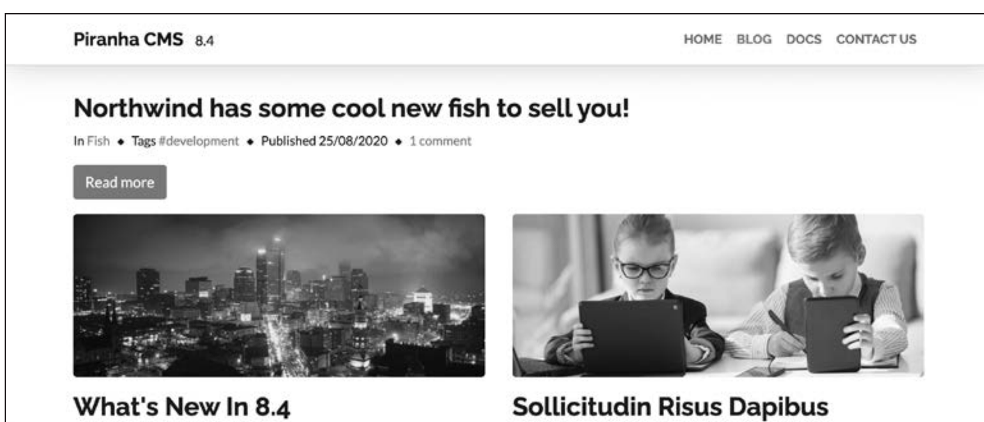


Рис. 17.20. Обновленный сайт Piranha CMS со ссылкой на страницу Contact Us (Свяжитесь с нами), расположенной в верхнем меню

3. В меню навигации отображаются только страницы верхнего уровня, поэтому нажмите ссылку About Us (О нас), чтобы перейти на данную страницу, а затем в адресной строке браузера добавьте оставшуюся часть слага и нажмите клавишу Enter, как показано в следующем URL: https://localhost:5001/about-us/our-location.

   На существующем сайте вы можете обеспечить навигацию к дочерним страницам, используя нечто наподобие навигационной панели Bootstrap `navbar` с выпадающим меню.

Прочитать о navbar фреймворка Bootstrap можно на сайте https://getbootstrap.com/docs/4.5/components/navbar/.

4. Закройте браузер.
5. Остановите работу сайта с помощью сочетания клавиш Ctrl+C на панели TERMINAL (Терминал).

Маршрутизация

Piranha CMS использует обычную систему маршрутизации ASP.NET Core.

На текущем сайте у нас есть четыре страницы, представляющие собой ответы на HTTP-запросы по относительным путям (слаги):

- Home (Главная): `/` или `/home`;
- Blog (Блог): `/blog`;

- Docs (Документы): `/docs`;
- Contact Us (Свяжитесь с нами): `/contact-us`;
- Our Location (Наше местоположение): `/about-us/our-location`;

Когда поступает запрос на слаг, Piranha CMS ищет в базе данных контента соответствующий элемент. При его обнаружении система ищет тип контента, поэтому для `/contact-us` обнаружится, что это стандартная страница. Если совпадение слага не найдено, возвращается HTTP-ответ 404 Missing resource.

Стандартная и архивная страницы для корректной работы не нуждаются в специально настроенных маршрутах, поскольку следующие маршруты заданы по умолчанию:

- `/page` для всех типов страниц, кроме страниц архива;
- `/archive` для архивных страниц;
- `/post` для сообщений в архиве;
- `/post/comment` для комментариев к сообщению.

Таким образом, URL входящих HTTP-запросов преобразуются в маршрут, который ASP.NET Core может обрабатывать обычным способом. Например, запрос https://localhost:5001/about-us переводится с помощью Piranha CMS на https://localhost:5001/page?id=154b519d-b5ed-4f75-8aa4-d092559363b0.

Это обрабатывается обычным контроллером ASP.NET Core MVC. Идентификатор страницы — GUID, который можно использовать для поиска данных содержимого страницы в базе данных Piranha CMS. Рассмотрим на примере.

1. В папке Controllers найдите и откройте файл `CmsController.cs`.
2. Обратите внимание, что класс `CmsController` — производный от класса Microsoft `Controller`.
3. Прокрутите вниз, чтобы найти метод действия `Page`, и обратите внимание: он использует атрибут Microsoft `[Route]`, чтобы указать следующее: этот метод действия отвечает на HTTP-запросы для относительного пути URL, `/page`, извлекает GUID с помощью привязки модели, определенной Microsoft, и применяет его для поиска модели страницы из базы данных, задействуя API Piranha, как показано ниже:

```
/// <summary>
/// Gets the page with the given id.
/// </summary>
/// <param name="id">The unique page id</param>
/// <param name="draft">If a draft is requested</param>
[Route("page")]
public async Task<IActionResult> Page(
```

```
    Guid id, bool draft = false)
  {
    try
    {
      var model = await _loader.GetPageAsync<StandardPage>(
        id, HttpContext.User, draft);

      return View(model);
    }
    catch (UnauthorizedAccessException)
    {
      return Unauthorized();
    }
  }
```

Обратите внимание: этот контроллер также содержит аналогичные методы действия с переопределенными упрощенными маршрутами для архивных страниц и сообщений, а также метод действия для обработки комментариев и сохранения их в базе данных.

Чтобы запрос не обрабатывался повторно системой Piranha, к переписанному URL добавляется параметр строки запроса `piranha_handled=true`.

Помимо встроенных переопределенных маршрутов, вы можете определить и дополнительные. Например, если вы создаете сайт электронной коммерции, то вам могут потребоваться специальные маршруты для каталогов и категорий товаров:

- *каталог товаров*: `/catalog`;
- *категории товаров*: `/catalog/beverages`.

О расширенной маршрутизации для системы Piranha CMS можно прочитать на сайте http://piranhacms.org/docs/application/advanced-routing.

Мультимедиа

Файлы мультимедиа могут загружаться через пользовательский интерфейс менеджера или программно с помощью API, предоставляемого Piranha CMS, через потоки и байтовые массивы.

Подробнее о программной загрузке медиафайлов с помощью API Piranha CMS можно прочитать на сайте http://piranhacms.org/docs/content/media.

Чтобы повысить вероятность того, что загруженный медиафайл совместим с обычными устройствами, Piranha CMS по умолчанию ограничивает типы медиафайлов, которые могут быть загружены, следующим набором типов:

- `.jpg`, `.jpeg` и `.png` для изображений;
- `.mp4` для видео;
- `.pdf` для документов.

Если вам необходимо загрузить другие типы файлов, например, GIF-изображения, то вы можете зарегистрировать дополнительные типы файлов мультимедиа в классе `Startup`.

1. Откройте файл `Startup.cs`.
2. В методе `Configure` после инициализации системы Piranha зарегистрируйте расширение файла GIF в качестве допустимого типа файла, как показано ниже (выделено полужирным шрифтом):

```
// инициализация системы Piranha
App.Init(api);

// регистрация GIF-файлов в качестве типа медиафайлов
App.MediaTypes.Images.Add(".gif", "image/gif");
```

Сервис приложения

Класс `ApplicationService` упрощает программный доступ к общим объектам для текущего запроса. Обычно он внедряется во все файлы Razor с помощью `_ViewImports.cshtml` под именем `WebApp`:

```
@inject Piranha.AspNetCore.Services.IApplicationService WebApp
```

В табл. 17.2 приведены наиболее используемые функции сервиса приложений.

Таблица 17.2

Код	Описание
@WebApp.PageId	Guid запрашиваемой страницы
@WebApp.Url	URL, изначально запрошенный браузером, до того, как он был перезаписан промежуточным программным обеспечением
@WebApp.Api	Доступ к полному API Piranha
@WebApp.Media.ResizeImage (ImageField image, int width, int? height = null)	Изменяет размер указанного ImageField до указанных размеров и возвращает сгенерированный URL к файлу с измененным размером. ImageField — это тип Piranha, который может ссылаться на загруженное изображение

Типы контента

Система Piranha позволяет разработчику определять три категории *типа контента*.

- *Сайты* используются для свойств, общих для всего другого контента. При отсутствии надобности делиться свойствами вам не нужен это тип контента. Даже если он вам необходим, каждому сайту обычно требуется только такой тип. Класс должен быть дополнен атрибутом `[SiteType]`. Шаблон проекта `piranha.mvc` не включает пример настраиваемого сайта.
- *Страницы* применяются для информационных страниц, таких как Contact Us (Свяжитесь с нами), и целевых, таких как главная страница или страница категории, в которой могут быть другие страницы в качестве дочерних. Страницы образуют иерархическое дерево, обеспечивающее структуру URL сайта, например `/contact-us/our-location` и `/about-us/job-vacancies`. Каждый сайт обычно содержит несколько типов страниц, таких как стандартная и архивная страницы, страницы категории и товара. Класс должен быть дополнен атрибутом `[PageType]`.
- *Сообщения* используются для «страниц», у которых не может быть дочерних страниц и которые могут быть перечислены только на архивных страницах. Сообщения можно отфильтровать и сгруппировать по дате, категории и тегам. Специальный тип страницы «архив» обеспечивает взаимодействие пользователя с сообщениями. Каждый сайт обычно имеет только один или два типа сообщений, например `NewsPost` (Новостное сообщение) или `EventPost` (Сообщение о мероприятии). Класс должен быть дополнен атрибутом `[PostType]`.

Типы компонентов

Зарегистрированные типы контента имеют встроенную поддержку функций создания, редактирования и удаления через менеджер Piranha. Структура типа контента обеспечивается путем его разделения на три категории *типов компонентов*, как показано в перечне ниже.

- *Поля* — самый маленький компонент контента. Поля могут быть, например, просто числом, датой или строковым значением. Поля похожи на свойства в классе C#. Свойство должно быть дополнено атрибутом `[Field]`.
- *Регионы* — небольшие фрагменты контента, которые появляются в фиксированном месте на странице или в сообщении и показываются пользователю способом, контролируемым разработчиком. Регионы состоят из одного или нескольких полей либо сортируемой коллекции полей.

  Регионы подобны сложным именованным свойствам в классе C#. Свойство должно быть дополнено атрибутом `[Region]`.

- *Блоки* — небольшие фрагменты контента, которые можно добавлять, переупорядочивать и удалять. Блоки предоставляют полную гибкость редактору контента. По умолчанию все страницы и сообщения могут содержать любое количество блоков, хотя это можно отключить для конкретного типа страницы или сообщения, установив свойству `UseBlocks` атрибута `[PageType]` значение `false`. Стандартные типы блоков включают многостолбцовый форматированный текст, цитаты и изображения. Разработчики могут определять пользовательские типы блоков с особым способом редактирования в менеджере Piranha.

Стандартные поля

Стандартные поля имеют заранее определенный способ редактирования и включают следующие элементы:

- `CheckBoxField`, `DateField`, `NumberField`, `StringField`, `TextField` — поля с простыми значениями;
- `PageField` и `PostField` — ссылка на страницу или запись по GUID;
- `DocumentField`, `ImageField`, `VideoField`, `MediaField` — ссылка на документ, изображение, видео или любой мультимедийный файл по GUID. По умолчанию `DocumentField` может быть в формате `.pdf`, `ImageField` — в формате `.jpg`, `.jpeg` или `.png`, `VideoField` — в формате `.mp4`, а `MediaField` — любым типом файла;
- `HtmlField`, `MarkdownField` — форматированные текстовые значения с настраиваемым редактором TinyMCE с панелью инструментов и редактором Markdown-разметки.

Настройка редактора форматированного текста

По умолчанию Piranha CMS включает текстовый редактор TinyMCE с открытым исходным кодом.

Более подробную информацию о редакторе TinyMCE можно узнать на сайте https://www.tiny.cloud/docs/.

Редактор TinyMCE настраивается с помощью файла `editorconfig.json`, как показано ниже.

```
{
  "plugins": "autoresize autolink code hr paste lists piranhalink piranhaimage",
  "toolbar": "bold italic | bullist numlist hr | alignleft aligncenter alignright
  | formatselect styleselect | piranhalink piranhaimage",
  "blockformats": "Paragraph=p;Header 1=h1;Header 2=h2;Header 3=h3;Header 4=h4;
  Code=pre;Quote=blockquote",
```

```
  "styleformats": [
    { "title": "Small", "tag": "small", "type": "inline" },
    { "title": "Code", "tag": "code", "type": "format" },
    { "title": "Lead", "tag": "p", "type": "block", "classes": "lead" },
    { "title": "Button Primary", "tag": "a", "type": "inline", "classes": "btn
    btn-primary" },
    { "title": "Button Light", "tag": "a", "type": "inline", "classes": "btn
    btn-light" }
  ]
}
```

Для включения встроенных ссылок и изображений Piranha CMS предоставляет два настраиваемых плагина TinyMCE.

Некоторые типы стандартной страницы

Рассмотрим некоторые типы контента, определенные шаблоном проекта Piranha Blog.

1. Найдите и откройте файл `Models/StandardPage.cs`. Обратите внимание: тип страницы должен наследоваться от `Page<T>`, где `T` — этот самый производный класс и должен быть дополнен атрибутом `[PageType]`:

```
using Piranha.AttributeBuilder; // [PageType]
using Piranha.Models; // Page<T>

namespace NorthwindCms.Models
{
  [PageType(Title = "Standard page")]
  public class StandardPage : Page<StandardPage>
  {
  }
}
```

2. Чтобы просмотреть источник, выберите `Page<StandardPage>` и нажмите клавишу `F12`.
3. Теперь выберите `GenericPage<T>` и нажмите клавишу `F12`.
4. Выберите `PageBase` и нажмите клавишу `F12`.
5. Проанализируйте класс `PageBase`, как показано ниже. Обратите внимание, что каждая страница имеет следующие свойства:
   - `SiteId` и `ParentId` — значения `Guid`;
   - блоки, представляющие собой список экземпляров блоков:

```
#region Assembly Piranha, Version=8.4.2.0, Culture=neutral,
PublicKeyToken=null
// Piranha.dll
```

```
#endregion
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Piranha.Extend;

namespace Piranha.Models
{
  public abstract class PageBase :
    RoutedContentBase, IBlockContent, IMeta, ICommentModel
  {
    protected PageBase();

    public Guid SiteId { get; set; }
    [StringLength(256)]
    public Guid? ParentId { get; set; }
    public int SortOrder { get; set; }
    [StringLength(128)]
    public string NavigationTitle { get; set; }
    public bool IsHidden { get; set; }
    [StringLength(256)]
    public string RedirectUrl { get; set; }
    public RedirectType RedirectType { get; set; }
    public Guid? OriginalPageId { get; set; }
    public IList<Block> Blocks { get; set; }
    public bool EnableComments { get; set; }
    public int CloseCommentsAfterDays { get; set; }
    public int CommentCount { get; set; }
    public bool IsCommentsOpen { get; }
    public bool IsStartPage { get; }
  }
}
```

6. Чтобы просмотреть источник, выберите `RoutedContentBase` и нажмите клавишу `F12`. Обратите внимание, что это класс, который определяет такие свойства, как `Slug`, для имени сегмента, используемого в путях URL, метаданных SEO и OpenGraph, таких как описание и ключевые слова, основное изображение и отрывок, а также дата публикации.
7. Теперь, чтобы просмотреть источник, выберите `ContentBase` и нажмите клавишу `F12`. Обратите внимание, что в нем содержится:
   - `Id` — `Guid`;
   - `TypeId` — `string`;
   - `Title` — `string`;
   - значения `DateTime` для времени создания и последнего изменения.

```
using System;
using System.ComponentModel.DataAnnotations;
```

```
namespace Piranha.Models
{
  public abstract class ContentBase
  {
    protected ContentBase();

    public Guid Id { get; set; }
    [StringLength(64)]
    public string TypeId { get; set; }
    [StringLength(128)]
    public string Title { get; set; }
    public IList<string> Permissions { get; set; }
    public DateTime Created { get; set; }
    public DateTime LastModified { get; set; }
  }
}
```

8. Найдите и откройте файл `Views/Cms/Page.cshtml`. Обратите внимание, что это представление:
   - строго типизировано и требует экземпляра класса `StandardPage`, который будет использован как значение свойства `Model`;
   - сохраняет `Title` в `ViewData`, чтобы его можно было представить в общем макете;
   - отображает дополнительные метатеги из модели;
   - отображает `Title` и `Blocks` в элементе `<div>` со стилями Bootstrap, как показано ниже:

```
@model StandardPage
@{
  ViewData["Title"] = !string.IsNullOrEmpty(Model.MetaTitle)
    ? Model.MetaTitle : Model.Title;
  var hasImage = Model.PrimaryImage.HasValue;
}
@section head {
  @WebApp.MetaTags(Model)
}

<header @(hasImage ? "class=has-image" : "") @(hasImage ?
$"style=background-image:url({ @Url.Content(WebApp.Media.
ResizeImage(Model.PrimaryImage, 1920, 400)) })" : "")>
  <div class="dimmer"></div>
  <div class="container text-center">
    <h1>@Model.Title</h1>
    @if (!string.IsNullOrWhiteSpace(Model.Excerpt))
    {
      <div class="row justify-content-center">
        <div class="col-lg-8 lead">
          @Html.Raw(Model.Excerpt)
        </div>
      </div>
```

```
    }
  </div>
</header>

<main>
  @foreach (var block in Model.Blocks)
  {
    <div class="block @block.CssName()">
      <div class="container">
        @Html.DisplayFor(m => block, block.GetType().Name)
      </div>
    </div>
  }
</main>
```

Некоторые типы архивной страницы и контента

Для архивов блогов на сайте необходимо определить как минимум два типа: тип страницы и тип сообщения. Первый действует как контейнер для перечисления, фильтрации и группировки сообщений в блоге.

1. Откройте файл `Models/StandardArchive.cs.` Обратите внимание, что это тип страницы, поэтому он дополнен атрибутом `[PageType]`, но у него также есть свойство `IsArchive`, установленное в значение `true`, и свойство с именем `Archive` типа `PostArchive <PostInfo>`:

```
using Piranha.AttributeBuilder;
using Piranha.Models;

namespace NorthwindCms.Models
{
  [PageType(Title = "Standard archive", IsArchive = true)]
  public class StandardArchive : Page<StandardArchive>
  {
    public PostArchive<PostInfo> Archive { get; set; }
  }
}
```

2. Найдите и откройте файл `Models/StandardPost.cs`. Обратите внимание: тип записи должен наследоваться от `Post<T>`, где `T` — этот самый производный класс и должен быть дополнен атрибутом `[PostType]`, и это сообщение содержит одно свойство для хранения любых комментариев.

```
using System.Collections.Generic;
using Piranha.AttributeBuilder;
using Piranha.Models;
namespace NorthwindCms.Models
{
  [PostType(Title = "Standard post")]
  public class StandardPost : Post<StandardPost>
```

```
  {
    public IEnumerable<Comment> Comments
      { get; set; } = new List<Comment>();
  }
}
```

Стандартные блоки

Стандартные блоки включают такие элементы, как:

- *столбцы* — содержит свойство `Items` с одним или несколькими элементами, представляющими собой экземпляры класса `Block`;
- *изображение* — содержит свойство `Body` типа `ImageField` с представлением, которое выводит его как `src` для элемента `<img>`;
- *цитата* — содержит свойство `Body` типа `TextField` с представлением, которое выводит его, заключенным в элемент `<blockquote>`;
- *текст* — содержит свойство `Body` типа `TextField`.

Типы компонентов и стандартных блоков

Рассмотрим некоторые типы компонентов, определенные в шаблоне проекта.

1. Добавьте в папку Models новый временный класс без пространства имен `ExploreBlocks.cs` и введите операторы для определения свойств, по одному для каждого типа блока:

   ```
   using Piranha.Extend.Blocks;

   class ExploreBlocks
   {
     AudioBlock ab;
     HtmlBlock hb;
     ColumnBlock cb;
     ImageBlock ib;
     ImageGalleryBlock igb;
     PageBlock pb;
     QuoteBlock qb;
     SeparatorBlock sb;
     TextBlock tb;
     VideoBlock vb;
   }
   ```

2. Щелкните кнопкой мыши внутри каждого типа блока, нажмите клавишу F12, просмотрите его определение и обратите внимание, что для определения блока

класс должен наследоваться от Block и содержать атрибут [BlockType]. Например, класс HtmlBlock:

```
#region Assembly Piranha, Version=8.4.2.0, Culture=neutral,
PublicKeyToken=null
// Piranha.dll
#endregion

using Piranha.Extend.Fields;

namespace Piranha.Extend.Blocks

{
  [BlockType(Name = "Content", Category = "Content",
    Icon = "fas fa-paragraph", Component = "html-block")]
  public class HtmlBlock : Block, ISearchable
  {
    public HtmlBlock();
    public HtmlField Body { get; set; }
    public string GetIndexedContent();
    public override string GetTitle();
  }
}
```

3. Найдите и откройте файл Views/Cms/DisplayTemplates/HtmlBlock.cshtml. Вы увидите, что он отображает блок, просто отображая необработанный HTML-код, хранящийся в свойстве Body:

```
@model Piranha.Extend.Blocks.HtmlBlock

@Html.Raw(Model.Body)
```

4. Найдите и откройте файл Views/Cms/DisplayTemplates/ColumnBlock.cshtm. Вы увидите, что он обрабатывает коллекцию элементов блока внутри элементов <div>, стилизованных с помощью Bootstrap:

```
@model Piranha.Extend.Blocks.ColumnBlock

<div class="row">
  @for (var n = 0; n < Model.Items.Count; n++)
  {
    <div class="col-md">
      @Html.DisplayFor(m => Model.Items[n],
        Model.Items[n].GetType().Name)
    </div>
  }
</div>
```

5. Проанализируйте модели и представления для других четырех встроенных типов блоков.

6. Закомментируйте весь класс или удалите файл из вашего проекта. Мы создали его только для проверки реализации встроенных типов блоков.

С системой сеток Bootstrap можно познакомиться на сайте https://getbootstrap.com/docs/4.1/layout/grid/.

Определение компонентов, типов контента и шаблонов

Теперь, когда вы ознакомились с функциональностью типов контента и компонентов, предоставляемых шаблоном проекта, рассмотрим более подробно, как они были определены, а затем определим некоторые пользовательские типы страниц, чтобы отобразить каталог товаров `Northwind`.

Сначала мы создадим контроллер MVC, отвечающий на HTTP-запрос GET для относительного пути `/import`, запрашивая в базе данных `Northwind` категории и их товары.

Затем, используя API Piranha CMS, мы найдем специальную страницу, представляющую корневой каталог товров, и в качестве дочерних элементов этой страницы мы программно создадим экземпляры настраиваемого типа `CategoryPage` с настраиваемой областью для хранения таких сведений, как имя, описание и изображение каждой категории, а также список экземпляров настраиваемого региона для хранения сведений о каждом товаре, включая название, цену и единицы на складе (рис. 17.21).

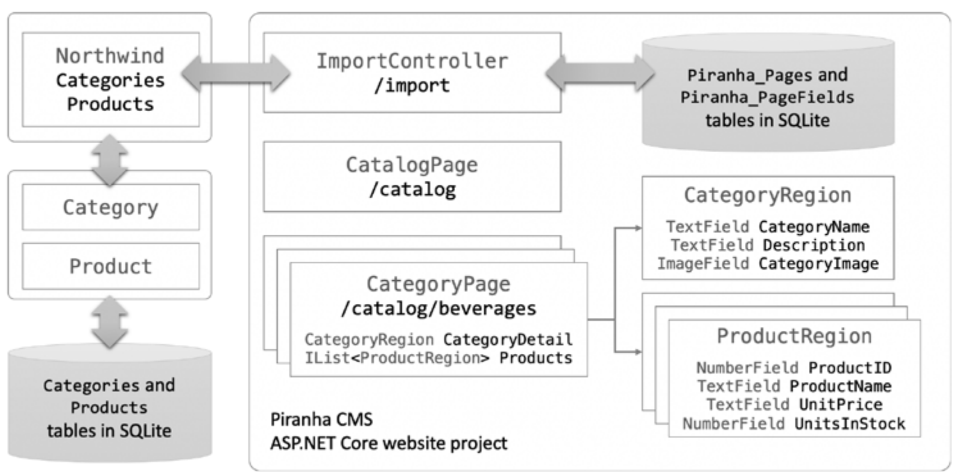


Рис. 17.21. Информационная связь сайта

Создание пользовательских регионов

Начнем с создания пользовательских регионов для категории и товара, чтобы данные могли храниться в базе данных Piranha CMS и редактироваться владельцем контента через пользовательский интерфейс менеджера.

1. В папку Models/Regions добавьте новый файл `CategoryRegion.cs` и операторы, чтобы определить регион для хранения сведений о категории из базы данных `Northwind`, используя подходящие типы полей:

```
using Piranha.Extend;
using Piranha.Extend.Fields;

namespace NorthwindCms.Models
{
  public class CategoryRegion
  {
    [Field(Title = "Category ID")]
    public NumberField CategoryID { get; set; }

    [Field(Title = "Category name")]
    public TextField CategoryName { get; set; }

    [Field]
    public HtmlField Description { get; set; }

    [Field(Title = "Category image")]
    public ImageField CategoryImage { get; set; }
  }
}
```

2. В папку `Models/Regions` добавьте новый файл `ProductRegion.cs` и операторы, чтобы определить регион для хранения сведений о товаре из базы данных `Northwind`, используя подходящий тип полей:

```
using Piranha.Extend;
using Piranha.Extend.Fields;
using Piranha.Models;

namespace NorthwindCms.Models
{
  public class ProductRegion
  {
    [Field(Title = "Product ID")]
    public NumberField ProductID { get; set; }

    [Field(Title = "Product name")]
    public TextField ProductName { get; set; }

    [Field(Title = "Unit price", Options = FieldOption.HalfWidth)]
    public StringField UnitPrice { get; set; }
```

```
    [Field(Title = "Units in stock", Options = FieldOption.HalfWidth)]
    public NumberField UnitsInStock { get; set; }
  }
}
```

Создание сущностной модели данных

На момент написания Piranha CMS 8.4 версия .NET 5 не поддерживается, поэтому мы не можем использовать контекст базы данных EF Core и проекты классов модели сущностей. Однако, поскольку для импорта категорий и товаров нам необходимы только базовые функции, мы в текущем проекте Piranha CMS создадим новый упрощенный контекст базы данных `Northwind` и классы модели сущностей.

1. В папку Models добавьте класс `Northwind.cs`.
2. Добавьте операторы для определения трех классов: `Northwind`, `Category` и `Product`:

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

namespace Packt.Shared
{
  public class Category
  {
    public int CategoryID { get; set; }
    public string CategoryName { get; set; }
    public string Description { get; set; }

    public virtual ICollection<Product> Products { get; set; }

    public Category()
    {
      this.Products = new HashSet<Product>();
    }
  }

  public class Product
  {
    public int ProductID { get; set; }
    public string ProductName { get; set; }
    public decimal? UnitPrice { get; set; }
    public short? UnitsInStock { get; set; }
    public bool Discontinued { get; set; }
    public int CategoryID { get; set; }

    public virtual Category Category { get; set; }
  }
  public class Northwind : DbContext
  {
    public DbSet<Category> Categories { get; set; }
```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<Category>()
        .HasMany(c => c.Products)
        .WithOne(p => p.Category);

      modelBuilder.Entity<Product>()
        .HasOne(p => p.Category)
        .WithMany(c => c.Products);
    }
  }
}
```

Создание пользовательских типов страниц

Теперь необходимо определить пользовательские типы страниц для каталога и категории товара.

1. В папку `Models` добавьте класс `CatalogPage.cs`, который не допускает блокирования, не имеет области содержимого, поскольку заполняется из базы данных `Northwind`, и имеет собственный путь `/catalog`:

```
using Piranha.AttributeBuilder;
using Piranha.Models;

namespace NorthwindCms.Models
{
  [PageType(Title = "Catalog page", UseBlocks = false)]
  [PageTypeRoute(Title = "Default", Route = "/catalog")]
  public class CatalogPage : Page<CatalogPage>
  {
  }
}
```

2. В папку Models добавьте класс `CategoryPage.cs`, который не допускает блокирования, имеет собственный путь `/catalog-category`, свойство для хранения сведений о категории с использованием региона и свойство для хранения списка товаров также с использованием региона:

```
using Piranha.AttributeBuilder;
using Piranha.Extend;
using Piranha.Models;
using System.Collections.Generic;

  namespace NorthwindCms.Models
  {
    [PageType(Title = "Category Page", UseBlocks = false)]
```

```
    [PageTypeRoute(Title = "Default", Route = "/catalog-category")]
    public class CategoryPage : Page<CategoryPage>
    {
      [Region(Title = "Category detail")]
      [RegionDescription("The details for this category.")]
      public CategoryRegion CategoryDetail { get; set; }

      [Region(Title = "Category products")]
      [RegionDescription("The products for this category.")]
      public IList<ProductRegion> Products
        { get; set; } = new List<ProductRegion>();
  }
}
```

Создание пользовательских моделей представления

Далее необходимо определить некоторые типы для заполнения страницы каталога, поскольку она будет использовать структуру иерархии страниц для определения категорий товаров, которые будут отображаться на странице каталога.

1. В папку Models добавьте класс `CategoryItem.cs`, который содержит свойства для хранения сводной информации о категории, включая ссылки на ее изображение и полную страницу категории:

```
namespace NorthwindCms.Models
{
  public class CategoryItem
  {
    public string Title { get; set; }
    public string Description { get; set; }
    public string PageUrl { get; set; }
    public string ImageUrl { get; set; }
  }
}
```

2. В папку Models добавьте класс `CatalogViewModel.cs`, который содержит свойства для ссылки на страницу каталога и для хранения списка элементов сводной информации о категории:

```
using System.Collections.Generic;

namespace NorthwindCms.Models
{
  public class CatalogViewModel
  {
    public CatalogPage CatalogPage { get; set; }
    public IEnumerable<CategoryItem> Categories { get; set; }
  }
}
```

Определение пользовательских шаблонов контента для типов контента

Теперь необходимо определить контроллеры и представления для отображения типов контента. С помощью карты сайта мы станем выбирать дочерние элементы страницы каталога, чтобы определить, какие категории должны отображаться в нем.

1. Найдите и откройте файл `Controllers/CmsController.cs`, импортируйте пространства имен `System.Linq` и `NorthwindCMS.Models` и добавьте операторы, чтобы определить два новых метода действий, `Catalog` и `Category`, и соответствующие пути к ним, как показано ниже (выделено полужирным шрифтом):

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Piranha;
using Piranha.AspNetCore.Services;
using Piranha.Models;
using NorthwindCms.Models;
using System.Linq;

namespace NorthwindCms.Controllers
{
  [ApiExplorerSettings(IgnoreApi = true)]
  public class CmsController : Controller
  {
  ...

    [Route("catalog")]
    public async Task<IActionResult> Catalog(Guid id)
    {
      var catalog = await _api.Pages.GetByIdAsync<CatalogPage>(id);

      var model = new CatalogViewModel
      {
        CatalogPage = catalog,
        Categories = (await _api.Sites.GetSitemapAsync())
          // получение страницы каталога
          .Where(item => item.Id == catalog.Id)
          // получение дочерних страниц
          .SelectMany(item => item.Items)
          // получение страницы и возврат упрощенной модели
          // представления для каждого дочернего элемента карты сайта
          .Select(item =>
          {
            var page = _api.Pages.GetByIdAsync<CategoryPage>
              (item.Id).Result;
```

```
            var ci = new CategoryItem
            {
              Title = page.Title,
              Description = page.CategoryDetail.Description,
              PageUrl = page.Permalink,
              ImageUrl = page.CategoryDetail.CategoryImage
                .Resize(_api, 200)
            };
            return ci;
          })
      };
      return View(model);
    }

    [Route("catalog-category")]
    public async Task<IActionResult> Category(Guid id)
    {
      var model = await _api.Pages
        .GetByIdAsync<Models.CategoryPage>(id);
      return View(model);
    }
  }
}
```

Мы использовали для имени маршрута элемент `catalog-category`, поскольку уже существует маршрут `category`, использующийся для группировки сообщений в блогах по категориям.

2. В папку Views/Cms добавьте Razor-файл `Catalog.cshtml`:

```
@using NorthwindCms.Models
@model CatalogViewModel
@{
  ViewBag.Title = Model.CatalogPage.Title;
}
<div class="container">
  <div class="row justify-content-center">
    <div class="col-sm-10">
      <h1 class="display-3">@Model.CatalogPage.Title</h1>
    </div>
  </div>
  <div class="row">
  @foreach(CategoryItem c in Model.Categories)
  {
    <div class="col-sm-4">
      <a class="card-link" href="@c.PageUrl">
        <div class="card border-dark" style="width: 18rem;">
          <img class="card-img-top" src="@c.ImageUrl"
            alt="Image of @c.Title" asp-append-version="true" />
          <div class="card-body">
            <h5 class="card-title text-info">@c.Title</h5>
```

```
            <p class="card-text text-info">@c.Description</p>
          </div>
        </div>
      </a>
    </div>
  }
  </div>
</div>
```

3. В папку Views/Cms добавьте Razor-файл `Category.cshtml`:

```
@using NorthwindCms.Models
@model CategoryPage
@{
  ViewBag.Title = Model.Title;
}
<div class="container">
  <div class="row justify-content-center">
    <div class="col-sm-10">
      <h1 class="display-4">
        @Model.CategoryDetail.CategoryName
      </h1>
      <p class="lead">@Model.CategoryDetail.Description</p>
    </div>
  </div>
  <div class="row">
  @if (Model.Products.Count == 0)
  {
    <div class="col-sm-10">
      There are no products in this category!
    </div>
  }
  else
  {
    @foreach(ProductRegion p in Model.Products)
    {
    <div class="col-sm-4">
      <div class="card border-dark" style="width: 18rem;">
        <div class="card-header">
          In Stock: @p.UnitsInStock.Value
        </div>
        <div class="card-body">
          <h5 class="card-title text-info">
            <small class="text-muted">@p.ProductID.Value</small>
            @p.ProductName.Value
          </h5>
          <p class="card-text text-info">
            Price: @p.UnitPrice.Value
          </p>
        </div>
      </div>
```

```
      </div>
      }
    }
    </div>
  </div>
```

Настройка запуска и импорта из базы данных

Наконец, нужно настроить типы контента и строку подключения к базе данных Northwind.

1. Найдите и откройте файл `Startup.cs` и импортируйте пространство имен `System.IO`.
2. В конец метода `ConfigureServices` добавьте оператор для регистрации контекста базы данных `Northwind`, как показано ниже (выделено полужирным шрифтом):

```
public void ConfigureServices(IServiceCollection services)
{
  ...
  string databasePath = Path.Combine("..", "Northwind.db");
  services.AddDbContext<Packt.Shared.Northwind>(options =>
    options.UseSqlite($"Data Source={databasePath}"));
}
```

3. В папку Controllers добавьте класс `ImportController.cs`, чтобы определить контроллер для импорта категорий и товаров из базы данных `Northwind` в экземпляры новых пользовательских типов контента:

```
using Microsoft.AspNetCore.Mvc;
using Piranha;
using Piranha.Models;
using System;
using System.Linq;
using System.Threading.Tasks;
using Packt.Shared;
using NorthwindCms.Models;
using Microsoft.EntityFrameworkCore; // метод расширения Include()

namespace NorthwindCms.Controllers
{
  public class ImportController : Controller
  {
    private readonly IApi api;
    private readonly Northwind db;

    public ImportController(IApi api, Northwind injectedContext)
    {
```

```
    this.api = api;
    db = injectedContext;
  }

  [Route("/import")]
  public async Task<IActionResult> Import()
  {
    int importCount = 0;
    int existCount = 0;

    var site = await api.Sites.GetDefaultAsync();

    var catalog = await api.Pages
      .GetBySlugAsync<CatalogPage>("catalog");

    foreach (Category c in
      db.Categories.Include(c => c.Products))
    {
      // если страница категории уже существует,
      // то перейдите к следующей итерации цикла
      CategoryPage cp = await api.Pages.GetBySlugAsync<CategoryPage>(
        $"catalog/{c.CategoryName.ToLower().Replace(' ', '-') }");

      if (cp == null)
      {
        importCount++;

        cp = await CategoryPage.CreateAsync(api);

        cp.Id = Guid.NewGuid();
        cp.SiteId = site.Id;
      cp.ParentId = catalog.Id;
      cp.CategoryDetail.CategoryID = c.CategoryID;
      cp.CategoryDetail.CategoryName = c.CategoryName;
      cp.CategoryDetail.Description = c.Description;

      // поиск каталога с именем Categories
      Guid categoriesFolderID =
         (await api.Media.GetAllFoldersAsync())
        .First(folder => folder.Name == "Categories").Id;

      // поиск изображения с правильным именем файла
      // для идентификатора категории
      var image = (await api.Media
        .GetAllByFolderIdAsync(categoriesFolderID))
        .First(media => media.Type == MediaType.Image
        && media.Filename == $"category{c.CategoryID}.jpeg");

      cp.CategoryDetail.CategoryImage = image;
```

```
          if (cp.Products.Count == 0)
          {
            // преобразование товаров для этой категории
            // в список экземпляров ProductRegion
            cp.Products = c.Products
              .Select(p => new ProductRegion
              {
                ProductID = p.ProductID,
                ProductName = p.ProductName,
                UnitPrice = p.UnitPrice.HasValue
                  ? p.UnitPrice.Value.ToString("c") : "n/a",
                UnitsInStock = p.UnitsInStock ?? 0
              }).ToList();
          }

          cp.Title = c.CategoryName;
          cp.MetaDescription = c.Description;
          cp.NavigationTitle = c.CategoryName;
          cp.Published = DateTime.Now;
          await api.Pages.SaveAsync(cp);
        }
        else
        {
          existCount++;
        }
      }
        TempData["import_message"] = $"{existCount} categories already
        existed. {importCount} new categories imported.";

        return Redirect("~/");
      }
    }
  }
```

4. В папке Views\Cms найдите и откройте файл `Page.cshtml` и в верхнюю часть элемента `<main>` добавьте операторы, которые будут выводить сообщение об импорте, если оно было установлено:

```
<main>
  @if(TempData["import_message"] != null)
  {
  <div class="container">
    <div class="row">
      <div class="col">
        <div class="alert alert-info" role="alert">
          <h4 class="alert-heading">Import</h4>
          <p>@TempData["import_message"]</p>
        </div>
      </div>
    </div>
  </div>
  }
```

Создание контента с использованием шаблона проекта

Вы можете получить больше вдохновения о том, как программно работать с контентом, просмотрев файл класса SetupController.cs, который создает начальные страницы на примере сайта Piranha CMS. Например, как создать страницу, подобную Docs (Документы), которая перенаправляет на другую страницу:

```
// добавить страницу Docs
var docsPage = await StandardPage.CreateAsync(_api);
docsPage.Id = Guid.NewGuid();
docsPage.SiteId = site.Id;
docsPage.SortOrder = 1;
docsPage.Title = "Read The Docs";
docsPage.NavigationTitle = "Docs"; // используется для генерации слага
...
docsPage.RedirectUrl = "https://piranhacms.org/docs";
docsPage.RedirectType = RedirectType.Temporary;
docsPage.Published = DateTime.Now;
await _api.Pages.SaveAsync(docsPage);
```

Ниже показано, как добавить блоки на страницу, например домашнюю:

```
// добавить стартовую страницу
var startPage = await StandardPage.CreateAsync(_api);
...

startPage.Blocks.Add(new HtmlBlock
{
  Body = "<h2>Because First Impressions Last</h2>" +
    "<p class=\"lead\">All pages and posts you create have a primary image
and excerpt available that you can use both to create nice looking headers
for your content, but also when listing or linking to it on your site. These
fields are totally optional and can be disabled for each content type.</p>"
});

startPage.Blocks.Add(new ColumnBlock
{
  Items = new List<Block>()
  {
    new ImageBlock
    {
      Aspect = new SelectField<ImageAspect>
        { Value = ImageAspect.Widescreen },
      Body = images["concentrated-little-kids-...jpg"]
    },
    new HtmlBlock
    {
      Body = "<h3>Add, Edit & Rearrange</h3>" + ...
    }
  }
});
```

Тестирование сайта Northwind CMS

Теперь мы готовы запустить сайт.

Загрузка изображений и создание корня каталога

Начнем с загрузки нескольких изображений для использования в восьми категориях товаров, а затем создадим страницу каталога, которая будет служить корнем в иерархии страниц, в которую мы позже импортируем контент из базы данных `Northwind`.

Скачать изображения можно, перейдя по следующей ссылке: https://github.com/markjprice/cs9dotnet5/tree/master/Assets/Categories.

1. На панели TERMINAL (Терминал) введите команду `dotnet run`, чтобы создать и запустить сайт.
2. Запустите браузер Google Chrome. В адресной строке введите следующий URL: https://localhost:5001/manager/. Войдите в систему как администратор с именем `admin` и паролем `password`.
3. Слева в меню на панели навигации нажмите ссылку Media (Мультимедиа) и нажмите кнопку +, чтобы добавить папку Categories (Категории).
4. Выберите папку Categories (Категории) и импортируйте восемь изображений категорий (рис. 17.22).

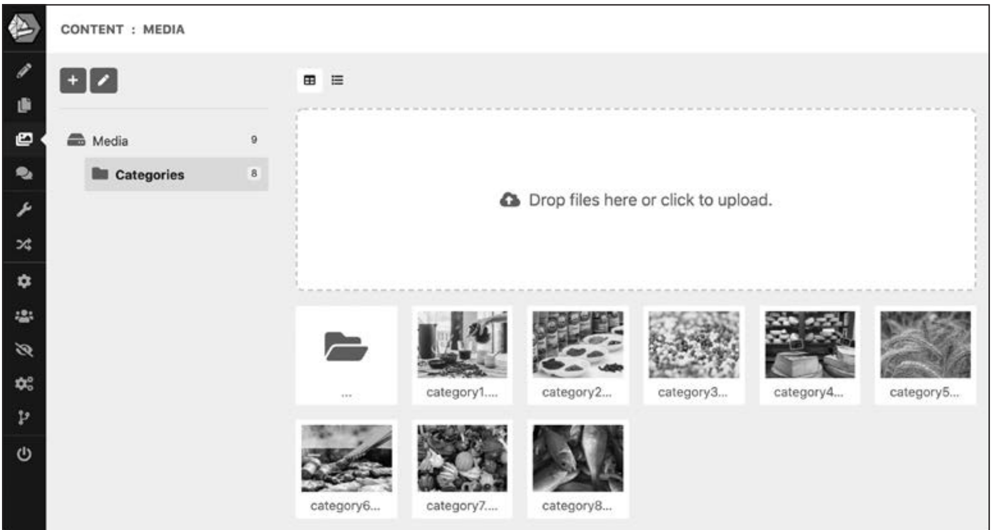


Рис. 17.22. Импорт изображений

5. Слева в меню на панели навигации нажмите ссылку Pages (Страницы), добавьте новую ссылку Catalog page (Страница каталога), измените ее название на Catalog (Каталог) и нажмите Publish (Опубликовать).

Импорт контента (содержимого) страниц категорий и товаров

В разделе CONTENT : PAGES (Содержание: страницы) владелец контента может вручную добавить под ссылкой Catalog (Каталог) новую ссылку Category page (Страница категории), но мы используем контроллер импорта, чтобы автоматически создать все страницы категорий и товаров.

1. В адресной строке браузера Google Chrome измените URL на https://localhost:5001/ и нажмите клавишу Enter.
2. Нажмите на ссылку Catalog (Каталог) и обратите внимание, что новая страница пуста.
3. В адресной строке браузера Google Chrome измените URL на https://localhost:5001/import/, нажмите клавишу Enter и обратите внимание, что после импорта категорий и товаров базы данных `Northwind` вы будете перенаправлены на главную страницу, где вы увидите сообщение об импорте, информирующее вас о том, что были импортированы восемь категорий. Если вы снова перейдете по адресу /import, будет отображено сообщение, что восемь категорий уже существуют и ни одна из них не была импортирована.
4. Выберите ссылку Catalog (Каталог) и обратите внимание, что категории были успешно импортированы (рис. 17.23).

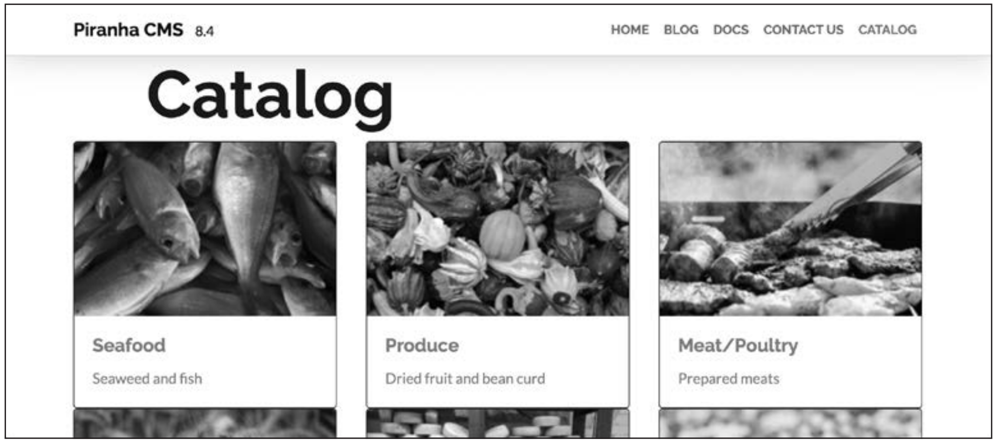


Рис. 17.23. Опубликованная страница Catalog (Каталог)

5. Выберите ссылку Meat/Poultry (Мясо/птица). Обратите внимание, что URL — https://localhost:5001/catalog/meat/poultry и содержит некоторые товары (рис. 17.24).

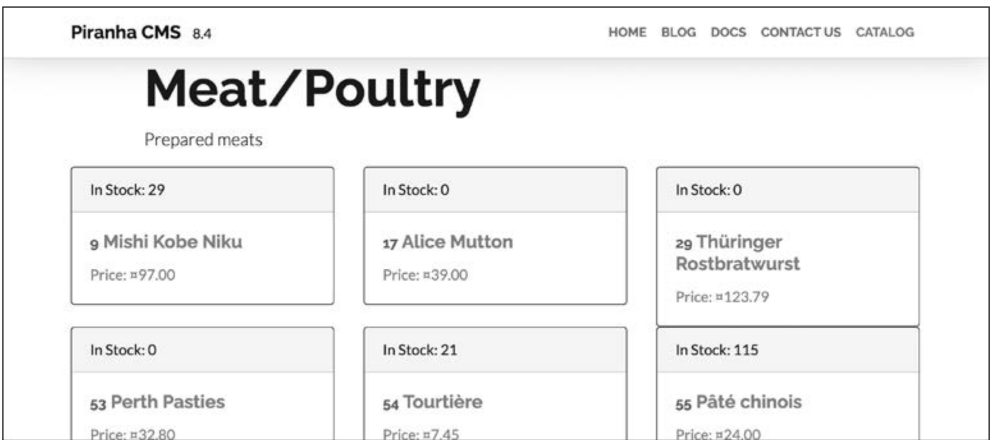


Рис. 17.24. Страница Meat/Poultry (Мясо/птица)

Управление контентом (содержимым) каталога

Теперь, когда мы импортировали контент каталога, владелец контента может использовать интерфейс менеджера Piranha, чтобы вносить изменения вместо редактирования исходных данных в базе данных `Northwind`.

1. В адресной строке браузера Google Chrome измените URL на https://localhost:5001/manager/ и при необходимости войдите в систему как администратор с именем `admin` и паролем `password`.
2. В разделе CONTENT : PAGES (Содержание: страницы) на странице Catalog (Каталог) выберите страницу Meat/Poultry (Мясо/птица) (рис. 17.25).
3. В разделе Meat/Poultry (Мясо/птица) вы увидите сведения о категории, включая ссылку на загрузку изображений, а затем нажмите ссылку Category products (Категория товаров).
4. В разделе Category products (Категория товаров) обратите внимание, что, хотя шесть строк представляют собой шесть наименований товаров, в строках не отображаются сведения о товаре. Вы можете написать расширения менеджера, чтобы улучшить представление о товаре.
5. Щелкните в любой строке на значке «многоточие», чтобы развернуть или свернуть эту строку, и обратите внимание, что администратор может редактировать данные, или щелкните на значке удаления, чтобы полностью удалить товар (рис. 17.26).
6. Закройте браузер.

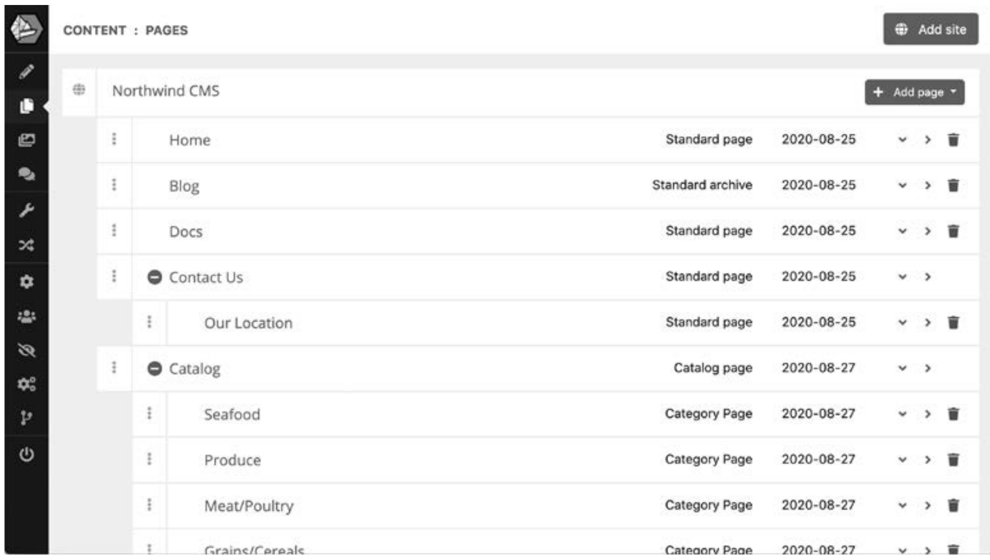


Рис. 17.25. Структура страницы Catalog (Каталог) в иерархии

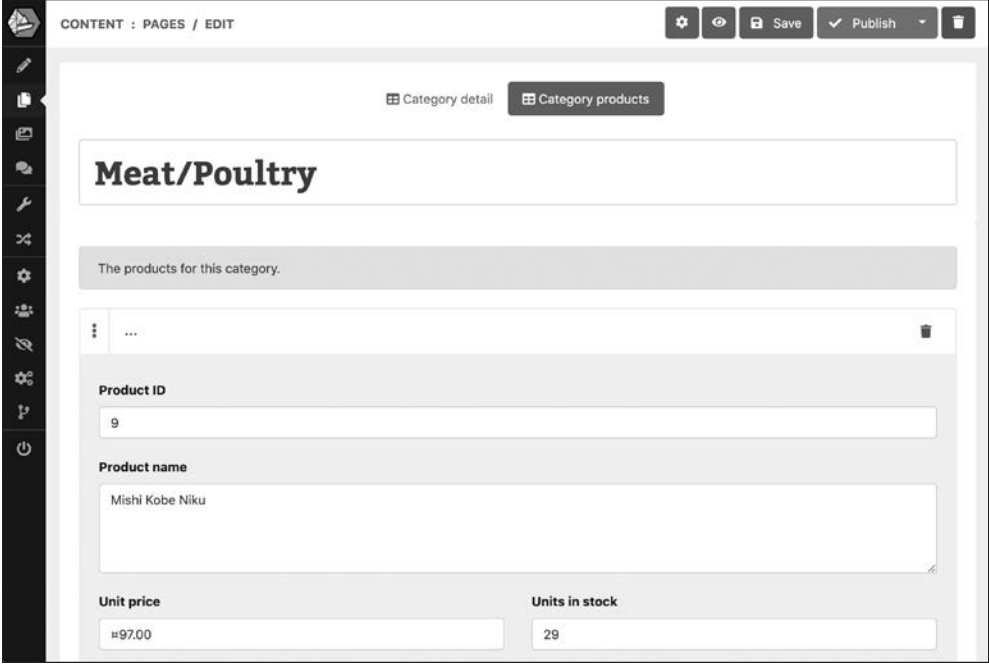


Рис. 17.26. Редактирование страницы Meat/Poultry (Мясо/птица)

Хранение контента в системе Piranha

Рассмотрим, как контент хранится в базе данных Piranha CMS.

1. Запустите программу SQLiteStudio.
2. Выберите Database ▸ Add a database (База данных ▸ Добавить базу данных) или нажмите сочетание клавиш Cmd+O.
3. Чтобы найти существующий файл базы данных на локальном компьютере, щелкните на значке желтой папки.
4. Перейдите в папку Code/PracticalApps/NorthwindCms, выберите базу данных `piranha.blog.db` и нажмите Open (Открыть).
5. В диалоговом окне Database (База данных) нажмите кнопку OK.
6. Для подключения к базе данных дважды щелкните на базе данных `piranha`.
7. Разверните пункт Tables (Таблицы), щелкните правой кнопкой мыши на таблице `Piranha_Pages` и выберите пункт контекстного меню Edit the table (Редактировать таблицу).
8. Перейдите на вкладку Data (Данные). Обратите внимание на значения столбцов, сохраненных для каждой страницы, включая LastModied, MetaDescription, NavigationTitle и PageTypeId (рис. 17.27).

Рис. 17.27. Вкладка Piranha_Pages Data

9. Щелкните правой кнопкой мыши на таблице `Piranha_PageFields` и выберите пункт контекстного меню Edit the table (Редактировать таблицу).
10. Перейдите на вкладку Data (Данные). Обратите внимание на значения столбцов, сохраненных для каждой страницы, включая CLRType, FieldId, RegionId и Value (рис. 17.28).

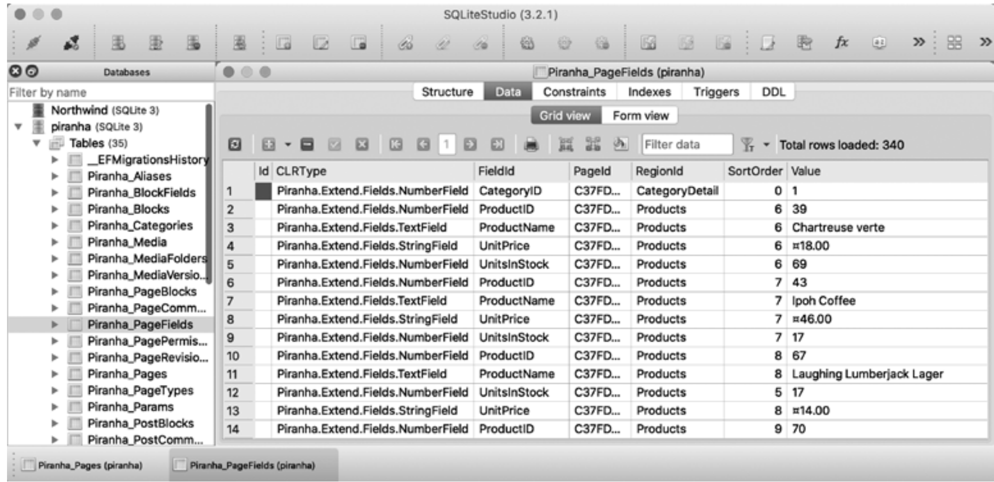


Рис. 17.28. Таблица Piranha_Pages

11. Щелкните правой кнопкой мыши на базе данных `piranha.blog` и выберите пункт контекстного меню Disconnect from the database (Отключение базы данных).
12. Закройте программу SQLiteStudio.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 17.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Каковы преимущества использования системы управления контентом для разработки сайта по сравнению с применением только ASP.NET Core?
2. Каков специальный относительный URL для доступа к UI управления Piranha CMS и какие имя пользователя и пароль настроены по умолчанию?
3. Что такое слаг?
4. В чем разница между сохранением контента и его публикацией?
5. Каковы три типа контента Piranha CMS и для чего они используются?
6. Каковы три типа компонентов Piranha CMS и для чего они применяются?
7. Перечислите три свойства, которые тип `Page` наследует от своих базовых классов, и объясните, для чего они используются.

8. Каким образом определить пользовательский тип региона?
9. Как определить маршруты для Piranha CMS?
10. Каким образом получить страницу из базы данных Piranha CMS?

Упражнение 17.2. Определение типа блока для отображения видео с YouTube

Прочитайте следующую статью, а затем определите тип блока со свойствами для управления такими параметрами, как автоматическое воспроизведение, с шаблоном отображения, использующим правильную разметку HTML: support.google.com/youtube/answer/171780.

Вам также следует обратиться к официальной документации Piranha (http://piranhacms.org/docs/extensions/blocks) для определения пользовательских блоков. Обратите внимание: для Piranha CMS 7.0 и более поздних версий определять представление менеджера для включения редактирования блоков необходимо с помощью Vue.js.

Упражнение 17.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- Piranha CMS: https://piranhacms.org/;
- репозиторий Piranha CMS: https://github.com/PiranhaCMS/piranha.core;
- вопросы о Piranha CMS на Stack Overflow: https://stackoverflow.com/questions/tagged/piranha-cms.

Резюме

Вы узнали, каким образом система управления веб-контентом позволяет разработчикам быстро создавать сайты, которые пользователи, не являющиеся техническими специалистами, могут задействовать для создания и управления собственным контентом. В качестве примера вы узнали о простой CMS с открытым исходным кодом на .NET Core под названием Piranha, ознакомились с некоторыми типами контента, предоставляемыми ее шаблоном проекта блога. И наконец, определили собственные регионы и типы страниц для работы с контентом, импортированным из базы данных `Northwind`.

Далее вы научитесь разрабатывать и использовать веб-сервисы.

18 Разработка и использование веб-сервисов

Эта глава посвящена тому, как разрабатывать веб-сервисы с помощью Web API в ASP.NET Core, а затем потреблять их, используя HTTP-клиенты, которые могут быть .NET-приложениями любого другого типа, включая сайт, настольное Windows-приложение или мобильное приложение.

Данная глава основана на знаниях и навыках, полученных в главах 11, 14 и 16.

В этой главе:

- разработка веб-сервисов с помощью Web API в ASP.NET Core;
- документирование и тестирование веб-сервисов;
- обращение к сервисам с помощью HTTP-клиентов;
- реализация расширенных функций;
- прочие коммуникационные технологии.

Разработка веб-сервисов с помощью Web API в ASP.NET Core

Прежде чем мы создадим современный веб-сервис, нам необходимо обсудить некоторые основы, чтобы задать контекст для этой главы.

Аббревиатуры, типичные для веб-сервисов

Хотя изначально протокол HTTP проектировался в целях передачи запросов и ответов, содержащих HTML и других ресурсы для просмотра человеком, с его помощью также можно создавать сервисы. Рой Филдинг, один из главных авторов спецификации протокола HTTP, в своей диссертации «Архитектурные стили и дизайн сетевых программных архитектур», где описана *«передача состояния представления»* (Representational State Transfer, REST), объясняет, что стандарт

HTTP отлично подходит для построения сервисов, так как определяет следующие понятия:

- URL для уникальной идентификации ресурсов, например https://localhost:5001/api/products/23;
- методы для выполнения типовых задач, такие как `GET`, `POST`, `PUT` и `DELETE`;
- возможность согласовывать медиатипы для обмена данными, такие как XML и JSON. Согласование содержимого происходит, когда клиент указывает заголовок запроса, например `Accept:application/xml,*/*;q=0.8`. Формат ответа по умолчанию, используемый Web API в ASP.NET Core, — JSON; это значит, одним из заголовков ответа будет `Content-Type:application/json; charset=utf-8`.

Более подробную информацию о медиа типах можно найти на сайте http://en.wikipedia.org/wiki/Media_type.

Веб-сервисы — это сервисы, использующие стандарт связи HTTP, вследствие чего их иногда называют HTTP-сервисами или сервисами RESTful. HTTP-сервисы или RESTful — главная тема данной главы.

Веб-сервисы также могут работать по *SOAP-протоколу* (Simple Object Access Protocol, простой протокол доступа к объектам), реализующему некоторые стандарты WS-*.

Более подробную информацию о стандартах WS-* можно получить на сайте https://en.wikipedia.org/wiki/List_of_web_service_specifications.

Microsoft .NET Framework 3.0 и более поздние версии включают технологию *удаленного вызова процедур (RPC)* под названием *Windows Communication Foundation (WCF)*, которая позволяет разработчикам легко создавать сервисы, включая сервисы SOAP, реализующие стандарты WS-*. Однако корпорация Microsoft считает технологию устаревшей и не поддерживает ее на современных платформах .NET.

gRPC — это система удаленного вызова процедур (RPC) с открытым исходным кодом, разработанная Google (g в gRPC).

Более подробную информацию об использовании gRPC можно получить на сайте https://devblogs.microsoft.com/premier-developer/grpc-asp-net-core-as-a-migration-path-for-wcfs- in-net-core /.

Разработка проекта Web API в ASP.NET Core

Создадим веб-сервис, предоставляющий способ работы с данными в базе данных Northwind, с помощью ASP.NET Core, чтобы данные могли использоваться любым клиентским приложением на любой платформе, которая может отправлять HTTP-запросы и получать HTTP-ответы.

1. В папке PracticalApps создайте папку NorthwindService.
2. В программе Visual Studio Code откройте рабочую область PracticalApps и добавьте папку NorthwindService.
3. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку NorthwindService.
4. На панели TERMINAL (Терминал) используйте шаблон webapi для создания нового проекта Web API в ASP.NET Core:

```
dotnet new webapi
```

5. Установите NorthwindService в качестве активного проекта и при появлении запроса добавьте необходимые ресурсы.
6. В папке Controllers найдите и откройте файл WeatherForecastController.cs, как показано ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;

namespace NorthwindService.Controllers
{
  [ApiController]
  [Route("[controller]")]
  public class WeatherForecastController : ControllerBase
  {
    private static readonly string[] Summaries = new[]
    {
      "Freezing", "Bracing", "Chilly", "Cool", "Mild",
      "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    };

    private readonly ILogger<WeatherForecastController> _logger;

    // Веб-API будет принимать только токены 1) для пользователей и
    // 2) при наличии области доступа access_as_user для данного API
```

```
    static readonly string[] scopeRequiredByApi =
      new string[] { "access_as_user" };

    public WeatherForecastController(
      ILogger<WeatherForecastController> logger)
    {
      _logger = logger;
    }

    [HttpGet]
    public IEnumerable<WeatherForecast> Get()
    {
      var rng = new Random();
      return Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        {
          Date = DateTime.Now.AddDays(index),
          TemperatureC = rng.Next(-20, 55),
          Summary = Summaries[rng.Next(Summaries.Length)]
        })
        .ToArray();
    }
  }
}
```

Просматривая предыдущий код, обратите внимание на следующие моменты.

- Класс `Controller` наследуется от `ControllerBase`. Эта процедура проще, чем класс `Controller`, используемый в MVC, поскольку он не содержит таких методов, как `View`, для генерации HTML-ответов с помощью файла Razor.
- Атрибут `[Route]` регистрирует относительный URL `weatherforecast` для клиентов, чтобы использовать его для выполнения HTTP-запросов, которые будут обрабатываться этим контроллером. Например, данный контроллер будет обрабатывать HTTP-запрос для https://localhost:5001/weatherforecast/. Некоторым разработчикам нравится добавлять к имени контроллера префикс `api/`, что является соглашением, позволяющим в проектах различать MVC и Web API. Если вы введете `[controller]`, как показано в примере, то будут использованы символы из имени класса до слова `Controller`, в данном случае `WeatherForecast`, или вы можете просто ввести другое имя без скобок, например `[Route("api/forecast")]`.
- Атрибут `[ApiController]` был введен в ASP.NET Core 2.1 и обеспечивает специфическое для REST поведение контроллеров, например автоматические HTTP-ответы с кодом состояния `400` для недопустимых моделей, как вы увидите далее в этой главе.
- Поле `scopeRequiredByApi` можно использовать для добавления авторизации, чтобы гарантировать, что ваш веб-API будет вызываться только клиент-

скими приложениями и сайтами от имени пользователей, имеющими необходимые области.

Более подробную информацию о токенах можно получить на сайте https://docs.microsoft.com/ru-ru/azure/active-directory/develop/scenario-protected-web-api-verification-scope-app-roles.

- Атрибут [HttpGet] регистрирует метод Get в классе Controller для ответа на HTTP-запросы GET, а его реализация использует объект Random, чтобы возвратить массив значений WeatherForecast со случайными температурами и сводками погоды, такими как Bracing или Balmy, в течение следующих пяти дней.

7. Добавьте второй метод Get с целочисленным параметром days, который при вызове позволяет указывать, за какое количество дней должен быть прогноз.
   - В начале метода добавьте комментарий, чтобы показать результаты метода GET и URL-адрес.
   - Добавьте новый метод с целочисленным параметром с именем days.
   - Вырежьте и вставьте исходные операторы кода реализации метода Get в новый метод Get.
   - Измените его для создания интерфейса IEnumerable из значений int вплоть до запрошенного количества дней и отредактируйте исходный метод Get для вызова нового метода Get и передачи значения 5:

```
// GET /weatherforecast
[HttpGet]
public IEnumerable<WeatherForecast> Get() // исходный метод
{
  return Get(5); // пятидневный прогноз
}

// GET /weatherforecast/7
[HttpGet("{days:int}")]
public IEnumerable<WeatherForecast> Get(int days) // новый метод
{
  var rng = new Random();

  return Enumerable.Range(1, days).Select(index =>
    new WeatherForecast
    {
      Date = DateTime.Now.AddDays(index),
      TemperatureC = rng.Next(-20, 55),
      Summary = Summaries[rng.Next(Summaries.Length)]
    })
    .ToArray();
}
```

Обратите внимание на шаблон формата, ограничивающий параметр `days` значениями `int` в атрибуте `[HttpGet]`.

Более подробную информацию об ограничениях маршрута можно получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/routing#route-constraint-reference.

Функциональность веб-сервисов

Проверим функциональность веб-сервисов.

1. На панели TERMINAL (Терминал) запустите сайт с помощью команды `dotnet run`.
2. Запустите браузер Google Chrome. В адресной строке введите следующий URL: https:/localhost:5001/. Обратите внимание: вы получите ответ с кодом состояния 404, поскольку мы не включили статические файлы и отсутствует файл `index.html`, а также контроллер MVC с настроенным маршрутом. Помните, что этот проект не предназначен для просмотра и взаимодействия с пользователем.
3. В браузере Google Chrome отобразите панель Developer tools (Инструменты разработчика). Затем в адресной строке введите следующий URL: https:/localhost:5001/weatherforecast. Обратите внимание: сервис Web API должен возвращать документ JSON с пятью объектами случайного прогноза погоды в массиве (рис. 18.1).

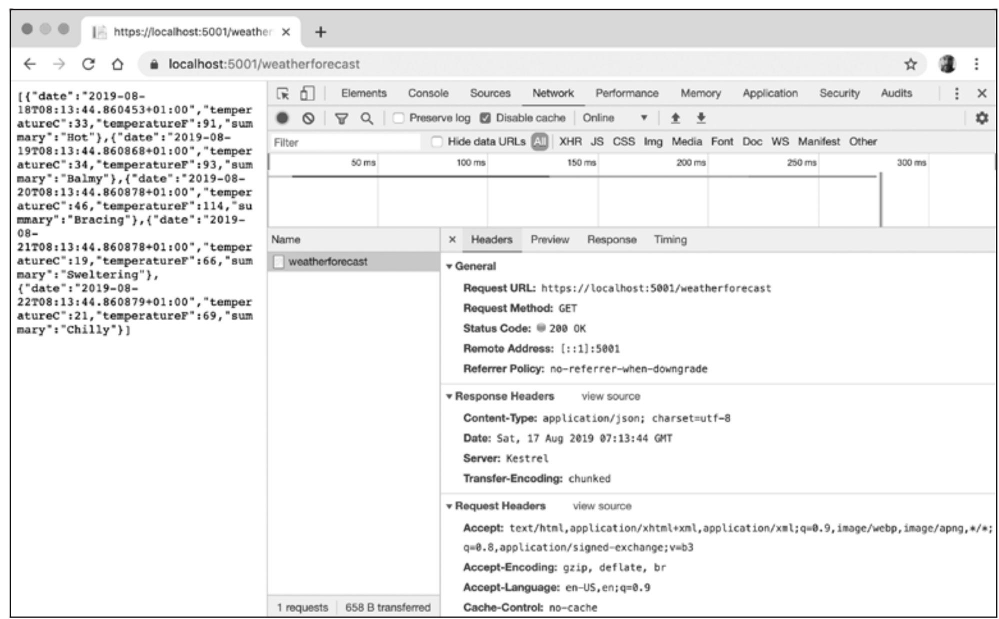


Рис. 18.1. Прогноз погоды. Запрос и ответ от веб-сервиса

4. Закройте панель Developer Tools (Инструменты разработчика).
5. Перейдите по адресу https:/localhost:5001/weatherforecast/14 и обратите внимание на ответ при запросе двухнедельного прогноза погоды (рис. 18.2).

https://localhost:5001/weather
localhost:5001/weatherforecast/14

[{"date":"2019-08-18T08:16:42.178169+01:00","temperatureC":5,"temperatureF":40,"summary":"Cool"},{"date":"2019-08-
19T08:16:42.178197+01:00","temperatureC":39,"temperatureF":102,"summary":"Scorching"},{"date":"2019-08-
20T08:16:42.178198+01:00","temperatureC":50,"temperatureF":121,"summary":"Warm"},{"date":"2019-08-
21T08:16:42.178199+01:00","temperatureC":51,"temperatureF":123,"summary":"Scorching"},{"date":"2019-08-
22T08:16:42.178199+01:00","temperatureC":18,"temperatureF":64,"summary":"Hot"},{"date":"2019-08-
23T08:16:42.178199+01:00","temperatureC":35,"temperatureF":94,"summary":"Mild"},{"date":"2019-08-
24T08:16:42.1782+01:00","temperatureC":10,"temperatureF":49,"summary":"Mild"},{"date":"2019-08-
25T08:16:42.1782+01:00","temperatureC":50,"temperatureF":121,"summary":"Balmy"},{"date":"2019-08-
26T08:16:42.178201+01:00","temperatureC":1,"temperatureF":33,"summary":"Hot"},{"date":"2019-08-
27T08:16:42.178201+01:00","temperatureC":-9,"temperatureF":16,"summary":"Hot"},{"date":"2019-08-
28T08:16:42.178201+01:00","temperatureC":8,"temperatureF":46,"summary":"Mild"},{"date":"2019-08-
29T08:16:42.178202+01:00","temperatureC":-1,"temperatureF":31,"summary":"Balmy"},{"date":"2019-08-
30T08:16:42.178202+01:00","temperatureC":-17,"temperatureF":2,"summary":"Sweltering"},{"date":"2019-08-
31T08:16:42.178203+01:00","temperatureC":35,"temperatureF":94,"summary":"Sweltering"}]

Рис. 18.2. Двухнедельный прогноз погоды в формате JSON

6. Закройте браузер Google Chrome.
7. На панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C, чтобы остановить консольное приложение и завершить работу веб-сервера Kestrel, на котором размещен сервис Web API в ASP.NET Core.

Создание веб-сервиса для базы данных Northwind

В отличие от контроллеров ASP.NET Core MVC контроллеры Web API не вызывают представления Razor для возврата пользователям HTML-ответов в их браузеры. Вместо этого они согласовывают контент с клиентским приложением, производящим HTTP-запрос, для возврата в HTTP-ответе данных в таких форматах, как XML, JSON или X-WWW-FORM-URLENCODED.

Затем клиентское приложение должно десериализовать данные из согласованного формата. В современных сервисах наиболее широко используется формат *JavaScript Object Notation (JSON)*, поскольку он достаточно компактный и отлично поддерживает JavaScript в браузере при создании *одностраничных приложений* (Single Page Applications, SPA) такими технологиями на стороне клиента, как Angular, React и Vue.

Мы будем ссылаться на сущностную модель данных Entity Framework Core для базы данных Northwind, созданную в главе 14.

1. В проекте `NorthwindService` найдите и откройте файл `NorthwindService.csproj`.
2. Добавьте ссылку на проект в `NorthwindContextLib`:

```
<ItemGroup>
  <ProjectReference Include=
    "..\NorthwindContextLib\NorthwindContextLib.csproj" />
</ItemGroup>
```

3. На панели TERMINAL (Терминал) введите следующую команду и убедитесь, что проект компилируется:

```
dotnet build
```

4. Найдите и откройте файл `Startup.cs`. Измените его, чтобы импортировать пространства имен `System.IO`, `Microsoft.EntityFrameworkCore`, `Microsoft.AspNetCore.Mvc.Formatters` и `Packt.Shared`, а также статически импортировать класс `System.Console`.
5. Добавьте в метод `ConfigureServices` перед вызовом `AddControllers` операторы, чтобы настроить контекст данных `Northwind`:

```
string databasePath = Path.Combine("..", "Northwind.db");
services.AddDbContext<Northwind>(options =>
  options.UseSqlite($"Data Source={databasePath}"));
```

6. Добавьте операторы для вывода в консоль имен форматеров вывода по умолчанию и поддерживаемых ими медиаформатов, а затем добавьте средства форматирования сериализатора XML и установите совместимость с ASP.NET Core 3.0 после вызова метода для добавления поддержки контроллера, как показано ниже:

```
services.AddControllers(options =>
  {
    WriteLine("Default output formatters:");
    foreach(IOutputFormatter formatter in options.OutputFormatters)
    {
      var mediaFormatter = formatter as OutputFormatter;
      if (mediaFormatter == null)
      {
        WriteLine($" {formatter.GetType().Name}");
      }
      else // Класс OutputFormatter содержит SupportedMediaTypes
      {
        WriteLine(" {0}, Media types: {1}",
          arg0: mediaFormatter.GetType().Name,
          arg1: string.Join(", ",
            mediaFormatter.SupportedMediaTypes));
      }
    }
  })
  .AddXmlDataContractSerializerFormatters()
  .AddXmlSerializerFormatters()
  .SetCompatibilityVersion(CompatibilityVersion.Version_3_0);
```

Более подробную информацию о преимуществах настройки совместимости версий можно найти на сайте https://docs.microsoft.com/ru-ru/aspnet/core/mvc/compatibility-version.

7. Запустите веб-сервис и обратите внимание, что имеется четыре форматера вывода по умолчанию, в том числе те, которые преобразуют значения `null` в код состояния `204 No Content` (Нет контента), и те, которые поддерживают ответы в формате простого текста и JSON:

```
Default output formatters:
  HttpNoContentOutputFormatter
  StringOutputFormatter, Media types: text/plain
  StreamOutputFormatter
  SystemTextJsonOutputFormatter, Media types: application/json, text/json,
application/*+json
```

8. Остановите веб-сервис.

Создание хранилищ данных для сущностей

Для обеспечения операций CRUD отлично подходит определение и реализация хранилища данных (репозитория). Аббревиатура CRUD включает в себя следующие операции:

- **C** — создание (Create);
- **R** — извлечение (или чтение) (Retrieve, Read);
- **U** — обновление (Update);
- **D** — удаление (Delete).

Создадим хранилище данных для таблицы `Customers` в `Northwind`. В этой таблице содержится всего 91 клиент, поэтому в целях улучшения масштабируемости и производительности при чтении записей о клиентах мы будем хранить копию всей таблицы в памяти. В реальном веб-сервисе вам необходимо использовать распределенный кэш, например *Redis*, хранилище структур данных с открытым исходным кодом, которое можно применять в качестве высокопроизводительной базы данных с высокой доступностью, кэш или брокер сообщений.

Более подробно о Redis можно прочитать на сайте https://redis.io.

Будем следовать современным технологиям и сделаем API хранилища асинхронным. Мы будем использовать класс контроллера с инъекцией параметров через конструктор, поэтому создается новый экземпляр контроллера для обработки каждого HTTP-запроса.

1. Создайте в проекте `NorthwindService` папку Repositories.
2. В папку Repositories добавьте два файла классов с именами `ICustomerRepository.cs` и `CustomerRepository.cs`.

3. Интерфейс `ICustomerRepository` определит пять методов:

```
using Packt.Shared;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace NorthwindService.Repositories
{
  public interface ICustomerRepository
  {
    Task<Customer> CreateAsync(Customer c);
    Task<IEnumerable<Customer>> RetrieveAllAsync();
    Task<Customer> RetrieveAsync(string id);
    Task<Customer> UpdateAsync(string id, Customer c);
    Task<bool?> DeleteAsync(string id);
  }
}
```

4. Класс `CustomerRepository` будет реализовывать эти пять методов, как показано ниже:

```
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Packt.Shared;
using System.Collections.Generic;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading.Tasks;

namespace NorthwindService.Repositories
{
  public class CustomerRepository : ICustomerRepository
  {
    // использование статического потокобезопасного словарного
    // поля для кэширования клиентов
    private static ConcurrentDictionary
      <string, Customer> customersCache;

    // использование поля экземпляра класса для контекста,
    // поскольку он не должен кэшироваться из-за своего
    // внутреннего кэширования
    private Northwind db;

    public CustomerRepository(Northwind db)
    {
      this.db = db;

      // предварительная загрузка клиентов из базы данных в обычный
      // словарь с CustomerID в качестве ключа, а затем
      // преобразование в потокобезопасный ConcurrentDictionary
      if (customersCache == null)
      {
        customersCache = new ConcurrentDictionary<string, Customer>(
          db.Customers.ToDictionary(c => c.CustomerID));
```

```
    }
  }

  public async Task<Customer> CreateAsync(Customer c)
  {
    // нормализация CustomerID в верхнем регистре
    c.CustomerID = c.CustomerID.ToUpper();

    // добавление в базу данных с помощью EF Core
    EntityEntry<Customer> added = await db.Customers.AddAsync(c);
    int affected = await db.SaveChangesAsync();
    if (affected == 1)
    {
      // если клиент новый, то добавление его в кэш, иначе
      // произойдет вызов метода UpdateCache
      return customersCache.AddOrUpdate(c.CustomerID, c, UpdateCache);
    }
    else
    {
      return null;
    }
  }

  public Task<IEnumerable<Customer>> RetrieveAllAsync()
  {
    // извлечение из кэша для производительности
    return Task.Run<IEnumerable<Customer>>(
       () => customersCache.Values);
  }

  public Task<Customer> RetrieveAsync(string id)
  {
    return Task.Run(() =>
    {
      // извлечение из кэша для производительности
      id = id.ToUpper();
      customersCache.TryGetValue(id, out Customer c);
      return c;
    });
  }

  private Customer UpdateCache(string id, Customer c)
  {
    Customer old;
    if (customersCache.TryGetValue(id, out old))
    {
      if (customersCache.TryUpdate(id, c, old))
      {
        return c;
      }
    }
    return null;
  }
```

```
    public async Task<Customer> UpdateAsync(string id, Customer c)
    {
      // нормализация ID клиента
      id = id.ToUpper();
      c.CustomerID = c.CustomerID.ToUpper();

      // обновление в базе данных
      db.Customers.Update(c);
      int affected = await db.SaveChangesAsync();
      if (affected == 1)
      {
        // обновление в кэше
        return UpdateCache(id, c);
      }
      return null;
    }

    public async Task<bool?> DeleteAsync(string id)
    {
      id = id.ToUpper();

      // удаление из базы данных
      Customer c = db.Customers.Find(id);
      db.Customers.Remove(c);
      int affected = await db.SaveChangesAsync();
      if (affected == 1)
      {
        // удаление из кэша
        return customersCache.TryRemove(id, out c);
      }
      else
      {
        return null;
      }
    }
  }
}
```

Реализация контроллера Web API

Существует несколько полезных атрибутов и методов для реализации контроллера, возвращающего данные вместо HTML.

В контроллерах MVC такой путь, как `/home/index/`, сообщает нам имя класса контроллера и имя метода действия, например класс `HomeController` и метод действия `Index`.

В контроллерах Web API такой путь, как `/weatherforecast/`, дает только указание использовать имя класса контроллера, например `WeatherForecastController`.

Чтобы определить имя метода действия для выполнения, мы должны отобразить методы HTTP, такие как `GET` и `POST`, в методы в класса контроллера.

Чтобы указать методы HTTP, на которые должен даваться ответ, необходимо пометить методы контроллера следующими атрибутами:

- методы действий `[HttpGet]`, `[HttpHead]` отвечают на HTTP-запросы `GET` или `HEAD`, чтобы получить ресурс и вернуть либо ресурс и его заголовки ответа, либо только заголовки;
- метод действия `[HttpPost]` отвечает на HTTP-запросы `POST` для создания нового ресурса;
- методы действий `[HttpPut]`, `[HttpPatch]` отвечают на HTTP-запросы `PUT` или `PATCH`, чтобы обновить существующий ресурс, либо заменив его, либо обновив некоторые его свойства;
- метод действия `[HttpDelete]` отвечает на HTTP-запросы `DELETE` для удаления ресурса;
- метод действия `[HttpOptions]` отвечает на HTTP-запросы `OPTIONS`.

Более подробную информацию о методе HTTP OPTIONS и других методах можно прочитать на сайте https://developer.mozilla.org/ru-ru/docs/Web/HTTP/Methods/OPTIONS.

Метод действия может возвращать типы .NET, такие как одно строковое значение, сложные объекты — классы или структуры — или коллекции сложных объектов. Web API в ASP.NET Core автоматически сериализует их в запрошенный формат данных, заданный в HTTP-запросе заголовоком `Accept`, например JSON, если был зарегистрирован подходящий сериализатор.

Осуществлять больший контроль над ответом помогают вспомогательные методы, которые возвращают оболочку `ActionResult` вокруг типа .NET.

Объявите тип возвращаемого значения метода действия как `IActionResult`, если он может возвращать разные типы на основе входных данных или других переменных. Объявите тип возвращаемого значения метода действия как `ActionResult<T>`, если он вернет только один тип, но с другими кодами состояния.

Дополните методы действий атрибутом [ProducesResponseType], чтобы указать все известные типы и коды состояния HTTP, которые клиент должен ожидать в ответе. Затем эту информацию можно публично предоставить для документирования сведений о взаимодействии клиента с вашим веб-сервисом. Это будет частью вашей официальной документации. Далее в главе вы узнаете, как можно установить анализатор кода, чтобы получать предупреждения, если вы не дополняете свои методы действий.

Например, метод действия, который получает товар на основе параметра `id`, будет дополнен тремя атрибутами: один для обозначения того, что он отвечает на запросы `GET` и имеет параметр `id`, и два для обозначения того, что происходит при успешном выполнении и при предоставлении клиентом неверного идентификатора товара, как показано ниже:

```
[HttpGet("{id}")]
[ProducesResponseType(200, Type = typeof(Product))]
[ProducesResponseType(404)]
public IActionResult Get(string id)
```

Класс `ControllerBase` содержит методы, позволяющие легко возвращать разные ответы:

- `Ok` возвращает код состояния HTTP `200` с ресурсом, преобразованным в предпочтительный для клиента формат, такой как JSON или XML. Обычно используется в ответ на HTTP-запрос `GET`;
- `CreatedAtRoute` возвращает код состояния HTTP `201` с указанием пути к новому ресурсу. Обычно используется в ответ на запрос `POST` для создания ресурса, который можно быстро выполнить;
- `Accepted` возвращает код состояния HTTP `202` для обозначения, что запрос обрабатывается, но не завершен. Обычно используется в ответ на запрос, запускающий фоновый процесс, выполнение которого занимает много времени;
- `NoContentResult` возвращает код состояния HTTP `204`. Обычно используется в ответ на запрос `PUT` для обновления существующего ресурса, и ответ не должен содержать обновленный ресурс;
- `BadRequest` возвращает код состояния HTTP `400` с необязательной строкой сообщения;
- `NotFound` возвращает код состояния HTTP `404` с автоматически заполненным телом `ProblemDetails` (требуется версия совместимости 2.2 или более поздняя).

Настройка хранилища данных клиентов и контроллера Web API

Настроим хранилище так, чтобы оно могло вызываться из контроллера Web API.

При запуске веб-сервиса вы зарегистрируете реализацию сервиса как зависимость с заданной областью (scoped) для хранилища данных, а затем воспользуетесь внедрением параметров конструктора, чтобы получить его в новом контроллере Web API для работы с клиентами.

Более подробную информацию о внедрении зависимостей можно найти на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/dependency-injection.

Чтобы показать пример различия между контроллерами MVC и Web API с помощью маршрутов, мы будем применять общеизвестное соглашение о добавлении к URL префикса `/api` для контроллера `customers`.

1. Найдите файл `Startup.cs`, откройте его и импортируйте пространство имен `NorthwindService.Repositories`.
2. Добавьте в конец метода `ConfigureServices` оператор, который зарегистрирует `CustomerRepository` для использования во время выполнения, как показано ниже:

```
services.AddScoped<ICustomerRepository, CustomerRepository>();
```

3. В папку Controllers добавьте новый класс `CustomersController.cs`.
4. Для работы с клиентами добавьте в класс `CustomersController` операторы для определения класса контроллера Web API:

```
using Microsoft.AspNetCore.Mvc;
using Packt.Shared;
using NorthwindService.Repositories;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace NorthwindService.Controllers
{
  // базовый адрес: api/customers
  [Route("api/[controller]")]
  [ApiController]
  public class CustomersController : ControllerBase
  {
    private ICustomerRepository repo;

    // конструктор внедряет хранилище, зарегистрированное в Startup
    public CustomersController(ICustomerRepository repo)
    {
      this.repo = repo;
    }

    // GET: api/customers
    // GET: api/customers/?country=[country]
    // всегда будет возвращать список клиентов, даже если он пуст
    [HttpGet]
    [ProducesResponseType(200,
      Type = typeof(IEnumerable<Customer>))]
    public async Task<IEnumerable<Customer>> GetCustomers(
      string country)
    {
      if (string.IsNullOrWhiteSpace(country))
      {
```

```
        return await repo.RetrieveAllAsync();
      }
      else
      {
        return (await repo.RetrieveAllAsync())
          .Where(customer => customer.Country == country);
      }
    }

    // GET: api/customers/[id]
    [HttpGet("{id}", Name = nameof(GetCustomer))] // именованный маршрут
    [ProducesResponseType(200, Type = typeof(Customer))]
    [ProducesResponseType(404)]
    public async Task<IActionResult> GetCustomer(string id)
    {
      Customer c = await repo.RetrieveAsync(id);
      if (c == null)
      {
        return NotFound(); // 404 Ресурс не обнаружен
      }
      return Ok(c); // 200 Возвращает OK с клиентом в теле ответа
    }

    // POST: api/customers
    // BODY: Customer (JSON, XML)
    [HttpPost]
    [ProducesResponseType(201, Type = typeof(Customer))]
    [ProducesResponseType(400)]
    public async Task<IActionResult> Create([FromBody] Customer c)
    {
      if (c == null)
      {
        return BadRequest(); // 400 Неверный запрос
      }
      if (!ModelState.IsValid)
      {
        return BadRequest(ModelState); // 400 Неверный запрос
      }
      Customer added = await repo.CreateAsync(c);
      return CreatedAtRoute( // 201 Создано
        routeName: nameof(GetCustomer),
        routeValues: new { id = added.CustomerID.ToLower() },
        value: added);
    }

    // PUT: api/customers/[id]
    // BODY: Customer (JSON, XML)
    [HttpPut("{id}")]
    [ProducesResponseType(204)]
    [ProducesResponseType(400)]
    [ProducesResponseType(404)]
```

```
    public async Task<IActionResult> Update(
      string id, [FromBody] Customer c)
    {
      id = id.ToUpper();
      c.CustomerID = c.CustomerID.ToUpper();

      if (c == null || c.CustomerID != id)
      {
        return BadRequest(); // 400 Неверный запрос
      }
      if (!ModelState.IsValid)
      {
        return BadRequest(ModelState); // 400 Неверный запрос
      }

      var existing = await repo.RetrieveAsync(id);
      if (existing == null)
      {
        return NotFound(); // 404 Ресурс не обнаружен
      }
      await repo.UpdateAsync(id, c);
      return new NoContentResult(); // 204 Нет контента
    }

    // DELETE: api/customers/[id]
    [HttpDelete("{id}")]
    [ProducesResponseType(204)]
    [ProducesResponseType(400)]
    [ProducesResponseType(404)]
    public async Task<IActionResult> Delete(string id)
    {
      var existing = await repo.RetrieveAsync(id);
      if (existing == null)
      {
        return NotFound(); // 404 Ресурс не обнаружен
      }

      bool? deleted = await repo.DeleteAsync(id);
      if (deleted.HasValue && deleted.Value) // короткозамкнутая AND
      {
        return new NoContentResult(); // 204 Нет контента
      }
      else
      {
        return BadRequest( // 400 Неверный запрос
          $"Customer {id} was found but failed to delete.");
      }
    }
  }
}
```

При анализе класса `Controller` обратите внимание на следующие моменты.

- Класс `Controller` регистрирует маршрут, имя которого начинается с `api/`, и включает имя контроллера, то есть `api/customer`.
- В конструкторе используется внедрение зависимостей для получения экземпляра зарегистрированного хранилища для клиентов.
- Существует пять методов выполнения операций CRUD для клиентов: два `GET` (все клиенты или один), `POST` (создание), `PUT` (обновление) и `DELETE` (удаление).
- В метод `GetCustomers` может передаваться строковый параметр с названием страны. Если такой параметр отсутствует, то в результате возвращаются все клиенты. Если присутствует, то используется для фильтрации клиентов по странам.
- Метод `GetCustomer` содержит маршрут с явным названием `GetCustomer`. Его можно использовать для создания URL после добавления нового клиента.
- Метод `Create` дополняет параметр `customer` атрибутом `[FromBody]`, чтобы дать привязке модели указание заполнить его значениями из тела HTTP-запроса `POST`.
- Метод `Create` возвращает ответ, который использует маршрут `GetCustomer`, чтобы клиент знал, как в дальнейшем получить вновь созданный ресурс. Мы сопоставляем эти методы, чтобы создать, а затем найти клиента.
- Методы `Create` и `Update` проверяют состояние модели клиента, переданного в теле HTTP-запроса, и при неверном запросе возвращают сообщение `400 Bad Request`, содержащее сведения об ошибках проверки модели.

Получая HTTP-запрос, сервис создает экземпляр класса контроллера, вызывает соответствующий метод действия, возвращает ответ в предпочитаемом клиентом формате и освобождает используемые контроллером ресурсы, включая хранилище и его контекст данных.

Спецификация деталей проблемы

Функция, добавленная в ASP.NET Core 2.1 и более поздних версиях, представляет собой реализацию веб-стандарта для спецификации деталей проблемы.

Более подробную информацию о предлагаемом стандарте можно найти на сайте https://tools.ietf.org/html/rfc7807.

В контроллерах, помеченных атрибутом `[ApiController]` в проекте с включенной совместимостью ASP.NET Core 2.2 или более поздней версии, методы действия, возвращающие параметр `IActionResult`, который возвращает клиентский код состояния (то есть `4xx`), будут автоматически включать в тело ответа сериализованный экземпляр класса `ProblemDetails`.

Подробне о реализации деталей проблем можно узнать на сайте https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.aspnetcore.mvc.problemdetails.

Если вы хотите взять на себя управление, то можете самостоятельно создать экземпляр `ProblemDetails` и включить дополнительную информацию.

Создадим неверный запрос, который требует возврата клиенту нестандартных данных.

1. В начале класса `CustomersController` импортируйте пространство имен `Microsoft.AspNetCore.Http`.
2. В начало метода `Delete` добавьте операторы для проверки того, соответствует ли идентификатор строковому значению `"bad"`, и если это так, то верните собственный объект сведений о проблеме:

```
// контроль проблемы
if (id == "bad")
{
  var problemDetails = new ProblemDetails
  {
    Status = StatusCodes.Status400BadRequest,
    Type = "https://localhost:5001/customers/failed-to-delete",
    Title = $"Customer ID {id} found but failed to delete.",
    Detail = "More details like Company Name, Country and so on.",
    Instance = HttpContext.Request.Path
  };
  return BadRequest(problemDetails); // 400 неверный запрос
}
```

Управление сериализацией XML

В файл Startup.cs добавим XmlSerializer, чтобы наша служба веб-API могла возвращать XML, а также при необходимости данные в формате JSON.

Однако XmlSerializer не может сериализовать интерфейсы. Для определения связанных дочерних сущностей наши классы сущностей используют `ICollection<T>`. В процессе выполнения это вызовет предупреждение, например, для класса `Customer` и его свойства `Orders`, как показано в следующих выходных данных:

```
warn: Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerOutputFormatter[1]
      An error occurred while trying to create an XmlSerializer for the type
      'Packt.Shared.Customer'.
      System.InvalidOperationException: There was an error reflecting type
      'Packt.Shared.Customer'.
      ---> System.InvalidOperationException: Cannot serialize member 'Packt.
      Shared.Customer.Orders' of type 'System.Collections.Generic.
```

```
ICollection`1[[Packt.
Shared.Order, NorthwindEntitiesLib, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null]]', see inner exception for more details.
```

Однако предупреждение можно предотвратить, исключив свойство `Orders` при сериализации клиента в XML.

1. В проекте NorthwindEntitiesLib найдите и откройте файл `Customers.cs`.
2. Импортируйте пространство имен `System.Xml.Serialization`.
3. Дополните свойство `Orders` атрибутом, чтобы игнорировать его при сериализации:

```
[InverseProperty(nameof(Order.Customer))]
[XmlIgnore]
public virtual ICollection<Order> Orders { get; set; }
```

Документирование и тестирование веб-сервисов

Вы можете легко протестировать веб-сервис, отправляя в браузере HTTP-запросы `GET`. Чтобы проверить другие HTTP-методы, потребуются более усовершенствованные инструменты.

Тестирование GET-запросов в браузерах

Вы будете применять браузер Google Chrome, чтобы тестировать три реализации запроса `GET`: для всех клиентов, для клиентов в указанной стране и для одного клиента, используя его уникальный идентификатор.

1. На панели TERMINAL (Терминал) выполните следующую команду, чтобы запустить веб-сервис `NorthwindService`:

```
dotnet run
```

2. Откройте окно браузера Google Chrome и в адресной строке укажите следующий URL: http://localhost:5001/api/customers. Вы должны увидеть возвращенный JSON-файл, содержащий список всех клиентов (91 строку) в базе данных `Northwind` (рис. 18.3).
3. В адресной строке браузера Google Chrome укажите следующий URL: localhost:5001/api/customers/?country=Germany. Вы должны увидеть возвращенный JSON-файл, содержащий только клиентов из Германии (рис. 18.4).

   Если вы получите пустой массив, то убедитесь, что ввели название страны, используя правильный регистр, поскольку запрос к базе данных чувствителен к регистру.

https://localhost:5001/api/customers

[{"customerID":"LAZYK","companyName":"Lazy K Kountry Store","contactName":"John Steel","contactTitle":"Marketing Manager","address":"12 Orchestra Terrace","city":"Walla Walla","region":"WA","postalCode":"99362","country":"USA","phone":"(509) 555-7969","fax":"(509) 555-6221","orders":null},{"customerID":"BERGS","companyName":"Berglunds snabbköp","contactName":"Christina Berglund","contactTitle":"Order Administrator","address":"Berguvsvägen 8","city":"Luleå","region":null,"postalCode":"S-958 22","country":"Sweden","phone":"0921-12 34 65","fax":"0921-12 34 67","orders":null},{"customerID":"LILAS","companyName":"LILA-Supermercado","contactName":"Carlos González","contactTitle":"Accounting Manager","address":"Carrera 52 con Ave. Bolívar #65-98 Llano Largo","city":"Barquisimeto","region":"Lara","postalCode":"3508","country":"Venezuela","phone":"(9) 331-6954","fax":"(9) 331-7256","orders":null},{"customerID":"TORTU","companyName":"Tortuga Restaurante","contactName":"Miguel Angel Paolino","contactTitle":"Owner","address":"Avda. Azteca 123","city":"México D.F.","region":null,"postalCode":"05033","country":"Mexico","phone":"(5) 555-2933","fax":null,"orders":null},

Рис. 18.3. Список клиентов из базы данных Northwind в формате JSON

https://localhost:5001/api/customers/?country=Germany

[{"customerID":"WANDK","companyName":"Die Wandernde Kuh","contactName":"Rita Müller","contactTitle":"Sales Representative","address":"Adenauerallee 900","city":"Stuttgart","region":null,"postalCode":"70563","country":"Germany","phone":"0711-020361","fax":"0711-035428","orders":null},{"customerID":"MORGK","companyName":"Morgenstern Gesundkost","contactName":"Alexander Feuer","contactTitle":"Marketing Assistant","address":"Heerstr. 22","city":"Leipzig","region":null,"postalCode":"04179","country":"Germany","phone":"0342-023176","fax":null,"orders":null},{"customerID":"BLAUS","companyName":"Blauer See Delikatessen","contactName":"Hanna Moos","contactTitle":"Sales Representative","address":"Forsterstr. 57","city":"Mannheim","region":null,"postalCode":"68306","country":"Germany","phone":"0621-08460","fax":"0621-08924","orders":null},{"customerID":"KOENE","companyName":"Königlich Essen","contactName":"Philip Cramer","contactTitle":"Sales Associate","address":"Maubelstr.

Рис. 18.4. Список клиентов, проживающих в Германии, в формате JSON

4. В адресной строке браузера Google Chrome укажите следующий URL: localhost:5001/api/customers/alfki. Вы должны увидеть возвращенный JSON-файл, содержащий только одну строку — клиента с именем Alfreds Futterkiste (рис. 18.5).

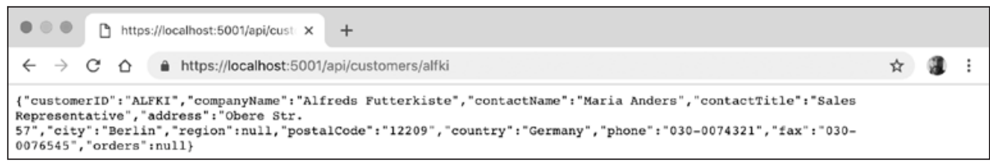


Рис. 18.5. Конкретная информация о клиенте в формате JSON

Вы не должны беспокоиться о регистре для значения идентификатора клиента, поскольку внутри класса `Controller` мы в программном коде нормализовали строковое значение в верхнем регистре.

Однако как мы можем проверить другие методы HTTP, такие как `POST`, `PUT` и `DELETE`? И как задокументировать наш веб-сервис, чтобы каждому было легко понять, как взаимодействовать с ним?

Чтобы решить первую проблему, мы можем установить расширение Visual Studio Code под названием *REST Client*. Чтобы решить вторую — можем включить *Swagger*, самую популярную технологию для документирования и тестирования HTTP API. Однако в первую очередь рассмотрим возможности расширения Visual Studio Code.

Тестирование HTTP-запросов с помощью расширения REST Client

REST Client позволяет отправлять HTTP-запрос и просматривать ответ непосредственно в Visual Studio Code.

Более подробную информацию об использовании REST Client можно получить на сайте https://github.com/Huachao/vscode-restclient/blob/master/README.md.

1. Если вы еще не установили REST Client от Huachao Mao (`humao.rest-client`), то сделайте это сейчас.
2. В программе Visual Studio Code откройте проект `NorthwindService`.
3. Запустите веб-сервис, если он не запущен, введя на панели TERMINAL (Терминал) следующую команду: `dotnet run`.
4. В папке NorthwindService создайте папку RestClientTest.
5. В папке RestClientTests создайте файл `get-customers.http` и отредактируйте его код, чтобы в нем содержался HTTP-запрос `GET` для получения всех клиентов, как показано ниже:

```
GET https://localhost:5001/api/customers/ HTTP/1.1
```

6. Выберите View ▸ Command Palette (Вид ▸ Панель команд), введите `rest client`, выберите команду Rest Client: Send Request и нажмите клавишу Enter (рис. 18.6).

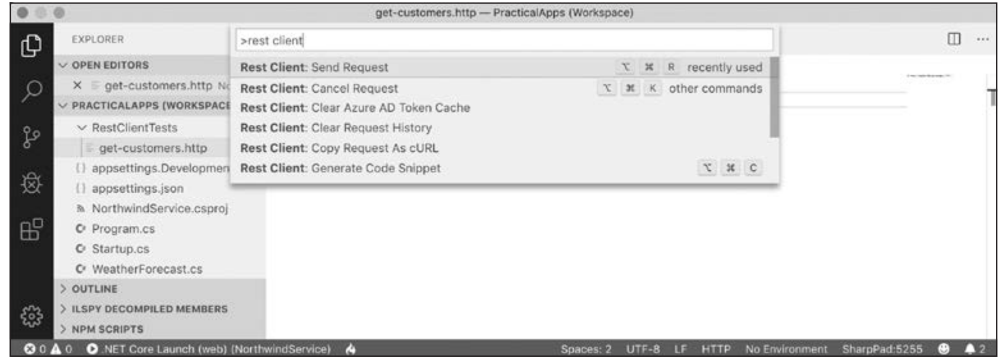


Рис. 18.6. Тестирование HTTP-запросов с помощью Rest Client

7. Обратите внимание: вкладка Response (Ответ) отображается в новой области окна с вкладками, расположенными по вертикали, и вы можете перераспределить открытые вкладки по горизонтали, перетаскивая их.

8. Введите больше HTTP-запросов `GET`, каждый из которых разделен тремя хеш-символами, чтобы проверить наличие клиентов в разных странах и получение информации об одном клиенте с помощью его идентификатора ID, как показано ниже:

```
###
GET https://localhost:5001/api/customers/?country=Germany HTTP/1.1
###
GET https://localhost:5001/api/customers/?country=USA HTTP/1.1
Accept: application/xml
###
GET https://localhost:5001/api/customers/ALFKI HTTP/1.1
###
GET https://localhost:5001/api/customers/abcxy HTTP/1.1
```

9. Для отправки запроса щелкните кнопкой мыши внутри каждого оператора и нажмите сочетание клавиш Ctrl+Alt+R или Cmd+Alt+R либо нажмите ссылку Send Request (Отправить запрос), расположенную над каждым запросом (рис. 18.7).

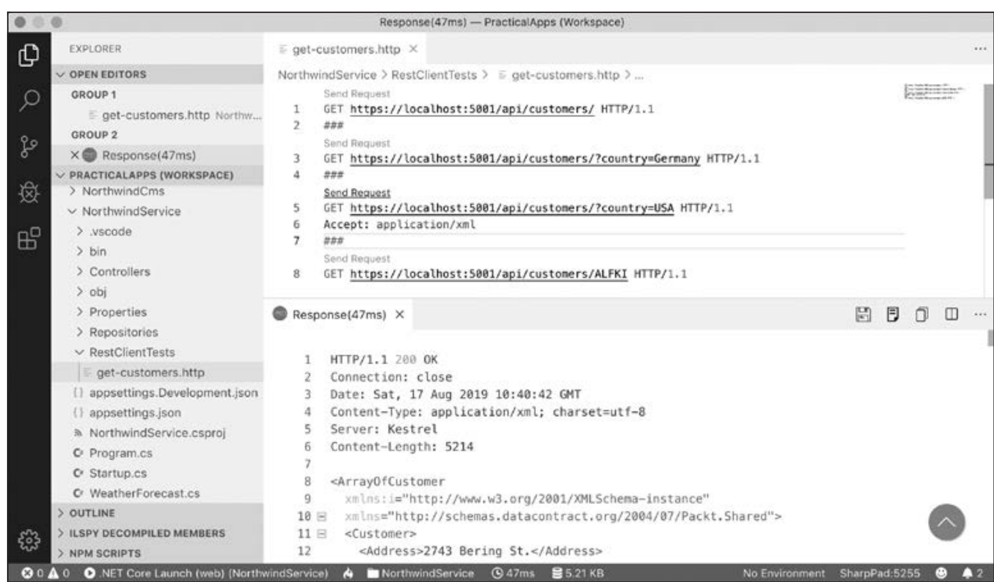


Рис. 18.7. Отправка запроса и получение ответа с помощью Rest Client

10. Создайте файл `create-customer.http` и отредактируйте его код, чтобы определить запрос `POST` для создания нового клиента:

```
POST https://localhost:5001/api/customers/ HTTP/1.1
Content-Type: application/json
Content-Length: 287
```

```
{
  "customerID": "ABCXY",
  "companyName": "ABC Corp",
  "contactName": "John Smith",
  "contactTitle": "Sir",
  "address": "Main Street",
  "city": "New York",
  "region": "NY",
  "postalCode": "90210",
  "country": "USA",
  "phone": "(123) 555-1234",
  "fax": null,
  "orders": null
}
```

Обратите внимание: REST Client будет отображать подсказки IntelliSense, если вы вводите обычные HTTP-запросы.

Из-за разных окончаний строк в разных операционных системах значение заголовка `Content-Length` будет отличаться в зависимости от операционных систем Windows и macOS или Linux. Если значение неверно, то запрос не будет выполнен.

11. Чтобы определить правильную длину контента, выделите тело запроса, а затем найдите в строке состояния количество символов (рис. 18.8).

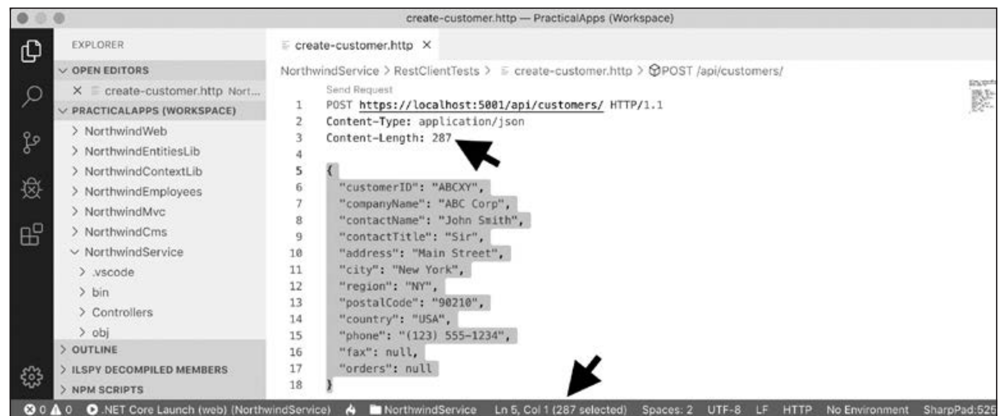


Рис. 18.8. Проверка правильности длины контента

12. Отправьте запрос и обратите внимание, что в ответе возвращается код состояния `201 Created`. Обратите также внимание, что местоположение (то есть URL-адрес) вновь созданного клиента — https://localhost:5001/api/Customers/abcxy (рис. 18.9).

Более подробную информацию об HTTP-запросах POST можно получить на сайте https://developer.mozilla.org/ru-ru/docs/Web/HTTP/Methods/POST.

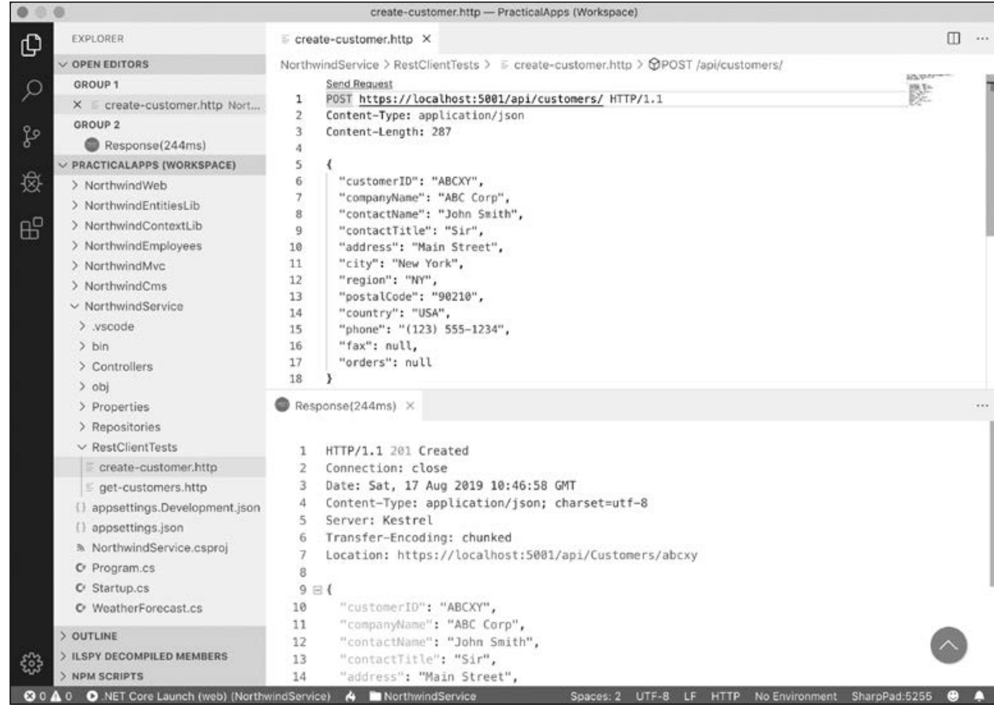


Рис. 18.9. Добавление нового клиента

В качестве дополнительной задачи я оставлю читателю создание файлов клиента REST для проверки обновления клиента (с помощью `PUT`) и удаление клиента (с помощью `DELETE`). Протестируйте их как на существующих, так и на несуществующих клиентах. Решения, касающиеся этой книги, находятся в репозитории GitHub.

Теперь, когда мы изучили быстрый и простой способ тестирования нашего сервиса, который также считается отличным способом изучения HTTP, как насчет внешних разработчиков? Мы хотим, чтобы им было максимально просто узнать, как работать с нашим сервисом, а затем вызвать его. Для этой цели мы будем использовать Swagger.

Swagger

Самая важная часть Swagger — это спецификация OpenAPI, которая определяет контракт в стиле REST для вашего API, детализируя все его ресурсы и операции в удобном для пользователя и машиночитаемом формате, что упрощает разработку, обнаружение и интеграцию.

Еще одна полезная для нас функция — *Swagger UI*, автоматически генерирующая документацию для вашего API со встроенными возможностями визуального тестирования.

Узнать больше о технологии Swagger можно на сайте https://swagger.io/.

Добавим библиотеку Swagger для нашего веб-сервиса, используя пакет `Swashbuckle`.

1. Если веб-служба запущена, остановите ее, нажав сочетание клавиш Ctrl+C на панели TERMINAL (Терминал).
2. Откройте проект `NorthwindService.csproj`. Добавьте ссылку на пакет для `Swashbuckle.AspNetCore`, как показано ниже (выделено полужирным шрифтом):

```
<ItemGroup>
  <PackageReference Include="Swashbuckle.AspNetCore" Version="5.5.1" />
</ItemGroup>
```

3. Найдите и откройте файл `Startup.cs`. Импортируйте пространства имен Swashbuckle `Swagger` и `SwaggerUI` и пространство имен моделей Microsoft OpenAPI:

```
using Swashbuckle.AspNetCore.Swagger;
using Swashbuckle.AspNetCore.SwaggerUI;
using Microsoft.OpenApi.Models;
```

4. Добавьте в метод `ConfigureServices` оператор для добавления поддержки Swagger, включая документацию для сервиса Northwind, указывающую, что это первая версия вашего сервиса:

```
// регистрация генератора Swagger
// и определение документа для сервиса Northwind
services.AddSwaggerGen(options =>
  {
    options.SwaggerDoc(name: "v1", info: new OpenApiInfo
      { Title = "Northwind Service API", Version = "v1" });
  });
```

Информацию о том, каким образом Swagger может поддерживать несколько версий API, можно получить на сайте https://stackoverflow.com/questions/30789045/leverage-multipleapiversions-in-swagger-with-attribute-versioning/30789944.

5. Добавьте в метод `Configure` операторы для использования `Swagger` и `Swagger UI`. Определите конечную точку для JSON-документа спецификации OpenAPI и перечислите методы HTTP, поддерживаемые нашим веб-сервисом, как показано ниже:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
  c.SwaggerEndpoint("/swagger/v1/swagger.json",
    "Northwind Service API Version 1");

  c.SupportedSubmitMethods(new[] {
    SubmitMethod.Get, SubmitMethod.Post,
    SubmitMethod.Put, SubmitMethod.Delete });
});
```

Тестирование запросов с помощью Swagger UI

Теперь вы готовы протестировать HTTP-запрос с помощью Swagger.

1. Запустите сервис Web API в ASP.NET под названием `NorthwindService`.
2. В адресной строке браузера Google Chrome введите следующий URL: https://localhost:5001/swagger/. Обратите внимание: были обнаружены и задокументированы контроллеры Web API Customer и WeatherForecast и используемые API схемы.
3. Выберите пункт GET/api/Customers/{id}, чтобы развернуть эту конечную точку. Вы увидите необходимый параметр для идентификатора клиента (рис. 18.10).

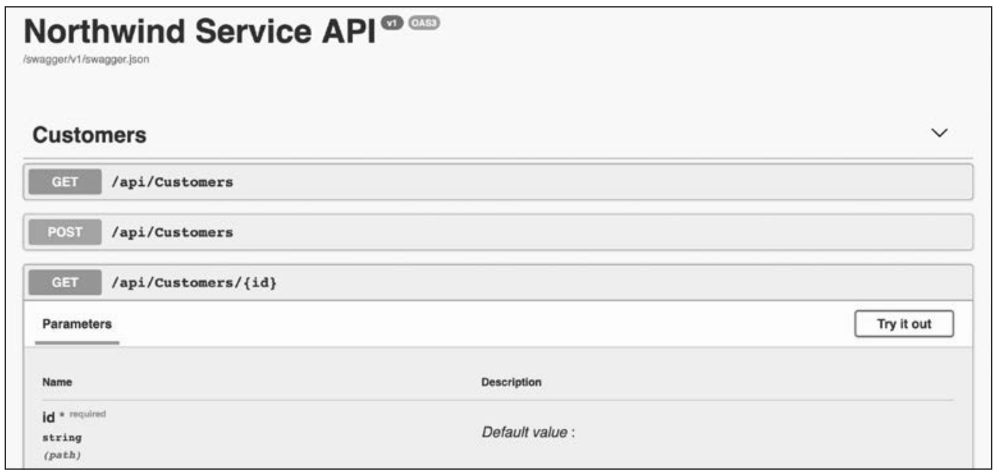


Рис. 18.10. Проверка параметров GET-запроса в Swagger

4. Нажмите кнопку Try it out (Попробовать), введите идентификатор ALFKI, а затем нажмите широкую синюю кнопку Execute (Выполнить) (рис. 18.11).

Рис. 18.11. Ввод идентификатора перед выполнением

5. Прокрутите вниз и обратите внимание на Request URL (URL запроса), Server response (Ответ сервера) с Code (Код) и Details (Сведения), включая Response body (Тело ответа) и Response headers (Заголовки ответа) (рис. 18.12).

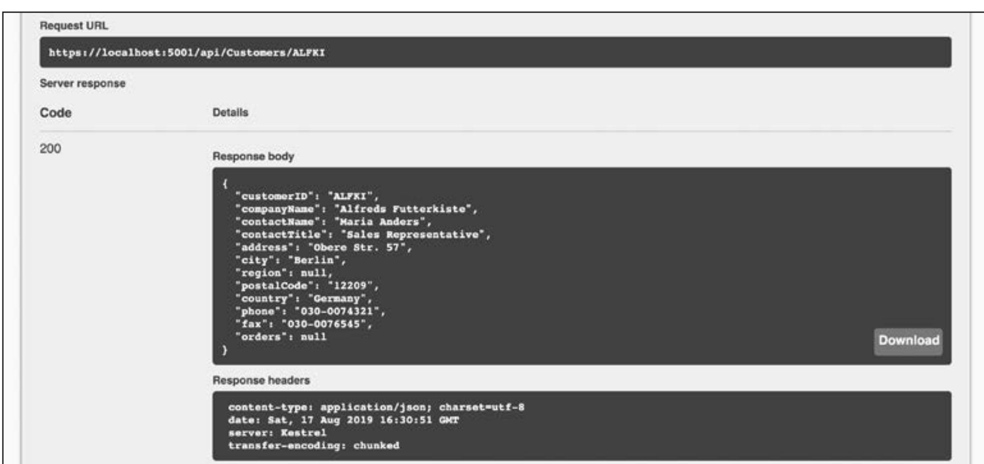


Рис. 18.12. Информация об ALFKI при успешном запросе Swagger

6. Прокрутите вверх, выберите `POST/api/Customers`, чтобы развернуть этот раздел, затем нажмите кнопку Try it out (Попробовать).
7. Щелкните кнопкой мыши внутри поля Request body (Тело запроса) и измените JSON-файл, чтобы определить нового клиента:

```
{
  "customerID": "SUPER",
  "companyName": "Super Company",
  "contactName": "Rasmus Ibensen",
```

```
  "contactTitle": "Sales Leader",
  "address": "Rotterslef 23",
  "city": "Billund",
  "region": null,
  "postalCode": "4371",
  "country": "Denmark",
  "phone": "31 21 43 21",
  "fax": "31 21 43 22",
  "orders": null
}
```

8. Нажмите кнопку Execute (Выполнить) и обратите внимание на Request URL (URL запроса), Server response (Ответ сервера) с Code (Код) и Details (Сведения), включая Response body (Тело ответа) и Response headers (Заголовки ответа) (рис. 18.13).

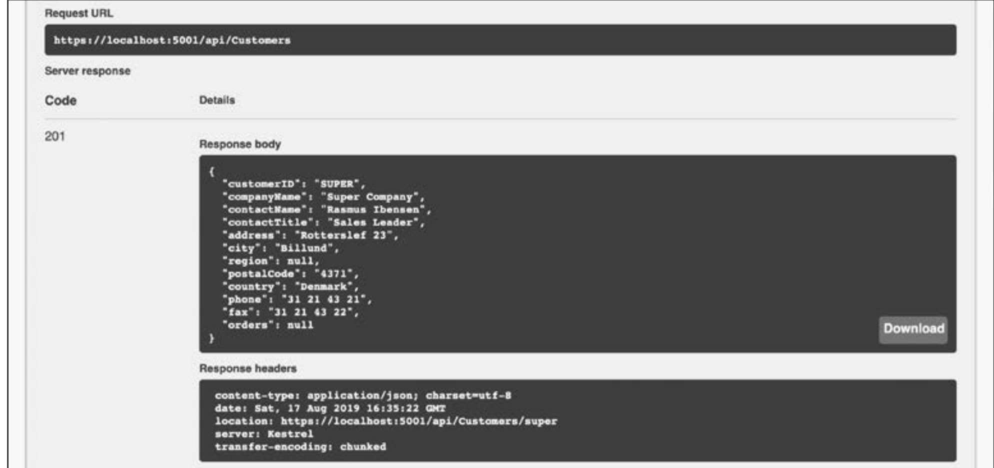


Рис. 18.13. Успешное добавление нового клиента

Код ответа `201` означает, что клиент был успешно создан.

9. Прокрутите вверх, выберите `GET/api/Customers`, нажмите кнопку Try it out (Попробовать), введите в качестве параметра название страны (`Denmark`) и нажмите кнопку Execute (Выполнить), чтобы подтвердить добавление нового клиента в базу данных (рис. 18.14).
10. Выберите `DELETE/api/Customers/{id}`, затем нажмите кнопку Try it out (Попробовать). Введите `super` для идентификатора `id` и нажмите кнопку Execute (Выполнить). Обратите внимание: Server response Code (код ответа сервера) — `204`, что указывает на успешное удаление (рис. 18.15).
11. Еще раз нажмите кнопку Execute (Выполнить). Обратите внимание: код ответа сервера `404` указывает на то, что клиент больше не существует, а в теле ответа содержится JSON-документ с подробным описанием проблемы (рис. 18.16).

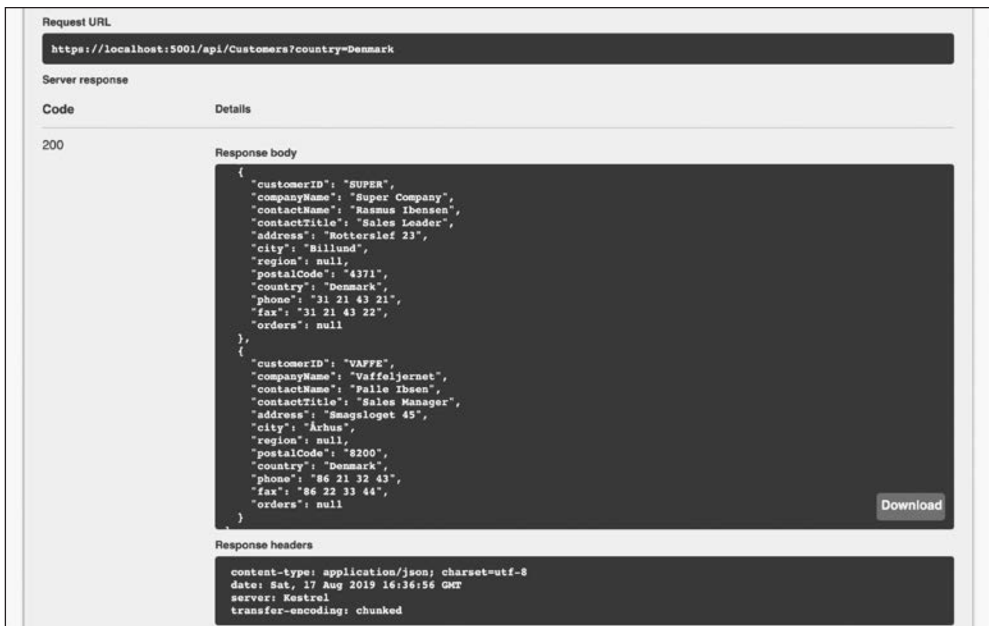


Рис. 18.14. Получение списка клиентов, проживающих в Дании

Рис. 18.15. Успешное удаление клиента

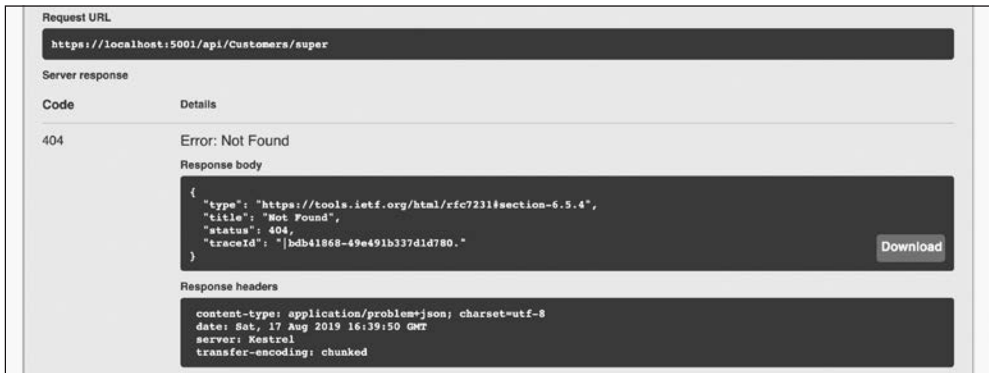


Рис. 18.16. JSON-документ

12. Введите `bad`, нажмите кнопку Execute (Выполнить). Обратите внимание: Server response Code (код ответа сервера) — `400` указывает на то, что клиент существует, но его не удалось удалить (в данном случае ввиду того, что веб-сервис имитирует эту ошибку), а тело ответа содержит JSON-документ с подробным описанием проблемы (рис. 18.17).

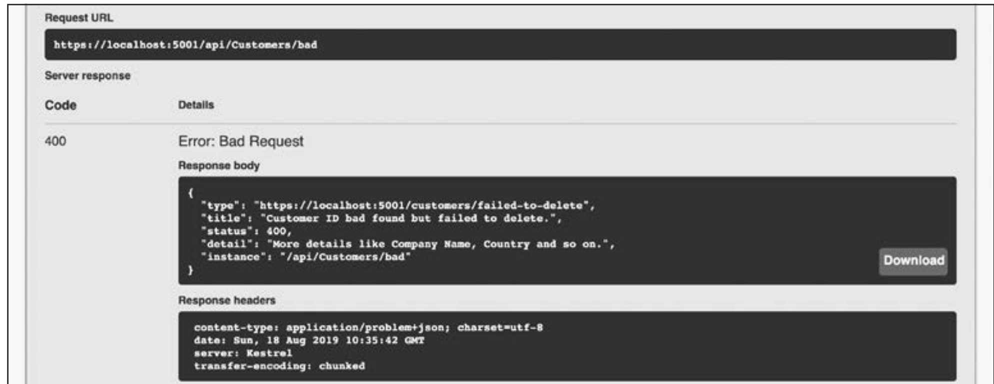


Рис. 18.17. Клиент действительно существует, но не может быть удален

13. Используйте методы GET, чтобы подтвердить удаление нового клиента из базы данных (первоначально в Дании было только два клиента).

    Я оставлю читателю тестирование обновления для существующего клиента с использованием метода PUT.

14. Закройте браузер Google Chrome.
15. На панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C, чтобы остановить консольное приложение и завершить работу веб-сервера Kestrel, на котором размещен ваш сервис.

Более подробно о важности документирования можно прочитать на сайте https://idratherbewriting.com/learnapidoc/.

Теперь вы готовы создавать приложения, обращающиеся к вашим веб-сервисам.

Обращение к сервисам с помощью HTTP-клиентов

Теперь, создав и протестировав сервис `Northwind`, мы узнаем, как вызывать его из любого приложения .NET Core, используя класс `HttpClient` и его новые настройки.

HttpClient

Самый простой способ обратиться к веб-сервису — это применить класс `HttpClient`. Однако многие люди используют его неправильно, поскольку он реализует интерфейс `IDisposable`, а даже документация Microsoft показывает его некорректное применение.

Обычно, когда тип реализует интерфейс `IDisposable`, вы должны создать его внутри оператора `using`, чтобы убедиться в его максимально быстром удалении. Класс `HttpClient` отличается тем, что является разделяемым, повторно используемым и частично безопасным для потоков.

В многопоточных сценариях с осторожностью следует использовать лишь свойства BaseAddress и DefaultRequestHeaders. Более подробную информацию и рекомендации можно получить на сайте https://medium.com/@nuno.caneco/c-httpclient-should-not-be-disposed-or-should-it-45d2a8f568bc.

Проблема связана с управлением базовыми сетевыми сокетами. Суть в том, что вам необходимо использовать один экземпляр сокета для каждой конечной точки HTTP, которую вы потребляете в течение эксплуатации вашего приложения.

Это позволит каждому экземпляру класса `HttpClient` установить значения по умолчанию, соответствующие конечной точке, с которой он работает, при этом эффективно управляя базовыми сетевыми сокетами.

В случае некорректного использования класса HttpClient и дестабилизации вашего программного обеспечения обратитесь за информацией по следующему адресу: https://aspnetmonsters.com/2016/08/2016-08-27-httpclientwrong/.

Настройка HTTP-клиентов с помощью HttpClientFactory

Корпорация Microsoft знает об этой проблеме, и в .NET Core 2.1 появился класс `HttpClientFactory` — фабрика, помогающая решить часть распространенных проблем, с которыми могут столкнуться разработчики. Таким образом Microsoft продвигает использование лучших приемов работы. Эту фабрику мы и будем использовать.

Более подробную информацию о том, как инициировать HTTP-запросы, можно найти на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/http-requests.

В следующем примере мы задействуем сайт MVC для Northwind в качестве клиента сервиса Web API для Northwind. Поскольку оба должны быть размещены на

веб-сервере одновременно, нам в первую очередь необходимо настроить их для использования разных номеров портов:

- Northwind Web API продолжит прослушивать порт 5001 с помощью HTTPS;
- Northwind MVC будет прослушивать порты 5000 и 5002 с помощью HTTP и HTTPS соответственно.

Настроим эти порты.

1. В проекте `NorthwindMvc` найдите и откройте файл `Program.cs`.
2. В метод `CreateHostBuilder` добавьте вызов метода расширения `UseUrls`, чтобы указать номер порта `5000` для HTTP и `5002` для HTTPS, как показано ниже (выделено полужирным шрифтом):

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
      webBuilder.UseStartup<Startup>();
      webBuilder.UseUrls(
        "http://localhost:5000",
        "https://localhost:5002"
      );
    });
```

3. Найдите и откройте файл `Startup.cs` и импортируйте пространство имен `System.Net.Http.Headers`.
4. В метод `ConfigureServices` добавьте оператор, чтобы разрешить `HttpClientFactory` с именованным клиентом выполнять вызовы в сервис Web API Northwind с помощью HTTPS через порт `5001` и запрашивать JSON в качестве формата ответа по умолчанию:

```
services.AddHttpClient(name: "NorthwindService",
  configureClient: options =>
  {
    options.BaseAddress = new Uri("https://localhost:5001/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
      "application/json", 1.0));
  });
```

Получение списка клиентов в контроллере в формате JSON

Теперь мы можем создать метод действия контроллера MVC, который использует фабрику для создания HTTP-клиента, делает запрос GET для клиентов и десериализует ответ JSON с помощью удобных методов расширения, представленных в .NET 5 в сборке System.Net.Http.Json и в пространстве имен.

Более подробную информацию о методах расширения HttpClient для работы с JSON можно найти на сайте https://github.com/dotnet/designs/blob/main/accepted/2020/json-http-extensions/json-http-extensions.md.

1. Найдите и откройте файл `Controllers/HomeController.cs` и импортируйте пространства имен `System.Net.Http` и `Newtonsoft.Json`.
2. Объявите поле для хранения фабрики HTTP-клиентов:

```
private readonly IHttpClientFactory clientFactory;
```

3. Установите поле в конструкторе:

```
public HomeController(
  ILogger<HomeController> logger,
  Northwind injectedContext,
  IHttpClientFactory httpClientFactory)
{
  _logger = logger;
  db = injectedContext;
  clientFactory = httpClientFactory;
}
```

4. Создайте метод действия для вызова сервиса Northwind, выборки всех клиентов и передачи их в представление:

```
public async Task<IActionResult> Customers(string country)
{
  string uri;
  if (string.IsNullOrEmpty(country))
  {
    ViewData["Title"] = "All Customers Worldwide";
    uri = "api/customers/";
  }
  else
  {
    ViewData["Title"] = $"Customers in {country}";
    uri = $"api/customers/?country={country}";
  }

  var client = clientFactory.CreateClient(
    name: "NorthwindService");

  var request = new HttpRequestMessage(
    method: HttpMethod.Get, requestUri: uri);

  HttpResponseMessage response = await client.SendAsync(request);

  var model = await response.Content
    .ReadFromJsonAsync<IEnumerable<Customer>>();

  return View(model);
}
```

5. В папке Views/Home создайте Razor-файл `Customers.cshtml`.
6. Для отображения клиентов отредактируйте файл шаблона Razor, как показано ниже:

```
@model IEnumerable<Packt.Shared.Customer>
<h2>@ViewData["Title"]</h2>
<table class="table">
  <thead>
    <tr>
      <th>Company Name</th>
      <th>Contact Name</th>
      <th>Address</th>
      <th>Phone</th>
    </tr>
  </thead>
  <tbody>
  @foreach (var item in Model)
  {
    <tr>
      <td>
        @Html.DisplayFor(modelItem => item.CompanyName)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.ContactName)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Address)
        @Html.DisplayFor(modelItem => item.City)
        @Html.DisplayFor(modelItem => item.Region)
        @Html.DisplayFor(modelItem => item.Country)
        @Html.DisplayFor(modelItem => item.PostalCode)
      </td>
      <td>
        @Html.DisplayFor(modelItem => item.Phone)
      </td>
    </tr>
  }
  </tbody>
</table>
```

7. Найдите и откройте файл `Views/Home/Index.cshtml` и добавьте форму после подсчета количества пользователей, позволяющую пользователям выбрать страну и увидеть клиентов:

```
<h3>Query customers from a service</h3>
<form asp-action="Customers" method="get">
  <input name="country" placeholder="Enter a country" />
  <input type="submit" />
</form>
```

Включение совместного использования ресурсов между источниками

Было бы полезно явно указать номер порта для `NorthwindService`, чтобы он не конфликтовал с используемыми по умолчанию портами `5000` для HTTP и `5002` для HTTPS, используемыми для таких сайтов, как `NorthwindMvc`, и включить *совместное использование ресурсов между источниками* (Cross-Origin Resource Sharing, CORS).

В целях повышения безопасности политика браузера по умолчанию для одного и того же источника не позволяет программному коду, загруженному из одного источника, получать доступ к ресурсам, загруженным из другого источника. CORS можно включить, чтобы разрешить запросы от указанных доменов. Узнать больше о CORS и ASP.NET Core можно на сайте https://docs.microsoft.com/ru-ru/aspnet/core/security/cors.

1. В проекте `NorthwindService` найдите и откройте файл `Program.cs`.
2. В метод `CreateHostBuilder` добавьте вызов метода расширения `UseUrls` и укажите номер порта `5001` для HTTPS, как показано ниже (выделено полужирным шрифтом):

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
      webBuilder.UseStartup<Startup>();
      webBuilder.UseUrls("https://localhost:5001");
    });
```

3. Откройте файл `Startup.cs` и добавьте в начало метода `ConfigureServices` оператор, чтобы добавить поддержку CORS, как показано ниже (выделено полужирным шрифтом):

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddCors();
```

4. Добавьте в метод `Configure` перед вызовом `UseEndpoints` оператор, чтобы использовать CORS и разрешить выполнение HTTP-запросов `GET`, `POST`, `PUT` и `DELETE` для любого сайта, такого как MVC для Northwind, имеющего источник https://localhost:5002:

```
// после UseRouting и перед UseEndpoints
app.UseCors(configurePolicy: options =>
  {
    options.WithMethods("GET", "POST", "PUT", "DELETE");
    options.WithOrigins(
```

```
    "https://localhost:5002" // для клиента MVC
  );
});
```

5. Выберите Terminal ▶ New Terminal (Терминал ▶ Новый терминал) и папку NorthwindService.
6. На панели TERMINAL (Терминал) запустите проект `NorthwindService` с помощью команды `dotnet run`. Убедитесь, что веб-служба прослушивает только порт 5001, как показано ниже:

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
```

7. Выберите Terminal ▶ New Terminal (Терминал ▶ Новый терминал) и папку NorthwindMvc.
8. На панели TERMINAL (Терминал) запустите проект `NorthwindMvc` с помощью команды `dotnet run`. Убедитесь, что сайт прослушивает порты 5000 и 5002, как показано ниже:

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5002
```

9. Запустите браузер Google Chrome. В адресной строке введите следующий URL: http://localhost:5000/. Обратите внимание: вы перенаправлены на HTTPS через порт 5002, и отображается главная страница сайта MVC для `Northwind`.
10. В форме клиента введите название страны, например Германия, Великобритания или США, нажмите кнопку Submit (Отправить) и увидите список клиентов (рис. 18.18).

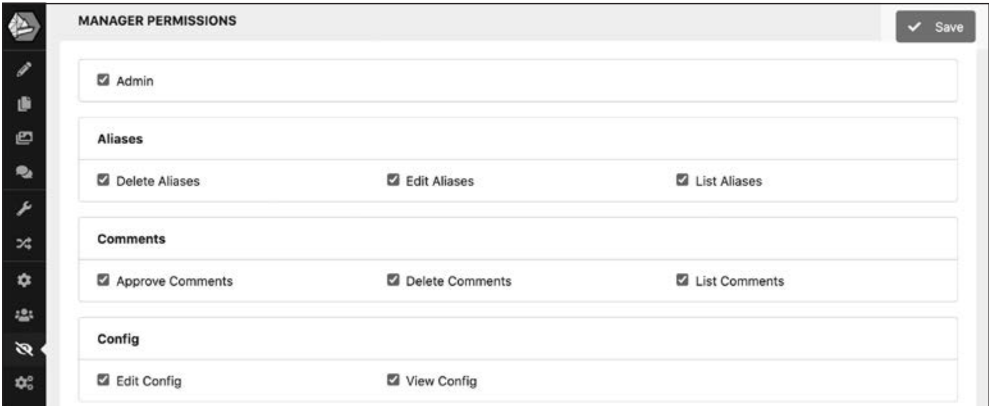


Рис. 18.18. Клиенты, проживающие в Великобритании

11. Вернитесь в браузер, очистите текстовое поле, в котором было введено название страны, нажмите кнопку Submit (Отправить) и просмотрите полученный список клиентов.

Реализация расширенных функций

Теперь, когда вы ознакомились с основами создания веб-службы и с последующим ее вызовом из клиента, рассмотрим некоторые дополнительные функции.

Мониторинг работоспособности — HealthCheck API

Существует множество платных сервисов, выполняющих проверку доступности сайта с помощью банального пингования, некоторые с более продвинутым анализом HTTP-ответа.

ASP.NET Core 2.2 и более поздние версии позволяют легко выполнять более подробные проверки работоспособности сайта. Например, ваш сайт может работать, но готов ли он полностью? Может ли он получить данные из своей базы данных?

1. Найдите и откройте файл `NorthwindService.csproj`.
2. Добавьте ссылку на проект, чтобы включить проверки работоспособности базы данных Entity Framework Core:

```
<PackageReference Include=
  "Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore"
  Version="5.0.0" />
```

3. На панели TERMINAL (Терминал) восстановите пакеты и скомпилируйте проект сайта:

```
dotnet build
```

4. Найдите и откройте файл `Startup.cs`.
5. В метод `ConfigureServices` добавьте оператор для добавления проверок работоспособности, в том числе в контекст базы данных Northwind, как показано ниже:

```
services.AddHealthChecks().AddDbContextCheck<Northwind>();
```

По умолчанию проверка контекста базы данных вызывает метод EF Core `CanConnectAsync`. Вы можете настроить выполнение операции, используя метод `AddDbContextCheck`.

6. В метод Configure добавьте оператор для использования базовых проверок работоспособности:

```
app.UseHealthChecks(path: "/howdoyoufeel");
```

7. Запустите веб-сервис. В адресной строке браузера введите следующий URL: https://localhost:5001/howdoyoufeel.
8. Обратите внимание, что в ответ отображается простой текст: Healthy.

Более подробную информацию о проверке работоспособности можно получить на сайте https://blogs.msdn.microsoft.com/webdev/2018/08/22/asp-net-core-2-2-0-preview1-healthcheck/.

Реализация анализаторов и соглашений Open API

В этой главе вы узнали, как включить Swagger для документирования веб-сервиса, дополнив класс Controller атрибутами.

В ASP.NET Core 2.2 или более поздних версиях содержатся API-анализаторы, которые использует отражение для классов контроллеров, которые были аннотированы атрибутом [ApiController], в целях их автоматического документирования. Анализатор предполагает некоторые соглашения API.

Чтобы использовать анализатор, в проекте необходимо ссылаться на пакет NuGet, как показано ниже:

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Api.Analyzers"
  Version="3.0.0" PrivateAssets="All" />
```

На момент написания книги версия пакета была 3.0.0-preview3-19153-02, но после публикации она уже будет полной версией 3.0.0. Проверить последнюю версию на предмет использования можно, перейдя по ссылке http://www.nuget.org/packages/Microsoft.AspNetCore.Mvc.Api.Analyzers/.

После установки контроллеры, которые не были должным образом оформлены, будут содержать предупреждения (зеленый знак тильды) и предупреждения при компиляции исходного кода с помощью команды dotnet build. Например, класс WeatherForecastController.

Автоматические исправления кода могут затем добавить соответствующие атрибуты [Produces] и [ProducesResponseType], хотя в настоящее время это работает только в среде разработки Visual Studio 2019. В программе Visual Studio Code вы

увидите предупреждение о том, что, по мнению анализатора, вам следует добавить атрибуты, но вам необходимо добавить их самостоятельно.

Обработка проходных отказов

Когда клиентское приложение или сайт вызывает веб-службу, это может быть в другой части света. Проблемы с сетью между клиентом и сервером могут вызвать ошибки, которые не имеют ничего общего с вашим кодом. Если клиент звонит и при этом возникает ошибка, приложение должно работать. При повторной попытке проблема может быть решена. Необходим способ, чтобы справиться с такими временными сбоями.

Чтобы преодолеть эти временные сбои, Microsoft рекомендует использовать стороннюю библиотеку Polly для реализации автоматических повторяющихся попыток с экспоненциальной задержкой. Вы определяете политику, а библиотека обрабатывает все остальное.

Более подробную информацию, касающуюся библиотеки Polly, вы можете найти на сайте https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/implement-resilient-applications/implement- http-call-retries-exponential-backoff-polly.

Система маршрутизации на основе конечных точек

В более ранних версиях ASP.NET Core система маршрутизации и расширяемая система промежуточного программного обеспечения не всегда легко работали вместе. Например, была необходимость реализовать технологию, подобную CORS, как в промежуточном программном обеспечении, так и в MVC, поэтому корпорация Microsoft вложила средства в улучшение маршрутизации с помощью новой системы *Endpoint Routing* (система маршрутизации на основе конечных точек), представленной в ASP.NET Core 2.2.

Корпорация Microsoft рекомендует по возможности переносить каждый проект ASP.NET Core в систему маршрутизации на основе конечных точек.

Маршрутизация на основе конечных точек разработана с целью обеспечить лучшую совместимость между платформами, нуждающимися в маршрутизации, например Razor Pages, MVC или Web API, и промежуточным программным обеспечением, которое должно понимать, как маршрутизация влияет на них, например локализация, авторизация, CORS и т. д.

Более подробную информацию о построении архитектуры приложения с использованием маршрутизации на основе конечных точек можно найти на сайте https://devblogs.microsoft.com/aspnet/asp-net-core-2-2-0-preview1-endpoint-routing/.

Система получила такое название, поскольку представляет собой таблицу маршрутов в виде скомпилированного дерева конечных точек, по которому система маршрутизации может эффективно перемещаться. Одно из самых больших улучшений — производительность маршрутизации и выбора метода действия.

Маршрутизация на основе конечных точек по умолчанию включена ASP.NET Core 2.2 или более поздней версии при условии установки совместимости с версией 2.2 или более поздней. Традиционные маршруты, зарегистрированные с помощью метода `MapRoute` или атрибутов, сопоставляются с новой системой.

Новая система маршрутизации включает в себя сервис генерации ссылок, зарегистрированный в качестве сервиса зависимостей, которому не требуется параметр `HttpContext`.

Настройки маршрутизации на основе конечных точек

Маршрутизация на основе конечных точек требует вызовы методов `app.UseRouting()` и `app.UseEndpoints()`:

- `app.UseRouting()` отмечает позицию конвейера, в которой принимается решение о маршрутизации;
- `app.UseEndpoints()` отмечает позицию конвейера, в которой выполняется выбранная конечная точка.

Промежуточное программное обеспечение, например локализация, выполняемая между ними, может видеть выбранную конечную точку и при необходимости переключаться на другую.

При маршрутизации на основе конечных точек используется тот же синтаксис шаблона маршрута, который применялся в ASP.NET MVC с 2010 года, и атрибут `[Route]`, представленный в ASP.NET MVC5 в 2013 году. Для миграции часто требуются только изменения в конфигурации `Startup`.

Контроллеры MVC, Razor Pages и фреймворки, такие как SignalR, ранее включались с помощью вызова метода `UseMvc()` или аналогичных методов. Однако теперь они добавляются в метод `UseEndpoints()`, поскольку все интегрированы в одну и ту же систему маршрутизации вместе с промежуточным программным обеспечением.

Определим некое промежуточное программное обеспечение, которое может выводить информацию о конечных точках.

1. Найдите и откройте файл `Startup.cs` и импортируйте пространства имен для работы с маршрутизацией на основе конечных точек:

```
using Microsoft.AspNetCore.Http;     // метод расширения GetEndpoint()
using Microsoft.AspNetCore.Routing;  // RouteEndpoint
```

2. Добавьте в метод `ConfigureServices` перед методом `UseEndpoints` оператор, чтобы определить лямбда-оператор для вывода информации о выбранной конечной точке в ходе выполнения каждого запроса:

```
app.Use(next => (context) =>
  {
    var endpoint = context.GetEndpoint();
    if (endpoint != null)
    {
      WriteLine("*** Name: {0}; Route: {1}; Metadata: {2}",
        arg0: endpoint.DisplayName,
        arg1: (endpoint as RouteEndpoint)?.RoutePattern,
        arg2: string.Join(", ", endpoint.Metadata));
    }

    // передача контекста следующему промежуточному
    // программному обеспечению в конвейере
    return next(context);
  });
```

Просматривая предыдущий код, обратите внимание на следующие моменты.

- Для метода `Use` требуется экземпляр `RequestDelegate` или эквивалентный лямбда-оператор.
- `RequestDelegate` содержит единственный параметр `HttpContext`, который упаковывает всю информацию о текущем HTTP-запросе (и соответствующем ответе).
- Импорт пространства имен `Microsoft.AspNetCore.Http` добавляет метод расширения `GetEndpoint` к экземпляру `HttpContext`.

3. Запустите веб-сервис.
4. В адресной строке браузера Google Chrome введите следующий URL: https://localhost:5001/weatherforecast.
5. На панели TERMINAL (Терминал) проанализируйте результат:

```
Request starting HTTP/1.1 GET https://localhost:5001/weatherforecast
*** Name: NorthwindService.Controllers.WeatherForecastController.
Get (NorthwindService); Route: Microsoft.AspNetCore.Routing.Patterns.
RoutePattern; Metadata: Microsoft.AspNetCore.Mvc.ApiControllerAttribute,
```

```
Microsoft.AspNetCore.Mvc.ControllerAttribute, Microsoft.AspNetCore.
Mvc.RouteAttribute, Microsoft.AspNetCore.Mvc.HttpGetAttribute,
Microsoft.AspNetCore.Routing.HttpMethodMetadata, Microsoft.
AspNetCore.Mvc.Controllers.ControllerActionDescriptor, Microsoft.
AspNetCore.Routing.RouteNameMetadata, Microsoft.AspNetCore.Mvc.
ModelBinding.UnsupportedContentTypeFilter, Microsoft.AspNetCore.Mvc.
Infrastructure.ClientErrorResultFilterFactory, Microsoft.AspNetCore.Mvc.
Infrastructure.ModelStateInvalidFilterFactory, Microsoft.AspNetCore.
Mvc.ApiControllerAttribute, Microsoft.AspNetCore.Mvc.ActionConstraints.
HttpMethodActionConstraint
```

6. Закройте браузер Google Chrome и остановите работу веб-сервиса.

Узнать больше о маршрутизации на основе конечных точек можно на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/routing.

Маршрутизация на основе конечных точек заменяет маршрутизацию на основе `IRouter`, используемую в ASP.NET Core 2.1 и более ранних версиях.

Если вам необходимо работать с ASP.NET Core 2.1 или более ранней версией, то о прежней системе маршрутизации можно прочитать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/routing?view=aspnetcore-2.1.

Добавление HTTP-заголовков для безопасности

ASP.NET Core имеет встроенную поддержку общих HTTP-заголовков безопасности, например HSTS. Также существует еще множество HTTP-заголовков, которые следует изучить.

Самый простой способ добавить эти заголовки — использовать класс промежуточного программного обеспечения.

1. В папке NorthwindService создайте файл с именем `SecurityHeadersMiddleware.cs` и измените его операторы:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Primitives;

namespace Packt.Shared
{
  public class SecurityHeaders
  {
    private readonly RequestDelegate next;
```

```
    public SecurityHeaders(RequestDelegate next)
    {
      this.next = next;
    }

    public Task Invoke(HttpContext context)
    {
      // добавьте любые заголовки ответа HTTP
      context.Response.Headers.Add(
      "super-secure", new StringValues("enable"));

      return next(context);
    }
  }
}
```

2. Откройте файл `Startup.cs` и перед вызовом `UseEndpoints` добавьте инструкцию для регистрации промежуточного программного обеспечения, как показано ниже.

   ```
   app.UseMiddleware<SecurityHeaders>();
   ```

3. Запустите веб-сервис.
4. Запустите браузер Google Chrome. Затем для записи запросов и ответов выберите Developer Tools (Инструменты разработчика) и вкладку Network (Сеть).
5. Перейдите по адресу https://localhost:5001/weatherforecast.
6. Обратите внимание на добавленный настраиваемый заголовок ответа HTTP с `super-secure` (рис. 18.19).

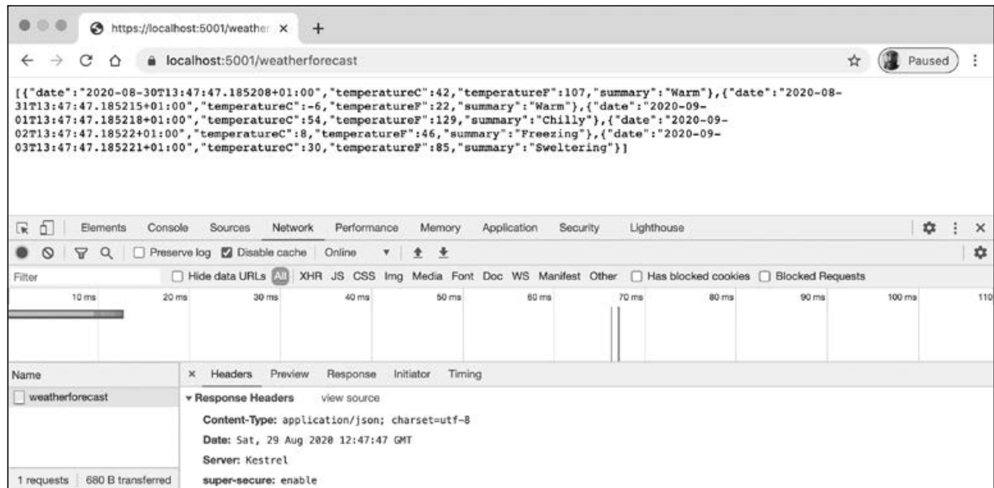


Рис. 18.19. Добавление настраиваемого HTTP-заголовка super-secure

Более подробную информацию об общих HTTP-заголовках для безопасности вы можете найти на сайте https://www.meziantou.net/security-headers-in-asp-net-core.htm.

Защита веб-сервисов

Улучшение веб-сервисов в .NET 5 включает простой метод расширения для включения анонимных HTTP-вызовов при использовании маршрутизации конечных точек.

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapControllers()
    .AllowAnonymous();
});
```

Прочие коммуникационные технологии

ASP.NET Core Web API не единственная технология Microsoft для реализации сервисов или обмена данными между компонентами распределенного приложения. Мы не будем подробно описывать эти технологии, однако вам необходимо знать, что они умеют и когда их следует использовать.

Windows Communication Foundation (WCF)

В 2006 году корпорация Microsoft выпустила .NET Framework 3.0 с несколькими базовыми платформами, одной из которых была *Windows Communication Foundation (WCF)*. Эта платформа абстрагировала реализацию бизнес-логики сервиса от технологии, используемой для связи с ним. Интенсивно применялась конфигурация XML для декларативного определения конечных точек, включая их адрес, привязку и контракт (часто называемые ABC конечных точек: address, binding, contract). Разобравшись, как это сделать, вы поймете, что это эффективная и в то же время гибкая технология.

Корпорация Microsoft решила не переносить WCF официально на .NET Core, но существует проект OSS, принадлежащий сообществу и называемый *Core WCF*, которым управляет .NET Foundation. Если вам нужно перенести существующий сервис из .NET Framework в .NET Core или создать клиент для сервиса WCF, то можете использовать Core WCF. Имейте в виду: Core WCF не может быть полноценным портом, поскольку части WCF зависят от операционной системы Windows.

Прочитать больше и скачать репозиторий Core WCF можно на сайте https://github.com/CoreWCF/CoreWCF.

Такие технологии, как WCF, позволяют создавать распределенные приложения. Клиентское приложение может выполнять *удаленные вызовы процедур* (remote procedure calls, RPC) серверного приложения. Вместо того чтобы использовать порт WCF, мы могли бы задействовать альтернативную технологию RPC.

gRPC

gRPC — современная высокопроизводительная RPC-платформа с открытым исходным кодом, которая может работать в любой среде.

Как и в случае с WCF, разработка API в gRPC построена на первичном определении контракта, что в данном случае позволяет избежать зависимости от языка реализации. Вы определяете контракты в файлах `.proto`, используя специфический синтаксис и инструменты для их преобразования в различные языки, такие как C#. Использование сети минимизируется с помощью двоичной сериализации `Protobuf`.

Более подробную информацию о gRPC можно получить на сайте https://grpc.io.

Корпорация Microsoft официально поддерживает gRPC в ASP.NET Core.

Информацию об использовании gRPC в ASP.NET Core можно найти на сайте https://docs.microsoft.com/ru-ru/aspnet/core/grpc/aspnetcore.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 18.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какой базовый класс вы должны унаследовать, чтобы создать класс контроллера для сервиса ASP.NET Core Web API?
2. Если вы дополнили свой класс `Controller` атрибутом `[ApiController]`, чтобы получить поведение по умолчанию, например автоматический ответ со статусом `400` для недопустимых моделей, то что еще должны сделать?

3. Что вы должны сделать, чтобы указать, какой метод действия контроллера будет выполняться в ответ на HTTP-запрос?
4. Что вы должны сделать, чтобы указать, какие ответы следует ожидать при вызове метода действия?
5. Перечислите три метода, которые можно вызывать для возврата ответов с разными кодами состояния.
6. Перечислите четыре способа тестирования веб-сервиса.
7. Почему не стоит оборачивать `HttpClient` в оператор `using`, чтобы очистить его ресурсы, когда закончите с ним работать, даже если он реализует интерфейс `IDisposable`, и что вы должны задействовать вместо этого?
8. Что такое CORS и почему важно включить его в веб-сервис?
9. Как вы можете разрешить клиентам определять, исправен ли ваш веб-сервис, в ASP.NET Core 2.2 и более поздних версиях?
10. Какие преимущества обеспечивает маршрутизация на основе конечных точек?

Упражнение 18.2. Создание и удаление клиентов с помощью HttpClient

Дополните проект сайта `NorthwindMvc`, добавив страницы, на которых у пользователя появится возможность заполнить форму для создания нового клиента или выполнить поиск клиента, а затем удалить его. Контроллер MVC должен совершать вызовы сервиса Northwind для создания и удаления клиентов.

Упражнение 18.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- создание Web API с помощью ASP.NET Core: https://docs.microsoft.com/ru-ru/aspnet/core/web-api/;
- Swagger: https://swagger.io/tools/;
- Swashbuckle для ASP.NET Core: https://github.com/domaindrivendev/Swashbuckle.AspNetCore;
- проверка работоспособности в ASP.NET Core: https://docs.microsoft.com/ru-ru/aspnet/core/host-and-deploy/health-checks;
- использование HttpClientFactory для реализации устойчивых HTTP-запросов: https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/implement-resilient-applications/use-httpclientfactory-to-implement-resilient-http-requests.

Резюме

Вы узнали, как создать сервис ASP.NET Core Web API, пригодный для вызова любым приложением на любой платформе, которая может отправлять HTTP-запрос и обрабатывать HTTP-ответ. Вы также научились тестировать и документировать API веб-сервисов с помощью Swagger, а также эффективно использовать сервисы.

Далее вы узнаете, как добавить интеллект в любое приложение, используя машинное обучение.

19 Разработка интеллектуальных приложений с помощью алгоритмов машинного обучения

Данная глава посвящена внедрению интеллекта в ваши приложения с помощью алгоритмов машинного обучения. Корпорация Microsoft создала кросс-платформенную библиотеку машинного обучения под названием ML.NET, предназначенную специально для разработчиков на C# и .NET.

В этой главе:

- общие сведения о машинном обучении;
- знакомство с ML.NET;
- рекомендация товаров пользователю.

Общие сведения о машинном обучении

Маркетологи в своих рекламных материалах используют такие термины, как *искусственный интеллект* или *наука о данных*. *Машинное обучение* — ее подмножество. Это один из практических способов добавить интеллект к программному обеспечению.

Вы можете изучить один из самых популярных и успешных методов науки о данных, записавшись на бесплатный восьминедельный курс Data Science: Machine Learning Гарвардского университета по следующей ссылке: https://www.edx.org/course/data-science-machine-learning-2.

В этой книге невозможно охватить в одной главе всю информацию о машинном обучении. Если вам нужно понять, как подобные алгоритмы работают «под капотом», то понадобится разобраться в темах науки о данных, включая матанализ,

статистику, теорию вероятностей и линейную алгебру. Затем будет необходимо углубленно изучить тему машинного обучения.

Чтобы узнать больше о машинном обучении, прочитайте книгу Себастьяна Рашка и Вахида Мирджалили «Python и машинное обучение»: https://www.packtpub.com/product/python-machine-learning-third-edition/9781789955750.

Моя цель — в этой главе дать вам минимальное концептуальное понимание, которое позволит реализовать ценное и практическое применение машинного обучения: порекомендовать пользователю покупку товаров на сайте электронной коммерции для повышения стоимости каждого заказа. Проанализировав все, что необходимо для этого сделать, вы можете сами решить, стоит ли изучать эту тему, или вы предпочитаете вкладывать усилия в изучение других тем, таких как создание сайтов или мобильных приложений.

Жизненный цикл систем машинного обучения

Жизненный цикл систем машинного обучения состоит из четырех этапов.

- *Анализ проблемы*. Какую проблему вы пытаетесь решить?
- *Сбор и обработка данных*. Необработанные данные, необходимые для решения проблемы, часто нужно преобразовать в форматы, подходящие для обработки алгоритмами машинного обучения.
- *Моделирование* делится на три подэтапа.
    - *Идентификация характеристик*. Это значения, влияющие на прогнозы, например пройденное расстояние и время суток, которые влияют на стоимость поездки на такси.
    - *Обучение модели*. Выберите и примените алгоритмы и установите гиперпараметры для генерации одной или нескольких моделей. Гиперпараметры устанавливаются до начала процесса обучения, в отличие от других параметров, полученных в его процессе.
    - *Оценка модели*. Выберите, какая модель лучше всего решает исходную проблему. Оценка моделей — это ручная задача, которая может занять месяцы.
- *Развертывание модели*. Внедрите модель в приложение, где она используется для прогнозирования реальных данных.

Ваша работа еще не закончена!

После развертывания модели для поддержания ее эффективности вам необходимо регулярно ее пересматривать. Со временем предсказания, которые она делает,

могут меняться и ухудшаться в связи с потенциальным изменением данных со временем.

Вы не должны предполагать статическую связь между входными и выходными данными, особенно при прогнозировании поведения человека, поскольку мода меняется. Популярность фильмов с супергероями в этом году не означает их популярности в следующем. Эта проблема называется *смещением концепций* или распадом модели. Если вы подозреваете данную проблему, то вам следует переучить модель или даже переключиться на лучший алгоритм или более надежные значения гиперпараметров.

Решить, какой алгоритм использовать и какие значения подходят для его гиперпараметров, сложно, поскольку комбинация потенциальных алгоритмов и значений гиперпараметров бесконечна.

Более подробную информацию о различных ролях специалистов, занимающихся наукой о данных и машинным обучением, можно найти на сайте https://www.datasciencecentral.com/profiles/blogs/difference-between-machine-learning-data-science-ai-deep-learning.

Наборы данных для обучения и тестирования

Вы не должны использовать весь свой набор данных для обучения вашей модели. Необходимо разделить его на два: *набор для обучения* и *набор для тестирования*. Неудивительно, что первый применяется для обучения модели. Второй служит для оценки того, что модель делает достаточно хорошие прогнозы, прежде чем ее разворачивать. Используй вы весь набор данных для обучения, у вас не осталось бы данных для проверки модели.

Разделение между обучением и тестированием может быть случайным для некоторых сценариев, но будьте осторожны! Вам необходимо понять, может ли набор данных иметь регулярные вариации. Например, использование такси зависит от времени суток и даже сезона и города. Скажем, в Нью-Йорке, как правило, очень популярно ездить на такси в течение всего года, в любое время. В Мюнхене же такси особо загружены во время Октоберфеста.

Если на ваш набор данных влияют сезонность и другие факторы, вам необходимо стратегически разделить набор данных. Вдобавок следует убедиться, что модель не *излишне подстроена* под специфические обучающие данные, как в примере ниже.

Когда я изучал информатику в Бристольском университете в период между 1990 и 1993 годами, на уроке нейронных сетей нам рассказывали историю (возможно, вымышленную, но она отображает важный момент). Британская армия привлекала

специалистов по обработке и анализу данных для создания модели машинного обучения, призванной обнаруживать русские танки, замаскированные в лесах Восточной Европы. Модель была снабжена тысячами изображений, но когда пришло время продемонстрировать ее возможности в реальности, она потерпела неудачу.

Во время тщательного анализа проекта ученые поняли, что все изображения, которые они использовали для обучения модели, соответствовали времени года весна, когда листва была живой, с оттенками зеленого. Однако испытание в реальных условиях произошло осенью, когда та была красной, желтой и коричневой.

Модель была *излишне подстроена* под весеннюю листву. Будь она более обобщенной, то могла бы работать и в другие сезоны. Обратная проблема — недостаточно точная подгонка; она описывает слишком обобщенную модель, которая не дает удовлетворительных результатов, будучи примененной к конкретному контексту.

О чрезмерной и недостаточно близкой подгонке и о том, как это компенсировать, можно узнать на сайте https://elitedatascience.com/overfitting-in-machine-learning.

Задачи машинного обучения

Существует множество задач или сценариев, где машинное обучение может помочь разработчикам.

- *Двоичная классификация* — это классификация элементов во входном наборе данных между двумя группами, прогнозирование того, к какой группе принадлежит каждый элемент. Например, решение о том, считается ли обзор книги Amazon *положительным* или *отрицательным*, что также известно как анализ эмоциональной окраски высказываний. Другие примеры включают в себя обнаружение спама в электронной почте и мошенничества с кредитными картами.
- *Многоклассовая классификация* относит экземпляры к одному из трех или более классов, предсказывая, к какой группе принадлежит каждый из них. Примером может послужить решение о том, следует ли классифицировать новостную статью как *светское обозрение*, *спорт*, *политику* или *науку и технику*. Двоичные и многоклассовые экземпляры — это примеры контролируемой классификации, поскольку определяемые метки должны быть заданы предварительно.
- *Кластеризация* предназначена для группировки элементов входных данных, чтобы элементы в одном кластере были больше похожи друг на друга, чем в других кластерах. Это не то же самое, что классификация, поскольку она не выдает каждому кластеру метку, и потому метки не должны быть предопределены, как при классификации. Вот почему кластеризация — пример *неконтролируемой*

классификации. После кластеризации группу можно обработать, чтобы определить шаблоны, а затем и метки.

- *Ранжирование* предназначено для упорядочения элементов входных данных на основе таких свойств, как звездные обзоры, контекст, лайки и т. д.
- *Рекомендации* предназначены для предложения других элементов, таких как товары или контент, которые могут понравиться пользователю на основе их прошлого поведения по сравнению с другими пользователями.
- *Регрессия* предназначена для прогнозирования числовых значений из входных данных. Регрессию можно использовать для прогнозирования и определения времени продажи товара, или стоимости поездки на такси из аэропорта Хитроу до отеля в центре Лондона, или расчета количества велосипедов в конкретном районе Амстердама.
- *Обнаружение аномалий* предназначено для выявления редких или необычных данных, которые могут указывать на проблему, подлежащую устранению, в таких областях, как медицинское, финансовое и механическое обслуживание.
- *Глубокое обучение* предназначено для обработки больших и сложных двоичных входных данных вместо входных данных в более структурированном формате, например для задач компьютерного распознавания образов, таких как обнаружение объектов и классификации изображений, или аудиозадач наподобие распознавания речи и *обработки естественного языка* (natural language processing, NLP).

Машинное обучение Microsoft Azure

В оставшейся части данной главы будет описано использование .NET-пакета с открытым исходным кодом в целях реализации машинного обучения. Однако прежде, чем мы углубимся в это, необходимо понять важную альтернативу.

Внедрение машинного обучения требует участия людей, имеющих сильные навыки в математике или смежных областях. Специалисты по обработке и анализу данных пользуются большим спросом, а трудоемкость и стоимость их найма не позволяют некоторым организациям внедрять машинное обучение в собственные приложения. Организация может иметь доступ к хранилищам данных, накопленных за многие годы, но ей трудно применять машинное обучение для улучшения принимаемых ими решений.

Машинное обучение Microsoft Azure преодолевает эти препятствия, предлагая предварительно созданные модели машинного обучения для выполнения задач, таких как распознавание лиц и обработка языка. Это позволяет организациям, не имеющим собственных специалистов по обработке данных, получать некоторую выгоду от своих данных. Поскольку организации нанимают специалистов или их разработчики приобретают навыки работы с данными, они могут разрабатывать свои модели для работы в Azure ML.

Однако по мере того, как организации развиваются и осознают ценность владения своими моделями машинного обучения и возможности применять их в любом месте, этим организациям все больше и больше будет необходима платформа, на которой их существующие разработчики могут начать работу с настолько малой кривой обучения, насколько возможно.

Знакомство с ML.NET

ML.NET — кросс-платформенный фреймворк с открытым кодом от Microsoft для реализации машинного обучения на .NET.

Разработчики C# могут использовать существующие API из .NET Standard для интеграции машинного обучения в свои приложения, не зная подробностей о создании и поддержании модели машинного обучения.

Документацию о ML.NET можно прочитать на сайте https://devblogs.microsoft.com/dotnet/introducing-ml-net-cross-platform-proven-and-open-source-machine-learning-framework/.

В настоящее время ML.NET содержит библиотеки машинного обучения, созданные Microsoft Research и используемые продуктами Microsoft, такими как PowerPoint, для разумной рекомендации шаблонов стилей, основанных на содержании презентации. Вскоре ML.NET будет поддерживать и другие популярные библиотеки, например Accord.NET, CNTK, Light GBM и TensorFlow, но мы не будем рассматривать их в этой книге.

Accord.NET Framework — фреймворк машинного обучения на .NET, объединенный с библиотеками обработки аудио и изображений, полностью написанный на языке C#. Более подробную информацию можно получить на сайте http://accord-framework.net.

Знакомство с Infer.NET

Фреймворк *Infer.NET* был создан компанией Microsoft Research в Кембридже в 2004 году. С тех пор об Infer.NET были написаны сотни статей.

Больше информации о Microsoft Infer.NET можно узнать на сайте https://docs.microsoft.com/ru-ru/dotnet/machine-learning/how-to-guides/matchup-app-infer-net.

Разработчики могут включить знания предметной области в модель для создания пользовательских алгоритмов вместо того, чтобы попытаться подогнать проблему к существующим алгоритмами, как бы это делалось с ML.NET.

Фреймворк Infer.NET используется корпорацией Microsoft для такой продукции, как Microsoft Azure, Xbox и Bing для поиска и перевода.

Infer.NET можно применять для классификации, рекомендаций и кластеризации. Мы не станем описывать его в этой книге.

Информацию о создании приложения, выдающего список подходящих игр, с помощью Infer.NET и вероятностного программирования можно получить на сайте https://docs.microsoft.com/ru-ru/dotnet/machine-learning/how-to-guides/matchup-app-infer-net.

Обучающие конвейеры ML.NET

Типичный конвейер состоит из шести таких этапов, как:

- *загрузка данных* — ML.NET поддерживает загрузку данных из следующих форматов: текст (CSV, TSV), Parquet, двоичный файл, `IEnumerable<T>` и наборы файлов;
- *преобразования* — ML.NET поддерживает следующие преобразования: обработку текста, изменение схемы (то есть структуры), обработку пропущенных значений, кодирование категориальных значений, нормализацию и выбор характеристики;
- *алгоритмы* — ML.NET поддерживает следующие алгоритмы: линейные, усиленные деревья, метод k-средних, *метод опорных векторов* (Support Vector Machine, SVM) и алгоритм среднего персептрона;
- *обучение модели* — вызов метода ML.NET `Train` для создания модели `PredictionModel`, которую вы можете использовать для прогнозирования;
- *оценка модели* — ML.NET поддерживает несколько оценщиков для определения точности вашей модели по различным метрикам;
- *развертывание модели* — ML.NET позволяет экспортировать модель в виде двоичного файла для развертывания в любом приложении .NET.

Традиционное приложение Hello World для машинного обучения предсказывает тип цветка ириса на основе четырех характеристик: длины и ширины лепестка, длины и ширины чашелистика. Вы можете пройти десятиминутный урок на сайте https://dotnet.microsoft.com/learn/machinelearning-ai/ml-dotnet-get-started-tutorial/intro.

Концепции обучения моделей

Система типов .NET не была разработана для машинного обучения и анализа данных, поэтому для этих задач более целесообразно применять несколько специализированных типов.

При работе с моделями ML.NET использует следующие типы .NET.

- Интерфейс `IDataView` представляет собой набор данных, которые:
  - *неизменяемые*, то есть не могут измениться;
  - *пригодны для использования курсора*, то есть с его помощью можно перебирать данные;
  - *лениво вычисляются*, то есть такие задачи, как преобразование, выполняются только когда курсор перебирает данные;
  - *разнородные*, то есть данные могут быть разных типов;
  - *схематизированы*, то есть имеют четко определенную структуру.

Более подробную информацию о дизайне интерфейса IDataView можно получить на сайте https://github.com/dotnet/machinelearning/blob/master/docs/code/IDataViewDesignPrinciples.md.

- Абстрактный класс `DataViewType` — все типы столбцов `IDataView` являются производными от него. Векторные типы требуют информации о размерности, чтобы указать длину векторного типа.
- Интерфейс `ITransformer` представляет компонент, который принимает входные данные, изменяет их и возвращает выходные. Например, токенизатор принимает текстовый столбец, содержащий фразы, в качестве входных данных и выводит векторный столбец с отдельными словами, извлеченными из фраз и расположенными вертикально в столбце. Большинство преобразователей работают над одним столбцом в единицу времени. Новые преобразователи могут быть сконструированы путем объединения других преобразователей в цепочку.
- Интерфейс `IDataReader<T>` представляет компонент для создания данных. Он берет экземпляр `T` и возвращает из него данные.
- Интерфейс `IEstimator<T>` представляет объект, обучающийся на данных. Результатом обучения будет преобразователь. Эстиматоры «жадные»: каждый вызов их метода `Fit` приводит к обучению, на что может уйти много времени!
- Класс `PredictionEngine<TSrc, TDst>` представляет функцию, которую можно рассматривать как машину, применяющую преобразователь к одной строке

во время прогнозирования. Если у вас имеется большое количество входных данных, для которых вы хотите сделать прогнозы, то вам необходимо создать представление данных, вызвать метод `Transform` для модели, чтобы сгенерировать строки прогноза, а затем задействовать курсор для чтения результатов. В действительности распространен сценарий использования в качестве входных данных одной строки, для которой вы хотите сделать прогноз, поэтому в целях упрощения процесса вы можете применить механизм прогнозирования.

Пропущенные значения и типы ключей

Язык R популярен для машинного обучения и в целях обозначения пропущенного значения использует специальное значение `NA`. Пакет ML.NET следует этому соглашению.

Типы ключей применяются для данных, представленных в виде числовых значений с кардинальностью:

```
[KeyType(10)]
public uint NumberInRange1To10 { get; set; }
```

Репрезентативным или базовым типом должен быть один из четырех целочисленных типов без знака .NET: `byte`, `ushort`, `uint` и `ulong`. Нулевое значение всегда означает `NA`, указывая, что его значение отсутствует. Репрезентативное значение 1 — всегда первое действительное значение типа ключа.

Счет должен быть установлен на единицу больше, чем рассчетное максимальное значение при счете от 1, поскольку 0 зарезервирован для пропущенных значений. Например, для диапазона 0–9 кардинальность должна быть равна 10. Любые значения за пределами указанной кардинальности будут отображаться в представление отсутствующего значения: 0.

Характеристики и метки

Входные данные модели машинного обучения называются *характеристиками*. Так, если существовала линейная регрессия, когда одно непрерывное количество, скажем цена бутылки вина, пропорционально другому, например рейтингу, предоставленному виноделами, то цена будет единственной характеристикой.

Значения, используемые для обучения модели машинного обучения, называются *метками*. В примере с вином значения рейтинга в наборе обучающих данных — метки.

В некоторых моделях значение метки неважно, поскольку наличие строки данных представляет совпадение. Это особенно часто встречается в рекомендациях.

Рекомендация товаров пользователю

Практическое применение машинного обучения, которое мы будем внедрять, — создание рекомендаций по использованию товаров на сайте электронной коммерции с целью повысить ценность заказа клиента.

Проблема заключается в том, как решить, какие товары рекомендовать пользователю.

Анализ проблем

Второго октября 2006 года компания Netflix объявила открытый конкурс на лучший алгоритм для прогнозирования пользовательских рейтингов фильмов, основанный только на предыдущих рейтингах. В наборе данных Netflix было 17 000 фильмов, 500 000 пользователей и 100 миллионов оценок. Например, пользователь 437 822 присваивает фильму 12 934 рейтинг 4 из 5.

Сингулярное разложение (Singular Value Decomposition, SVD) — метод декомпозиции для сокращения матрицы, призванный упростить последующие вычисления. Саймон Функ поделился с сообществом тем, как он и его команда использовали метод, чтобы подняться на вершину рейтинга в ходе соревнования.

Узнать больше об использовании SVD Саймона Функа можно на сайте https://sifter.org/~simon/journal/20061027.2.html.

Подходы Саймона Функа были предложены для рекомендательных систем. Матричная факторизация — класс алгоритмов совместного фильтрования, применяемых в рекомендательных системах, использующих SVD.

Более подробную информацию об использовании матричной факторизации в рекомендательных системах можно получить на сайте https://en.wikipedia.org/wiki/Matrix_factorization_(recommender_systems).

Мы будем использовать алгоритм одноклассной матричной факторизации, поскольку располагаем только информацией об истории заказов на покупку. Товарам не были присвоены рейтинги или другие факторы, которые можно было бы задействовать в других многофакторных подходах факторизации.

Оценка, полученная с помощью матричной факторизации, сообщает нам о вероятности положительного результата. Чем больше значение оценки, тем выше вероятность. Оценка не является вероятностью, поэтому, делая прогнозы, мы должны предсказать множество оценок совместной покупки товаров и отсортировать их по наивысшему баллу.

В матричной факторизации используется метод совместного фильтрования, который предполагает следующее: если Alice (Алиса) относительно товара придерживается того же мнения, что и Bob (Боб), то, скорее всего, относительно другого товара она будет думать так же, как он. Следовательно, она с большей вероятностью добавит тот товар в корзину, который выбрал Bob (Боб).

Сбор и обработка данных

Пример базы данных Northwind содержит следующие таблицы:

- `Products` — состоит из 77 строк, каждая из которых содержит идентификатор товара типа `integer`;
- `Orders` — состоит из 830 строк, каждая из которых содержит одну или несколько связанных строк с подробными сведениями;
- `Order Details` — состоит из 2155 строк, каждая из которых содержит целочисленный идентификатор товара, обозначающий заказанный товар.

Мы можем использовать эти данные, чтобы создать наборы данных для моделей обучения и тестирования, а те, в свою очередь, могут затем делать прогнозы относительно других товаров, которые клиент может добавить в корзину.

Например, клиент Vinet сделал заказ № 10 248 на три товара с идентификаторами 11, 42 и 72 (рис. 19.1).

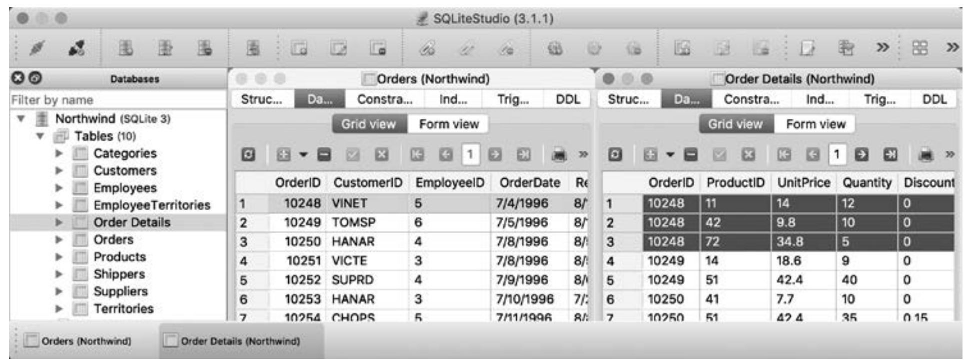


Рис. 19.1. Информация о заказе № 10 248

Мы напишем запрос LINQ для перекрестного соединения этих данных, чтобы сгенерировать простой текстовый файл с двумя столбцами, отображающими товары, которые были куплены совместно:

```
ProductID   CoboughtProductID
11          42
11          72
42          11
42          72
72          11
72          42
```

К сожалению, база данных `Northwind` практически каждый товар сопоставляет с любым другим. Чтобы получить более реалистичные наборы данных, мы будем фильтровать их в зависимости от страны. Мы создадим три набора данных: для Германии, Великобритании и США. Последний будет использоваться для тестирования.

Разработка проекта сайта NorthwindML

Мы создадим сайт ASP.NET Core MVC, который отображает список всех товаров, сгруппированных по категориям, и позволяет пользователю добавлять товары в корзину. Их корзина будет храниться во временном файле cookie в виде списка идентификаторов товаров, разделенных тире.

1. В папке PracticalApps создайте подпапку NorthwindML.
2. В программе Visual Studio Code откройте рабочую область `PracticalApps` и добавьте в нее папку NorthwindML.
3. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку NorthwindML.
4. На панели `TERMINAL` (Терминал) для создания нового проекта ASP.NET Core MVC введите следующую команду:

   ```
   dotnet new mvc
   ```

5. Найдите и откройте файл проекта `NorthwindML.csproj`.
6. Добавьте ссылки на пакеты SQLite, ML.NET и ML.NET Recommender, а также на проекты библиотеки контекста и сущностных классов `Northwind`, созданных в главе 14, как показано ниже (выделено полужирным шрифтом):

   ```
   <Project Sdk="Microsoft.NET.Sdk.Web">
     <PropertyGroup>
       <TargetFramework>net5.0</TargetFramework>
     </PropertyGroup>

     <ItemGroup>
       <PackageReference
   ```

```
      Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson"
      Version="5.0.0" />
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="5.0.0" />
    <PackageReference Include="Microsoft.ML" Version="1.5.1" />
    <PackageReference Include="Microsoft.ML.Recommender"
      Version="0.17.1" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindContextLib\NorthwindContextLib.csproj" />
    <ProjectReference Include=
      "..\NorthwindEmployees\NorthwindEmployees.csproj" />
  </ItemGroup>
</Project>
```

7. На панели TERMINAL (Терминал) необходимо восстановить пакеты и скомпилировать проект с помощью следующей команды:

```
dotnet build
```

8. Найдите и откройте файл класса `Startup.cs` и импортируйте пространства имен `Packt.Shared`, `System.IO` и `Microsoft.EntityFrameworkCore`.
9. Чтобы зарегистрировать контекст данных Northwind, добавьте в метод `ConfigureServices` оператор:

```
string databasePath = Path.Combine("..", "Northwind.db");
services.AddDbContext<Northwind>(options =>
  options.UseSqlite($"Data Source={databasePath}"));
```

Создание моделей данных и представления

Проект `NorthwindML` будет имитировать сайт электронной коммерции, позволяющий пользователям добавлять желаемые товары в свою корзину. В первую очередь определим некоторые модели для представления этой задачи.

1. В папке Models создайте файл класса `CartItem.cs` и добавьте операторы, чтобы определить класс со свойствами для идентификатора и наименования товара, как показано ниже:

```
namespace NorthwindML.Models
{
  public class CartItem
  {
    public long ProductID { get; set; }
    public string ProductName { get; set; }
  }
}
```

2. В папке Models создайте файл класса `Cart.cs` и добавьте операторы, чтобы определить класс со свойствами для элементов в корзине:

```
using System.Collections.Generic;

namespace NorthwindML.Models
{
  public class Cart
  {
    public IEnumerable<CartItem> Items { get; set; }
  }
}
```

3. В папке Models создайте файл класса `ProductCobought.cs` и добавьте операторы для определения класса со свойствами, используемыми для записи в случае приобретения товара вместе с другим товаром, и кардинальности (или максимально возможного значения) свойства `ProductID`:

```
using Microsoft.ML.Data;

namespace NorthwindML.Models
{
  public class ProductCobought
  {
    [KeyType(77)] // максимально возможное значение ProductID
    public uint ProductID { get; set; }

    [KeyType(77)]
    public uint CoboughtProductID { get; set; }
  }
}
```

4. В папке Models создайте файл класса `Recommendation.cs`, который будет использоваться в качестве вывода алгоритма машинного обучения. Добавьте операторы для определения класса со свойствами, применяемыми для отображения рекомендованного идентификатора товара с его характеристиками, который будет задействован в качестве результат алгоритма машинного обучения:

```
namespace NorthwindML.Models
{
  public class Recommendation
  {
    public uint CoboughtProductID { get; set; }
    public float Score { get; set; }
  }
}
```

5. В папке Models создайте файл класса `EnrichedRecommendation.cs` и добавьте операторы для наследования из класса `Recommendation` с дополнительным

свойством, позволяющим отображать наименование товара, как показано ниже:

```
namespace NorthwindML.Models
{
  public class EnrichedRecommendation : Recommendation
  {
    public string ProductName { get; set; }
  }
}
```

6. В папке Models создайте файл класса `HomeCartViewModel.cs` и добавьте операторы, чтобы определить класс со свойствами для хранения корзины покупок пользователя и списка рекомендаций:

```
using System.Collections.Generic;

namespace NorthwindML.Models
{
  public class HomeCartViewModel
  {
    public Cart Cart { get; set; }
    public List<EnrichedRecommendation> Recommendations { get; set; }
  }
}
```

7. В папке Models создайте файл класса `HomeIndexViewModel.cs` и добавьте операторы, чтобы определить класс со свойствами и показать, были ли созданы наборы данных для обучения:

```
using System.Collections.Generic;
using Packt.Shared;

namespace NorthwindML.Models
{
  public class HomeIndexViewModel
  {
    public IEnumerable<Category> Categories { get; set; }
    public bool GermanyDatasetExists { get; set; }
    public bool UKDatasetExists { get; set; }
    public bool USADatasetExists { get; set; }
    public long Milliseconds { get; set; }
  }
}
```

8. На панели TERMINAL (Терминал) скомпилируйте проект:

```
dotnet build
```

Реализация контроллера

Теперь мы можем изменить существующий класс `HomeContoller`, чтобы выполнить необходимые операции.

1. В папке wwwroot создайте папку Data, в которой будут храниться наборы данных для обучающих моделей.
2. В папке Controllers найдите и откройте файл `HomeController.cs`. Импортируйте некоторые пространства имен:

```
using Packt.Shared;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Hosting;
using System.IO;
using Microsoft.ML;
using Microsoft.ML.Data;
using Microsoft.Data;
using Microsoft.ML.Trainers;
```

3. Объявите в классе `HomeController` поля для названия файла и стран, для которых мы будем создавать наборы данных:

```
private readonly static string datasetName = "dataset.txt";
private readonly static string[] countries =
  new[] { "Germany", "UK", "USA" };
```

4. Объявите некоторые поля для зависимостей: сервиса данных `Northwind` и сервиса среды веб-сервера — и задайте их в конструкторе:

```
// сервисы зависимости
private readonly ILogger<HomeController> _logger;
private readonly Northwind db;
private readonly IWebHostEnvironment webHostEnvironment;

public HomeController(ILogger<HomeController> logger,
  Northwind db, IWebHostEnvironment webHostEnvironment)
{
  _logger = logger;
  this.db = db;
  this.webHostEnvironment = webHostEnvironment;
}
```

5. Добавьте закрытый метод, чтобы вернуть путь к файлу, хранящийся на сайте в папке Data:

```
private string GetDataPath(string file)
{
  return Path.Combine(webHostEnvironment.ContentRootPath,
    "wwwroot", "Data", file);
}
```

6. Добавьте закрытый метод для создания экземпляра класса `HomeIndexViewModel`, в который загружаются все товары из базы данных `Northwind` и который указывает, были ли созданы наборы данных:

```
private HomeIndexViewModel CreateHomeIndexViewModel()
{
  return new HomeIndexViewModel
  {
    Categories = db.Categories
      .Include(category => category.Products),
    GermanyDatasetExists = System.IO.File.Exists(
      GetDataPath("germany-dataset.txt")),
    UKDatasetExists = System.IO.File.Exists(
      GetDataPath("uk-dataset.txt")),
    USADatasetExists = System.IO.File.Exists(
      GetDataPath("usa-dataset.txt"))
  };
}
```

Нам пришлось добавить префикс `System.IO` к классу `File`, поскольку класс `ControllerBase` содержит метод `File`, что вызвало бы конфликт совпадения имен.

7. В метод действия `Index` добавьте операторы для создания модели представления и передайте ее представлению Razor, как показано ниже (выделено полужирным шрифтом):

```
public IActionResult Index()
{
  var model = CreateHomeIndexViewModel();
  return View(model);
}
```

8. Добавьте метод действия, который генерирует наборы данных для каждой страны, а затем возращает представление `Index` по умолчанию:

```
public IActionResult GenerateDatasets()
{
  foreach (string country in countries)
  {
    IEnumerable<Order> ordersInCountry = db.Orders
      // фильтр в зависимости от страны для создания
      // разных наборов данных
      .Where(order => order.Customer.Country == country)
      .Include(order => order.OrderDetails)
      .AsEnumerable(); // переключение на сторону клиента

    IEnumerable<ProductCobought> coboughtProducts =
      ordersInCountry.SelectMany(order =>
        from lineItem1 in order.OrderDetails // перекрестное соединение
        from lineItem2 in order.OrderDetails
        select new ProductCobought
```

```
          {
            ProductID = (uint)lineItem1.ProductID,
            CoboughtProductID = (uint)lineItem2.ProductID
          })
          // исключение совпадений между одним и тем же товаром
          .Where(p => p.ProductID != p.CoboughtProductID)
          // удаление дубликатов с помощью группировки по обоим значениям
          .GroupBy(p => new { p.ProductID, p.CoboughtProductID })
          .Select(p => p.FirstOrDefault())
          // упрощение чтения результатов человеком путем сортировки
          .OrderBy(p => p.ProductID)
          .ThenBy(p => p.CoboughtProductID);

      StreamWriter datasetFile = System.IO.File.CreateText(
        path: GetDataPath($"{country.ToLower()}-{datasetName}"));

      // заголовок, разделенный табуляцией
      datasetFile.WriteLine("ProductID\tCoboughtProductID");

      foreach (var item in coboughtProducts)
      {
        datasetFile.WriteLine("{0}\t{1}",
          item.ProductID, item.CoboughtProductID);
      }
      datasetFile.Close();
    }
    var model = CreateHomeIndexViewModel();
    return View("Index", model);
  }
```

Обучение рекомендательных моделей

Чтобы установить максимальное значение для идентификатора товара, в наборе данных мы предоставляем `KeyType`. Идентификаторы товара уже закодированы как целые числа, что необходимо для алгоритма, который мы будем использовать. Поэтому для обучения моделей достаточно вызвать метод `MatrixFactorizationTrainer` с несколькими дополнительными параметрами.

1. В файл `HomeController.cs` добавьте метод действия для обучения моделей, как показано ниже:

```
public IActionResult TrainModels()
{
  var stopWatch = Stopwatch.StartNew();

  foreach (string country in countries)
  {
    var mlContext = new MLContext();

    IDataView dataView = mlContext.Data.LoadFromTextFile(
      path: GetDataPath($"{country}-{datasetName}"),
      columns: new[]
```

```
      {
        new TextLoader.Column(name: "Label",
          dataKind: DataKind.Double, index: 0),
        // Количество ключей — это кардинальность,
        // то есть максимально допустимое значение.
        // Данный столбец используется для обучения модели.
        // Когда результаты показываются, столбцы сопоставляются
        // с экземплярами нашей модели, которые могли бы иметь
        // другую кардинальность, но имеют такую же.
        new TextLoader.Column(
          name: nameof(ProductCobought.ProductID),
          dataKind: DataKind.UInt32,
          source: new [] { new TextLoader.Range(0) },
          keyCount: new KeyCount(77)),
        new TextLoader.Column(
          name: nameof(ProductCobought.CoboughtProductID),
          dataKind: DataKind.UInt32,
          source: new [] { new TextLoader.Range(1) },
          keyCount: new KeyCount(77))
      },
      hasHeader: true,
      separatorChar: '\t');

    var options = new MatrixFactorizationTrainer.Options
    {
      MatrixColumnIndexColumnName =
        nameof(ProductCobought.ProductID),
      MatrixRowIndexColumnName =
        nameof(ProductCobought.CoboughtProductID),
      LabelColumnName = "Label",
      LossFunction = MatrixFactorizationTrainer
        .LossFunctionType.SquareLossOneClass,
      Alpha = 0.01,
      Lambda = 0.025,
      C = 0.00001
    };

    MatrixFactorizationTrainer mft = mlContext.Recommendation()
      .Trainers.MatrixFactorization(options);

    ITransformer trainedModel = mft.Fit(dataView);

    mlContext.Model.Save(trainedModel,
      inputSchema: dataView.Schema,
      filePath: GetDataPath($"{country}-model.zip"));
  }

  stopWatch.Stop();

  var model = CreateHomeIndexViewModel();
  model.Milliseconds = stopWatch.ElapsedMilliseconds;
  return View("Index", model);
}
```

Реализация корзины с рекомендациями

Создадим функцию корзины покупок и позволим пользователям добавлять товары. Вдобавок в корзине будут видны рекомендации по покупке других товаров, которые можно быстро добавить.

Такой тип рекомендации известен как *совместная покупка* или «товары, часто покупаемые вместе». Это значит, клиентам будет рекомендован набор товаров, основанный на истории покупок их и других клиентов.

В этом примере мы всегда будем применять модель Германии для прогнозирования. На реальном сайте вы можете выбрать модель на основе текущего местоположения пользователя, чтобы он получал рекомендации, аналогичные другим пользователям из этой страны.

1. Добавьте в файл `HomeController.cs` метод действия, который добавляет товар в корзину, а затем отображает корзину с тремя лучшими рекомендациями по покупке других товаров, которые пользователь может добавить:

```
// GET /Home/Cart
// отображение корзины и рекомендаций
// GET /Home/Cart/5
// добавление товара в корзину
public IActionResult Cart(int? id)
{
  // текущая корзина хранится в виде файлов cookie
  string cartCookie = Request.Cookies["nw_cart"] ?? string.Empty;

  // если пользователь нажал кнопку Add to Cart (Добавить в корзину)
  if (id.HasValue)
  {
    if (string.IsNullOrWhiteSpace(cartCookie))
    {
      cartCookie = id.ToString();
    }
    else
    {
      string[] ids = cartCookie.Split('-');

      if (!ids.Contains(id.ToString()))
      {
        cartCookie = string.Join('-', cartCookie, id.ToString());
      }
    }
    Response.Cookies.Append("nw_cart", cartCookie);
  }

  var model = new HomeCartViewModel
  {
```

```
    Cart = new Cart
    {
      Items = Enumerable.Empty<CartItem>()
    },
    Recommendations = new List<EnrichedRecommendation>()
  };

  if (cartCookie.Length > 0)
  {
    model.Cart.Items = cartCookie.Split('-').Select(item =>
      new CartItem
      {
        ProductID = long.Parse(item),
        ProductName = db.Products.Find(long.Parse(item)).ProductName
      });
  }

  if (System.IO.File.Exists(GetDataPath("germany-model.zip")))
  {
    var mlContext = new MLContext();
    ITransformer modelGermany;

    using (var stream = new FileStream(
      path: GetDataPath("germany-model.zip"),
      mode: FileMode.Open,
      access: FileAccess.Read,
      share: FileShare.Read))
    {
      modelGermany = mlContext.Model.Load(stream,
        out DataViewSchema schema);
    }

    var predictionEngine = mlContext.Model.CreatePredictionEngine
      <ProductCobought, Recommendation>(modelGermany);

    var products = db.Products.ToArray();

    foreach (var item in model.Cart.Items)
    {
      var topThree = products.Select(product =>
        predictionEngine.Predict(
          new ProductCobought
          {
            ProductID = (uint)item.ProductID,
            CoboughtProductID = (uint)product.ProductID
          })
        ) // возвращает IEnumerable<Recommendation>
        .OrderByDescending(x => x.Score)
        .Take(3)
        .ToArray();
```

```
        model.Recommendations.AddRange(topThree
          .Select(rec => new EnrichedRecommendation
          {
            CoboughtProductID = rec.CoboughtProductID,
            Score = rec.Score,
            ProductName = db.Products.Find(
              (long)rec.CoboughtProductID).ProductName
          }));
      }

      // отображение трех лучших рекомендаций по покупке товара
      model.Recommendations = model.Recommendations
        .OrderByDescending(rec => rec.Score)
        .Take(3)
        .ToList();
    }
    return View(model);
  }
```

2. В папке Views\Home найдите и откройте файл `Index.cshtml`. Отредактируйте его код, чтобы вывести его модель представления. Включите ссылки для генерации наборов данных и обучения моделей:

```
@using Packt.Shared
@model HomeIndexViewModel
@{
  ViewData["Title"] = "Products - Northwind ML";
}
<h1 class="display-3">@ViewData["Title"]</h1>
<p class="lead">
<div>See product recommendations in your shopping cart.</div>
<ol>
  <li>First,
    <a asp-controller="Home"
      asp-action="GenerateDatasets">
      generate some datasets</a>.</li>
  <li>Second,
    <a asp-controller="Home" asp-action="TrainModels">
      train the models</a>.</li>
  <li>Third, add some products to your
    <a asp-controller="Home" asp-action="Cart">cart</a>.</li>
</ol>
<div>
@if (Model.GermanyDatasetExists || Model.UKDatasetExists)
{
  <text>Datasets for training:</text>
}
@if (Model.GermanyDatasetExists)
{
  <a href="/Data/germany-dataset.txt"
    class="btn btn-outline-primary">Germany</a>
```

```
}
@if (Model.UKDatasetExists)
{
  <a href="/Data/uk-dataset.txt"
    class="btn btn-outline-primary">UK</a>
}
@if (Model.USADatasetExists)
{
  <text>Dataset for testing:</text>
  <a href="/Data/usa-dataset.txt"
    class="btn btn-outline-primary">USA</a>
}
</div>
@if (Model.Milliseconds > 0)
{
<hr />
<div class="alert alert-success">
  It took @Model.Milliseconds milliseconds to train the models.
</div>
}
</p>
<h2>Products</h2>
@foreach (Category category in Model.Categories)
{
<h3>@category.CategoryName <small>@category.Description</small></h3>
<table>
<tbody>
@foreach (Product product in category.Products)
{
  <tr>
    <td>
      <a asp-controller="Home" asp-action="Cart"
        asp-route-id="@product.ProductID"
        class="btn btn-outline-success">Add to Cart</a>
    </td>
    <td>
      @product.ProductName
    </td>
  </tr>
}
</tbody>
</table>
}
```

3. В папке Views\Home создайте Razor-файл `Cart.cshtml` и отредактируйте для вывода его модели представления:

```
@model HomeCartViewModel
@{
  ViewData["Title"] = "Shopping Cart - Northwind ML";
}
```

```
<h1>@ViewData["Title"]</h1>
<table class="table table-bordered">
  <thead>
    <tr>
      <th>Product ID</th>
      <th>Product Name</th>
    </tr>
  </thead>
<tbody>
@foreach (CartItem item in Model.Cart.Items)
{
  <tr>
    <td>@item.ProductID</td>
    <td>@item.ProductName</td>
  </tr>
}
</tbody>
</table>
<h2>Customers who bought items in your cart also bought the following
products</h2>
@if (Model.Recommendations.Count() == 0)
{
<div>No recommendations.</div>
}
else
{
<table class="table table-bordered">
  <thead>
    <tr>
      <th></th>
      <th>Co-bought Product</th>
      <th>Score</th>
    </tr>
  </thead>
  <tbody>
  @foreach (EnrichedRecommendation rec in Model.Recommendations)
  {
    <tr>
      <td>
        <a asp-controller="Home" asp-action="Cart"
           asp-route-id="@rec.CoboughtProductID"
           class="btn btn-outline-success">Add to Cart</a>
      </td>
      <td>
        @rec.ProductName
      </td>
      <td>
        @rec.Score
      </td>
    </tr>
```

```
    }
  </tbody>
</table>
}
```

Тестирование сайта с рекомендациями по использованию товара

Теперь мы готовы протестировать функцию рекомендаций на сайте.

1. Запустите сайт с помощью следующей команды:

```
dotnet run
```

2. Запустите браузер Google Chrome. В адресной строке введите следующий URL: https://localhost:5001/.
3. На главной странице щелкните на ссылке Generate some datasets (Создать несколько наборов данных). Обратите внимание, что созданы ссылки на три набора данных (рис. 19.2).

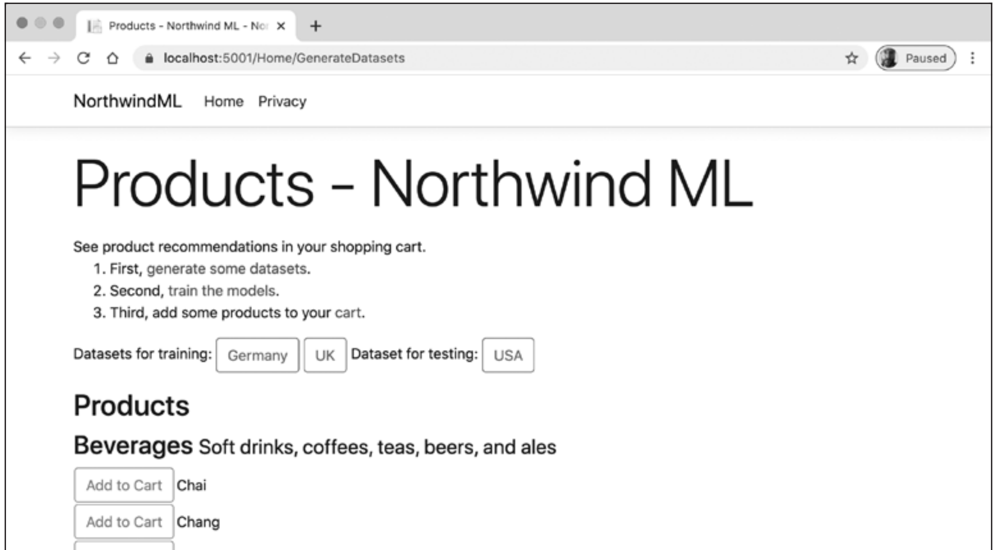


Рис. 19.2. Страница создания наборов данных

4. Щелкните кнопкой мыши на наборе данных UK (Великобритания). Вы должны увидеть, что пять товаров были куплены совместно с Product ID 1: 5, 11, 23, 68 и 69 (рис. 19.3).

Рис. 19.3. Набор данных Великобритании

5. Вернитесь в браузер.
6. Щелкните на ссылке train the models (обучить модели). Обратите внимание, сколько времени понадобилось для обучения моделей.
7. В программе Visual Studio Code в проекте NorthwindML разверните папку wwwroot, затем папку Data. Обратите внимание на модели в файловой системе, сохраненные в виде двоичных ZIP-файлов.
8. В браузере Google Chrome на главной странице Products — Northwind ML (Товары — Northwind ML) прокрутите вниз список товаров, нажмите кнопку Add to Cart (Добавить в корзину) рядом с любым товаром, например Outback Lager. Обратите внимание: на странице Shopping Cart (Корзина покупок) отображаются рекомендуемые товары с оценками (рис. 19.4).

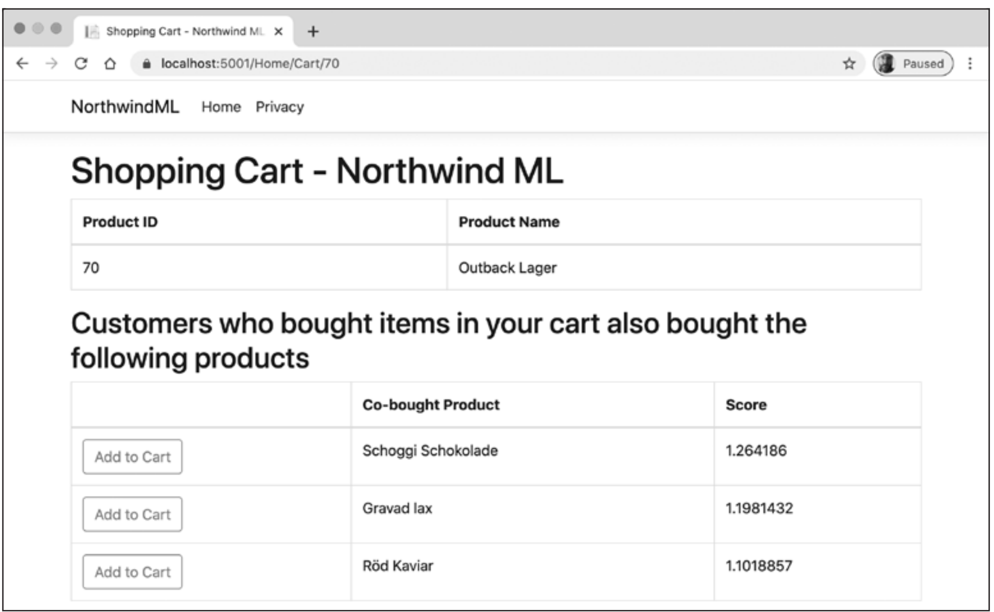


Рис. 19.4. Страница Shopping Cart (Корзина)

9. Вернитесь в браузер, добавьте в свою корзину еще один товар, например Ipoh Coffee. Обратите внимание на изменения в списке с тремя основными рекомендациями из-за того, что вероятность совместной покупки NuNuCa Nuß-Nougat-Creme выше, чем для предыдущей первой рекомендации, Schoggi Schokolade (рис. 19.5).
10. Закройте браузер и остановите работу сайта.

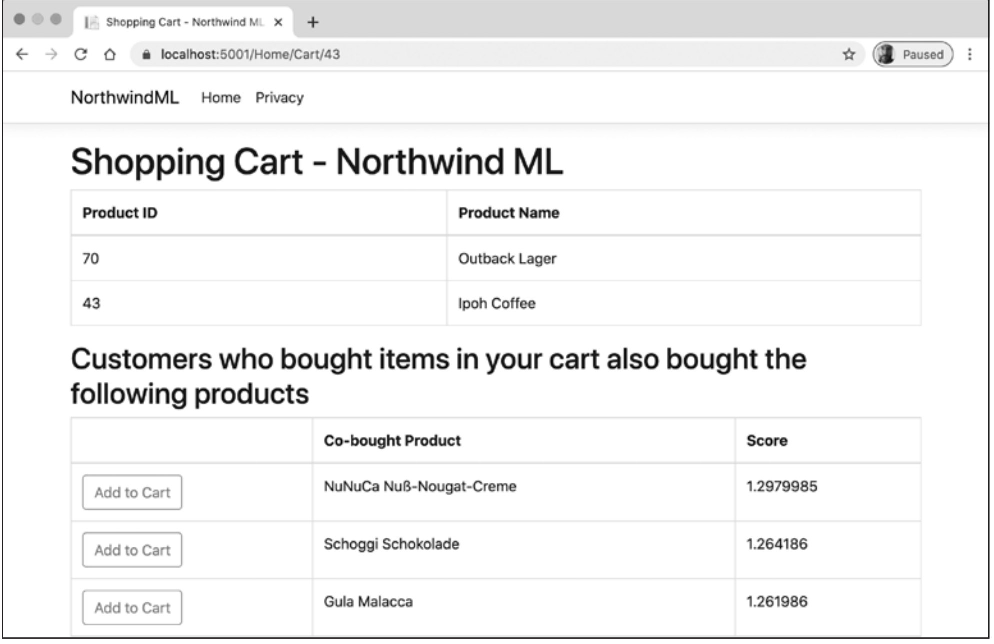


Рис. 19.5. Из трех основных товаров два были изменены

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 19.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Каковы четыре основных этапа жизненного цикла машинного обучения?
2. Каковы три подэтапа шага моделирования?

3. Почему модели необходимо перетренировать после развертывания?
4. Почему вы должны разделить свой набор данных на два: для обучения и для тестирования?
5. В чем разница между кластеризацией и классификацией задач машинного обучения?
6. Какой класс вы должны создать, чтобы выполнить любую задачу машинного обучения?
7. В чем разница между меткой и характеристикой?
8. Что представляет собой интерфейс `IDataView`?
9. Что представляет собой параметр `count` в атрибуте `[KeyType(count: 10)]`?
10. Что означает оценка с матричной факторизацией?

Упражнение 19.2. Примеры

В Microsoft существует большое количество примеров обучающих проектов для изучения ML.NET:

- анализ настроений для отзывов пользователей: https://github.com/dotnet/machinelearning-samples/tree/master/samples/csharp/getting-started/BinaryClassification_SentimentAnalysis;
- сегментация клиентов — пример кластеризации: https://github.com/dotnet/machinelearning-samples/tree/master/samples/csharp/getting-started/Clustering_CustomerSegmentation;
- обнаружение спама для текстовых сообщений: https://github.com/dotnet/machinelearning-samples/tree/master/samples/csharp/getting-started/BinaryClassification_SpamDetection;
- установка меток на задачи в GitHub: https://github.com/dotnet/machinelearning-samples/tree/master/samples/csharp/end-to-end-apps/MulticlassClassification-GitHubLabeler;
- рекомендации к фильмам — пример задачи матричной факторизации: https://github.com/dotnet/machinelearning-samples/tree/master/samples/csharp/getting-started/MatrixFactorization_MovieRecommendation;
- прогноз тарифа на такси: https://github.com/dotnet/machinelearning-samples/tree/master/samples/csharp/getting-started/Regression_TaxiFarePrediction;
- спрос на велосипеды — пример задачи регрессии: https://github.com/dotnet/machinelearning-samples/tree/master/samples/csharp/getting-started/Regression_BikeSharingDemand;
- eShopDashboardML — прогнозирование продаж: https://github.com/dotnet/machinelearning-samples/tree/master/samples/csharp/end-to-end-apps/Forecasting-Sales.

Упражнение 19.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- обновления API и инструментов ML.NET, август 2020 года: https://devblogs.microsoft.com/dotnet/august-ml-net-api-and-tooling-updates/;
- что такое ML.NET и как я понимаю основы машинного обучения: https://docs.microsoft.com/ru-ru/dotnet/machine-learning/how-does-mldotnet-work;
- машинное обучение: глоссарий важных терминов: https://docs.microsoft.com/ru-ru/dotnet/machine-learning/resources/glossary;
- набор видео по ML.NET от Channel 9: https://aka.ms/dotnet3-mlnet;
- набор видео по ML.NET на YouTube: https://aka.ms/mlnetyoutube;
- объяснимость и интерпретируемость машинного обучения: две концепции, которые могут помочь восстановить доверие к ИИ: https://www.kdnuggets.com/2018/12/machine-learning-explainability-interpretability-ai.html;
- задачи машинного обучения в ML.NET: https://docs.microsoft.com/ru-ru/dotnet/machine-learning/resources/tasks;
- преобразование данных машинного обучения — ML.NET: https://docs.microsoft.com/ru-ru/dotnet/machine-learning/resources/transforms;
- примеры ML.NET: https://github.com/dotnet/machinelearning-samples/blob/master/README.md;
- примеры от сообщества: https://github.com/dotnet/machinelearning-samples/blob/master/docs/COMMUNITY-SAMPLES.md;
- справочник по API ML.NET: https://docs.microsoft.com/en-gb/dotnet/api/?view=ml-dotnet;
- ML.NET: механизм машинного обучения для разработчиков .NET: https://msdn.microsoft.com/ru-ru/magazine/mt848634;
- создание механизма рекомендаций для приложений .NET с помощью машинного обучения в Azure: https://devblogs.microsoft.com/dotnet/dot-net-recommendation-system-for-net-applications-using-azure-machine-learning/.

Резюме

Вы познакомились с практическим примером применения интеллектуальной обработки данных на сайтах с помощью ML.NET.

Текущее решение не будет хорошо масштабироваться, поскольку в данный момент загружает весь список товаров в память. Встраивание подобного интеллекта

в приложения — это больше, чем профессия. Я надеюсь, текущая глава либо вызвала интерес к более глубокому изучению темы машинного обучения и науки о данных, либо показала вам достаточно, чтобы вы могли осознанно выбрать для себя другие области разработки на C# и .NET.

Далее вы научитесь создавать пользовательские веб-интерфейсы с помощью Blazor, новой технологии веб-компонентов от Microsoft, которая позволяет веб-разработчикам создавать клиентские одностраничные приложения (SPA) с использованием C# вместо JavaScript.

20 Создание пользовательских веб-интерфейсов с помощью Blazor

В этой главе рассказывается об использовании Microsoft Blazor для создания пользовательских интерфейсов для Интернета.

Я расскажу, что такое Blazor, опишу его преимущества и недостатки. Вы узнаете, как создавать компоненты Blazor, которые могут выполнять свой код на веб-сервере или в браузере. При размещении веб-интерфейсов на сервере Blazor, для обеспечения обновлений пользовательского интерфейса в браузере используется SignalR. При размещении с Blazor WebAssembly компоненты выполняют свой код на стороне клиента и для взаимодействия с сервером должны выполнять HTTP-вызовы.

В этой главе:

- знакомство с Blazor;
- сборка компонентов с помощью Blazor Server;
- создание компонентов с использованием Blazor WebAssembly.

Знакомство с Blazor

В главе 14 я познакомил вас с Blazor (и SignalR, который используется Blazor Server). Blazor поддерживается всеми современными браузерами.

С перечнем поддерживаемых платформ вы можете ознакомиться на сайте https://docs.microsoft.com/ru-ru/aspnet/core/blazor/supported-platforms.

Модели хостинга Blazor

Напомню, что Blazor — это единая модель приложения с двумя основными моделями хостинга.

- Blazor Server работает на стороне сервера, поэтому код C#, который вы пишете, имеет полный доступ ко всем ресурсам, которые могут понадобиться вашей

бизнес-логике, без необходимости аутентификации. Затем Blazor Server использует SignalR для передачи обновлений пользовательского интерфейса клиентской стороне. Сервер должен поддерживать активное соединение SignalR с каждым клиентом и отслеживать текущее состояние каждого клиента, поэтому Blazor Server плохо масштабируется при поддержке большого количества клиентов. Впервые он был выпущен в составе .NET Core 3.0 в сентябре 2019 года и включен в .NET 5.0 и более поздние версии.

- Blazor WebAssembly работает на стороне клиента, поэтому ваш код C# имеет доступ только к ресурсам в браузере и должен выполнять HTTP-вызовы (которые могут потребовать аутентификации), прежде чем сможет получить доступ к ресурсам на сервере. Впервые он был выпущен в мае 2020 года как расширение для .NET Core 3.1 в версии 3.2, поскольку это текущий выпуск и, следовательно, не предусмотрена долгосрочная поддержка .NET Core 3.1. Версия .NET Core 3.2 использовала библиотеки Mono runtime и Mono; версия .NET 5 использует среду выполнения Mono и библиотеки .NET 5. Как сказал Дэниел Рот, *«Blazor WebAssembly работает на интерпретаторе .NET IL без JIT, поэтому он не дает преимуществ в скорости. Мы добились значительных улучшений скорости в .NET 5, и мы надеемся улучшить ситуацию в .NET 6»*.

Хотя Blazor Server поддерживается в Internet Explorer 11, Blazor WebAssembly — нет.

Blazor WebAssembly имеет дополнительную поддержку для прогрессивных веб-приложений (PWA), что означает, что пользователь сайта может использовать меню браузера, чтобы добавить приложение на свой рабочий стол и запустить приложение на компьютере.

Подробнее о моделях хостинга вы можете узнать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/blazor/hosting-models.

Компоненты Blazor

Важно понимать, что Blazor используется для создания компонентов пользовательского интерфейса. Компоненты определяют, как визуализировать пользовательский интерфейс, реагировать на пользовательские события, и могут быть составлены, вложены и скомпилированы в библиотеку классов NuGet Razor для упаковки и распространения.

В будущем Blazor может не ограничиваться только созданием компонентов пользовательского интерфейса с использованием веб-технологий. У Microsoft есть экспериментальная технология, известная как Blazor Mobile Bindings, позволяющая разработчикам использовать Blazor для создания компонентов мобильного пользовательского интерфейса. Вместо использования HTML и CSS для создания

пользовательского веб-интерфейса Blazor использует XAML и Xamarin.Forms для создания кросс-платформенного мобильного пользовательского интерфейса.

Более подробную информацию о Blazor Mobile Bindings вы найдете на сайте https://devblogs.microsoft.com/aspnet/mobile-blazor-bindings-experiment/.

Microsoft также экспериментирует с гибридной моделью, которая позволяет создавать приложения как для Интернета, так и для мобильных устройств.

Более подробную информацию о приложениях Blazor Hybrid вы найдете на сайте https://devblogs.microsoft.com/aspnet/hybrid-blazor-apps-in-mobile-blazor-bindings-july-update/.

Blazor и Razor

Вы можете задаться вопросом, почему компоненты Blazor используют файловое расширение `.razor`. Razor — это синтаксис разметки шаблонов, позволяющий одновременно использовать HTML и C#. В более старых технологиях, поддерживающих Razor, расширение файла `.cshtml` используется для обозначения сочетания C# и HTML.

Razor используется для:

- *представлений* ASP.NET Core MVC и *частичных представлений* с расширением `.cshtml`. Бизнес-логика может быть абстрагирована в класс контроллера, рассматривающий представление в качестве шаблона для отправки модели представления с последующим выводом его на веб-страницу;
- *страниц Razor* с расширением `.cshtml`. Бизнес-логика может быть встроенной или содержать файл с расширением `.cshtml.cs`. Результат — это веб-страница;
- *компонентов Blazor* с расширением `.razor`. Результатом вывода не является веб-страница, хотя шаблоны могут использоваться для обертывания компонента, чтобы он отображался как веб-страница, а директива `@page` будет использоваться для назначения пути, определяющего URL-адрес для извлечения компонента как страницы.

Сравнение шаблонов проектов Blazor

Один из способов понять, какой выбор между моделями размещения Blazor Server и Blazor WebAssembly — проанализировать различия в их шаблонах проектов по умолчанию.

Обзор шаблона проекта Blazor Server

Рассмотрим шаблон по умолчанию для проекта Blazor Server. Вы заметите, что это то же самое, что и шаблон ASP.NET Core Razor Pages, только с несколькими ключевыми дополнениями.

1. В папке PracticalApps создайте папку NorthwindBlazorServer.
2. В программе Visual Studio Code откройте рабочую область `PracticalApps` и добавьте папку `NorthwindBlazorServer`.
3. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал), а затем выберите папку `NorthwindBlazorServer`.
4. На панели TERMINAL (Терминал) используйте шаблон `blazorserver` для создания нового проекта Blazor Server:

   ```
   dotnet new blazorserver
   ```

5. Выберите `NorthwindBlazorServer` в качестве активного проекта `OmniSharp`.
6. В папке `NorthwindBlazorServer` найдите и откройте файл `NorthwindBlazorServer.csproj`. Обратите внимание, что он идентичен проекту ASP.NET Core, который использует веб-пакет SDK и платформу .NET 5.0.
7. Найдите и откройте файл `Program.cs`. Обратите внимание, что он идентичен проекту ASP.NET Core.
8. Найдите и откройте файл `Startup.cs`. Обратите внимание на метод `ConfigureServices`, содержащий вызов метода `AddServerSideBlazor`, как показано ниже (выделено полужирным шрифтом):

   ```
   public void ConfigureServices(IServiceCollection services)
   {
     services.AddRazorPages();
     services.AddServerSideBlazor();
     services.AddSingleton<WeatherForecastService>();
   }
   ```

9. Обратите внимание на метод `Configure`, похожий на проект Razor Pages ASP.NET Core. Отличие заключается в вызове методов `MapBlazorHub` и `MapFallbackToPage` при настройке конечных точек, настраивающих приложение ASP.NET Core для приема входящих подключений SignalR для компонентов Blazor, а другие запросы возвращаются к странице Razor с именем `_Host.cshtml`, как показано ниже:

   ```
   app.UseEndpoints(endpoints =>
   {
     endpoints.MapBlazorHub();
     endpoints.MapFallbackToPage("/_Host");
   });
   ```

10. В папке `Pages` найдите и откройте файл `_Host.cshtml`, содержимое которого показано ниже:

```
@page "/"
@namespace NorthwindBlazorServer.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{
  Layout = null;
}

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport"
        content="width=device-width, initial-scale=1.0" />
  <title>NorthwindBlazorServer</title>
  <base href="~/" />
  <link rel="stylesheet"
        href="css/bootstrap/bootstrap.min.css" />
  <link href="css/site.css" rel="stylesheet" />
  <link href="_content/NorthwindBlazorServer/_framework/scoped.styles.css"
        rel="stylesheet" />
</head>
<body>
  <component type="typeof(App)"
             render-mode="ServerPrerendered" />

  <div id="blazor-error-ui">
    <environment include="Staging,Production">
      An error has occurred. This application may no longer respond until
      reloaded.
    </environment>
    <environment include="Development">
      An unhandled exception has occurred. See browser dev tools for details.
    </environment>
    <a href="" class="reload">Reload</a>
 <a class="dismiss">✕</a>
  </div>

  <script src="_framework/blazor.server.js"></script>
</body>
</html>
```

При просмотре предыдущего кода обратите внимание на следующее.

- В теге `<body>` компонент Blazor типа `App`, который предварительно отображается на сервере.
- Код `<div id="blazor-error-ui">` позволяет отобразить ошибки Blazor, которые отображаются в виде желтой полосы внизу веб-страницы.

- Блок скрипта для `blazor.server.js` управляет подключением SignalR к серверу.

11. В папке `NorthwindBlazorServer` найдите и откройте файл `App.razor`. Обратите внимание, что файл определяет путь для всех компонентов, найденных в текущей сборке:

```
<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData"
               DefaultLayout="@typeof(MainLayout)" />
  </Found>
  <NotFound>
    <LayoutView Layout="@typeof(MainLayout)">
      <p>Sorry, there's nothing at this address.</p>
    </LayoutView>
  </NotFound>
</Router>
```

При просмотре предыдущего кода обратите внимание на следующее.

- Если соответствующий путь найден, то выполняется метод `RouteView`, который устанавливает макет по умолчанию для компонента в `MainLayout` и передает любые параметры данных пути компоненту.
- Если соответствующий путь не найден, то выполняется метод `LayoutView`, который выводит внутреннюю разметку (в данном случае простой элемент абзаца с сообщением, что по данному адресу ничего нет) внутри `MainLayout`.

12. В папке Shared найдите и откройте файл `MainLayout.razor`. Обратите внимание, что он определяет тег `<div>` для боковой панели, содержащей меню навигации, и тег `<div>` для основного содержимого:

```
@inherits LayoutComponentBase

<div class="page">
  <div class="sidebar">
    <NavMenu />
  </div>

  <div class="main">
    <div class="top-row px-4">
      <a href="https://docs.microsoft.com/aspnet/"
        target="_blank">About</a>
    </div>
    <div class="content px-4">
      @Body
    </div>
  </div>
</div>
```

13. В папке Shared найдите и откройте файл `MainLayout.razor.css`. Обратите внимание, что он содержит изолированные стили CSS для каждого компонента.
14. В папке Shared найдите и откройте файл `NavMenu.razor`. Обратите внимание, что файл содержит три пункта меню: `Home`, `Counter` и `Fetch data`. Позже мы вернемся к этой теме, когда добавим наш собственный компонент.
15. В папке `Pages` найдите и откройте файл `FetchData.razor`. Обратите внимание, что он определяет компонент, который извлекает прогноз погоды из внедренной зависимой службы, а затем отображает данные прогноза погоды в таблице, как показано ниже:

```
@page "/fetchdata"

@using NorthwindBlazorServer.Data
@inject WeatherForecastService ForecastService

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

@if (forecasts == null)
{
  <p><em>Loading...</em></p>
}
else
{
  <table class="table">
    <thead>
      <tr>
        <th>Date</th>
        <th>Temp. (C)</th>
        <th>Temp. (F)</th>
        <th>Summary</th>
      </tr>
    </thead>
    <tbody>
      @foreach (var forecast in forecasts)
      {
        <tr>
          <td>@forecast.Date.ToShortDateString()</td>
          <td>@forecast.TemperatureC</td>
          <td>@forecast.TemperatureF</td>
          <td>@forecast.Summary</td>
        </tr>
      }
    </tbody>
  </table>
}
```

```
@code {
  private WeatherForecast[] forecasts;

  protected override async Task OnInitializedAsync()
  {
    forecasts = await ForecastService
      .GetForecastAsync(DateTime.Now);
  }
}
```

16. В папке `Data` найдите и откройте файл `WeatherForecastService.cs`. Обратите внимание, что это не класс контроллера веб-API, это просто обычный класс, возвращающий случайные данные о погоде:

```
using System;
using System.Linq;
using System.Threading.Tasks;

namespace NorthwindBlazorServer.Data
{
  public class WeatherForecastService
  {
    private static readonly string[] Summaries = new[]
    {
      "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",
      "Hot", "Sweltering", "Scorching"
    };

    public Task<WeatherForecast[]> GetForecastAsync(
      DateTime startDate)
    {
      var rng = new Random();
      return Task.FromResult(
        Enumerable.Range(1, 5)
        .Select(index => new WeatherForecast
      {
        Date = startDate.AddDays(index),
        TemperatureC = rng.Next(-20, 55),
        Summary = Summaries[rng.Next(Summaries.Length)]
      }).ToArray());
    }
  }
}
```

CSS-изоляция

Для стилизации компоненты Blazor часто нуждаются в собственном коде CSS. Для обеспечения отсутствия конфликтов с CSS на уровне сайта Blazor поддерживает технологию CSS-изоляции. Если у вас есть компонент `Index.razor`, создайте файл CSS с именем `Index.razor.css`.

Более подробно о необходимости CSS-изоляции для компонентов Blazor вы можете узнать на сайте: https://github.com/dotnet/aspnetcore/issues/10170.

Запуск шаблона проекта Blazor Server

Теперь, когда мы рассмотрели шаблон проекта и его важные компоненты, характерные для Blazor Server, мы можем запустить сайт и проанализировать результаты.

1. На панели TERMINAL (Терминал) введите команду для запуска сайта, как показано ниже:

   ```
   dotnet run
   ```

2. Запустите браузер, перейдите по адресу https://localhost:5001/ и щелкните на ссылке Fetch data (Получить данные) (рис. 21.1).

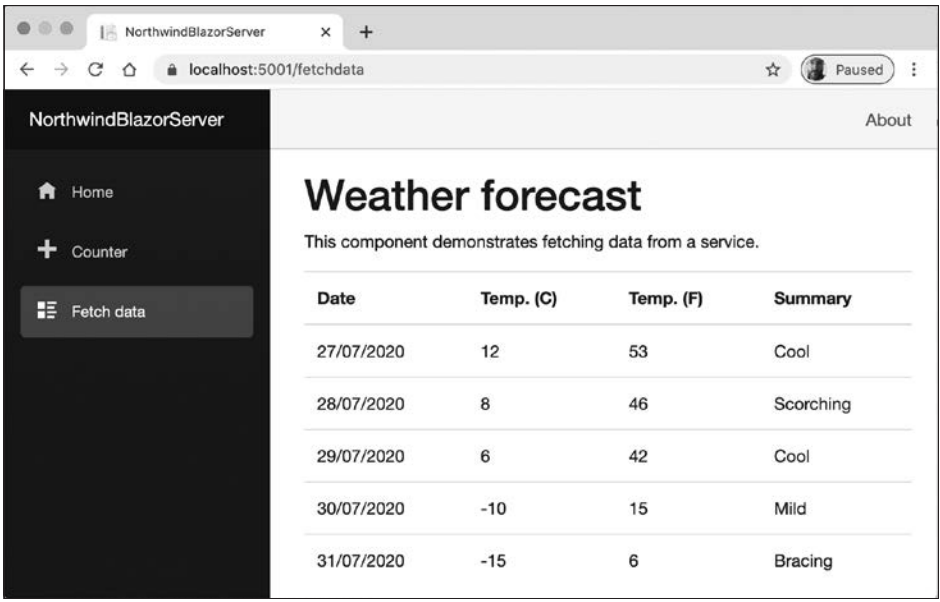


Рис. 20.1. Получение данных о погоде

3. Измените путь на `/apples`. Обратите внимание на отсутствующее сообщение (рис. 20.2).
4. Закройте браузер Google Chrome.
5. Чтобы остановить веб-сервер, в программе Visual Studio Code на панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C.

Рис. 20.2. Сообщение об отсутствии компонента

Обзор шаблона проекта Blazor WebAssembly

Создадим проект Blazor WebAssembly. Я не буду публиковать в книге код, если он такой же, как в проекте Blazor Server.

1. В папке PracticalApps создайте папку NorthwindBlazorWasm.
2. В программе Visual Studio Code откройте рабочую область `PracticalApps` и добавьте папку NorthwindBlazorWasm.
3. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и папку NorthwindBlazorWasm.
4. На панели TERMINAL (Терминал) используйте шаблон `blazorwasm` с флагами `--pwa` и `--hosted`, чтобы создать новый проект Blazor WebAssembly, размещенный в ASP.NET Core, поддерживающий функцию PWA вашей операционной системы:

   ```
   dotnet new blazorwasm --pwa --hosted
   ```

   При просмотре кода сгенерированного проекта обратите внимание на следующее.

   - Создаются решение и три папки проекта: Client, Server и Shared.
   - Папка Shared — это библиотека классов, содержащая модели для метеослужбы.
   - Папка Server — это сайт ASP.NET Core для размещения метеослужбы. Этот сайт содержит такую же реализацию для возврата случайных данных прогноза погоды, что и ранее, однако реализован как соответствующий класс контроллера веб-API. Файл проекта содержит ссылки на проект Shared и Client, а также ссылку на пакет для поддержки WebAssembly на стороне сервера.
   - Папка Client — проект Blazor WebAssembly.
5. В папке `Client` найдите и откройте файл `NorthwindBlazorWasm.Client.csproj`. Обратите внимание, что файл использует пакет SDK Blazor WebAssembly и со-

держит три ссылки на пакеты, а также использует Service Worker, необходимый для поддержки PWA:

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <ServiceWorkerAssetsManifest>service-worker-assets.js</
ServiceWorkerAssetsManifest>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.AspNetCore.Components.WebAssembly"
      Version="5.0.0" />
    <PackageReference
      Include="Microsoft.AspNetCore.Components
        .WebAssembly.DevServer"
      Version="5.0.0" PrivateAssets="all" />
    <PackageReference
      Include="System.Net.Http.Json"
      Version="5.0.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include=
      "..\Shared\NorthwindBlazorWasm.Shared.csproj" />
  </ItemGroup>

  <ItemGroup>
    <ServiceWorker Include=
      "wwwroot\service-worker.js" PublishedContent=
      "wwwroot\service-worker.published.js" />
  </ItemGroup>

</Project>
```

6. В папке `Client` найдите и откройте файл `Program.cs`. Обратите внимание, что разработчик хоста предназначен для WebAssembly, а не для серверного ASP.NET Core и он регистрирует службу зависимостей для выполнения HTTP-запросов, что является чрезвычайно распространенным требованием для приложений Blazor WebAssembly:

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("#app");

builder.Services.AddScoped(sp => new HttpClient
  { BaseAddress = new Uri(
    builder.HostEnvironment.BaseAddress) });

await builder.Build().RunAsync();
```

7. В папке `wwwroot` найдите и откройте файл `index.html`. Обратите внимание на файлы `manifest.json` и `service-worker.js` для поддержки работы с флеш-файлами, а также на сценарий `blazor.webassembly.js`, который загружает все пакеты NuGet для Blazor WebAssembly, как показано ниже (выделено полужирным шрифтом):

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0,
   maximum-scale=1.0, user-scalable=no" />
  <title>NorthwindBlazorWasm</title>
  <base href="/" />
  <link href="css/bootstrap/bootstrap.min.css"
        rel="stylesheet" />
  <link href="css/app.css" rel="stylesheet" />
  <link href="_framework/scoped.styles.css"
        rel="stylesheet" />
  <link href="manifest.json" rel="manifest" />
  <link rel="apple-touch-icon" sizes="512x512"
        href="icon-512.png" />
</head>

<body>
  <div id="app">Loading...</div>

  <div id="blazor-error-ui">
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
 <a class="dismiss">✕</a>
  </div>
  <script src="_framework/blazor.webassembly.js"></script>
  <script>navigator.serviceWorker
    .register('service-worker.js');</script>
</body>

</html>
```

8. Обратите внимание, что в папке Client следующие файлы идентичны Blazor Server: `App.razor`, `Shared\MainLayout.razor`, `Shared\NavMenu.razor`, `SurveyPrompt.razor`, `Pages\Counter.razor` и `Pages\Index.razor`.

9. В папке Pages найдите и откройте файл `FetchData.razor`. Обратите внимание, что код аналогичен Blazor Server, за исключением внедренной службы зависимостей для выполнения HTTP-запросов:

```
@page "/fetchdata"
@using NorthwindBlazorWasm.Shared
```

```
@inject HttpClient Http

<h1>Weather forecast</h1>

...

@code {
  private WeatherForecast[] forecasts;

  protected override async Task OnInitializedAsync()
  {
    forecasts = await
      Http.GetFromJsonAsync<WeatherForecast[]>(
        "WeatherForecast");
  }
}
```

10. Запустите серверный проект:

```
cd Server
dotnet run
```

11. Обратите внимание, что приложение имеет те же функции, что и раньше, но код выполняется внутри браузера, а не на сервере.
12. Закройте браузер Google Chrome.
13. Чтобы остановить веб-сервер, в программе Visual Studio Code на панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C.

Сборка компонентов с помощью Blazor Server

В данном разделе создадим компонент для составления списка, создания и редактирования клиентов в базе данных `Northwind`.

Определение и тестирование простого компонента

В существующий проект Blazor Server добавим новый компонент.

1. В проекте `NorthwindBlazorServer` добавьте в папку `Pages` новый файл `Customers.razor`.

Имена файлов компонентов должны начинаться с заглавной буквы. В противном случае будут обнаружены ошибки компиляции!

2. Добавьте операторы для регистрации `/customers` в качестве пути. Выведите заголовок для компонента клиентов и определите блок кода:

```
@page "/customers"

<h1>Customers</h1>

@code {

}
```

3. В папке `Shared` найдите и откройте файл `NavMenu.razor` и добавьте элемент списка для нашего нового компонента с меткой Customers (Клиенты), как показано ниже:

```
<li class="nav-item px-3">
  <NavLink class="nav-link" href="customers">
    <span class="oi oi-people"
      aria-hidden="true"></span> Customers
  </NavLink>
</li>
```

Доступные значки вы можете увидеть на сайте https://iconify.design/icon-sets/oi/.

4. Запустите проект сайта, перейдите к нему и щелкните на ссылке Customers (Клиенты) (рис. 20.3)

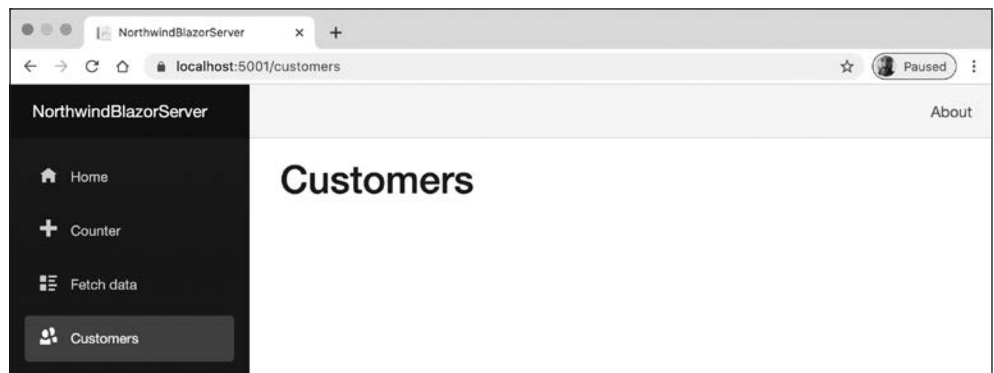


Рис. 20.3. Компонент Customers отображается в виде веб-страницы

5. Закройте браузер Google Chrome.
6. Чтобы остановить веб-сервер, в программе Visual Studio Code на панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C.

Получение сущностей и их компонентов

Теперь, когда вы ознакомились с минимальной реализацией компонента, мы можем добавить к нему некоторые полезные функции. В данном случае для извлечения клиентов из базы данных мы будем использовать контекст базы данных `Northwind`.

1. Найдите и откройте файл `NorthwindBlazorServer.csproj` и добавьте операторы для ссылки на проект контекста базы данных `Northwind`, как показано ниже (выделено полужирным шрифтом):

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include=
      "..\NorthwindContextLib\NorthwindContextLib.csproj" />
  </ItemGroup>

</Project>
```

2. На панели TERMINAL (Терминал) восстановите пакеты и скомпилируйте проект с помощью следующей команды:

```
dotnet build
```

3. Найдите и откройте файл `Startup.cs` и добавьте пространства имен `System.IO`, `Microsoft.EntityFrameworkCore` и `Packt.Shared`:

```
using Microsoft.EntityFrameworkCore;
using Packt.Shared;
using System.IO;
```

4. Добавьте в метод `ConfigureServices` следующий оператор. Это необходимо для того, чтобы зарегистрировать класс контекста базы данных Northwind для использования SQLite в качестве поставщика базы данных и указать строку подключения к базе данных:

```
string databasePath = Path.Combine("..", "Northwind.db");
services.AddDbContext<Northwind>(options =>
  options.UseSqlite($"Data Source={databasePath}"));
```

5. Найдите и откройте файл `_Imports.razor` и импортируйте пространства имен `NorthwindBlazorServer.Data`, `Microsoft.EntityFrameworkCore` и `Packt.Shared`, чтобы создаваемым нами компонентам Blazor не нужно было импортировать пространства имен по отдельности, как показано ниже (выделено полужирным шрифтом):

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
@using NorthwindBlazorServer
@using NorthwindBlazorServer.Shared
@using NorthwindBlazorServer.Data
@using Microsoft.EntityFrameworkCore
@using Packt.Shared
```

6. В папке Pages найдите и откройте файл `Customers.razor`, введите контекст базы данных `Northwind` и используйте его для вывода таблицы всех клиентов, как показано ниже:

```
@page "/customers"
@inject Northwind db

<h1>Customers</h1>
@if (customers == null)
{
  <p><em>Loading...</em></p>
}
else
{
  <table class="table">
    <thead>
      <tr>
        <th>ID</th>
        <th>Company Name</th>
        <th>Address</th>
        <th>Phone</th>
        <th></th>
      </tr>
    </thead>
    <tbody>
      @foreach (var customer in customers)
      {
      <tr>
        <td>@customer.CustomerID</td>
        <td>@customer.CompanyName</td>
        <td>@customer.Address<br/>
        @customer.City<br/>
        @customer.PostalCode<br/>
        @customer.Country</td>
        <td>@customer.Phone</td>
        <td>
```

```
          <a class="btn btn-info"
            href="editcustomer/@customer.CustomerID">
              <i class="oi oi-pencil"></i></a>
          <a class="btn btn-danger"
            href="deletecustomer/@customer.CustomerID">
              <i class="oi oi-trash"></i></a>
        </td>
      </tr>
      }
    </tbody>
  </table>
}

@code {
  private IEnumerable<Customer> customers;

  protected override async Task OnInitializedAsync()
  {
    customers = await db.Customers.ToListAsync();
  }
}
```

7. Чтобы запустить сайт, на панели TERMINAL (Терминал) введите следующую команду:

```
dotnet run
```

8. В Google Chrome введите адрес https://localhost:5001/ и щелкните на ссылке Customers (Клиенты). Обратите внимание на таблицу клиентов, загружаемых из базы данных и отображаемых на веб-странице (рис. 20.4).

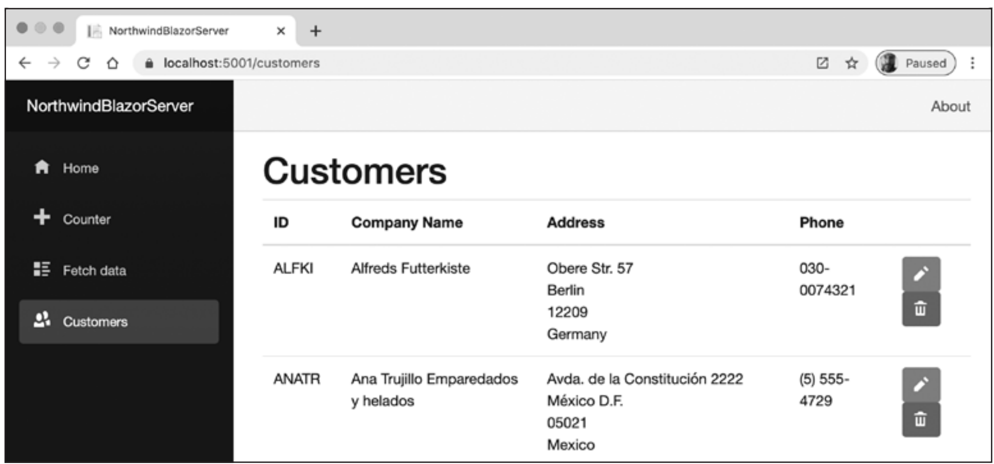


Рис. 20.4. Список клиентов

9. Закройте браузер Google Chrome.
10. Чтобы остановить веб-сервер, в программе Visual Studio Code на панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C.

Более подробно о настройке элементов <head> вы можете узнать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/blazor/fundamentals/additional-scenarios-influence-html-head-tag-elements.

Абстрагирование службы для компонента Blazor

В настоящее время для выборки клиентов компонент Blazor напрямую вызывает контекст базы данных Northwind. Это прекрасно работает в Blazor Server, поскольку компонент выполняется на сервере. Однако компонент Blazor не работал бы при размещении в Blazor WebAssembly.

Чтобы обеспечить повторное использование компонентов, создадим локальную службу зависимостей.

1. В папке `Data` добавьте новый файл `INorthwindService.cs` и измените его содержимое для определения контракта локальной службы, которая абстрагирует операции CRUD:

```
using System.Collections.Generic;
using System.Threading.Tasks;

namespace Packt.Shared
{
  public interface INorthwindService
  {
    Task<List<Customer>> GetCustomersAsync();
    Task<Customer> GetCustomerAsync(string id);
    Task<Customer> CreateCustomerAsync(Customer c);
    Task<Customer> UpdateCustomerAsync(Customer c);
    Task DeleteCustomerAsync(string id);
  }
}
```

2. В папке `Data` добавьте новый файл с именем `NorthwindService.cs` и измените его содержимое для реализации интерфейса `INorthwindService`, используя контекст базы данных `Northwind`:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Packt.Shared;

namespace NorthwindBlazorServer.Data
```

```
{
  public class NorthwindService : INorthwindService
  {
    private readonly Northwind db;

    public NorthwindService(Northwind db)
    {
      this.db = db;
    }

    public Task<List<Customer>> GetCustomersAsync()
    {
      return db.Customers.ToListAsync();
    }

    public Task<Customer> GetCustomerAsync(string id)
    {
      return db.Customers.FirstOrDefaultAsync
        (c => c.CustomerID == id);
    }

    public Task<Customer> CreateCustomerAsync(Customer c)
    {
      db.Customers.Add(c);
      db.SaveChangesAsync();
      return Task.FromResult<Customer>(c);
    }

    public Task<Customer> UpdateCustomerAsync(Customer c)
    {
      db.Entry(c).State = EntityState.Modified;
      db.SaveChangesAsync();
      return Task.FromResult<Customer>(c);
    }

    public Task DeleteCustomerAsync(string id)
    {
      Customer customer = db.Customers.FirstOrDefaultAsync
        (c => c.CustomerID == id).Result;
      db.Customers.Remove(customer);
      return db.SaveChangesAsync();
    }
  }
}
```

3. Найдите и откройте файл `Startup.cs`. В методе `ConfigureServices` добавьте операторы для регистрации `NorthwindService` в качестве временной службы, реализующей интерфейс `INorthwindService`:

```
services.AddTransient
  <INorthwindService, NorthwindService>();
```

4. В папке Pages найдите и откройте файл `Customers.razor`, удалите директиву для внедрения контекста базы данных `Northwind` и добавьте директиву для внедрения зарегистрированной службы `Northwind`:

   ```
   @inject INorthwindService service
   ```

5. Для вызова службы измените метод `OnInitializedAsync`:

   ```
   customers = await service.GetCustomersAsync();
   ```

6. При необходимости запустите проект сайта `NorthwindBlazorServer`, чтобы проверить, сохраняет ли он ту же функциональность, что и ранее.

Использование форм Blazor

Для построения форм Microsoft предоставляет готовые компоненты. Мы будем использовать их для предоставления, создания и редактирования функций для клиентов.

Определение форм с помощью компонента EditForm

Для упрощения использования форм Blazor Microsoft предоставляет компонент `EditForm` и элементы форм, например `InputText`.

Компонент `EditForm` может содержать модель, настроенную для привязки ее к объекту со свойствами и обработчиками событий для настраиваемой проверки, а также для распознавания стандартных атрибутов проверки Microsoft в классе модели, как показано ниже:

```
<EditForm Model="@customer" OnSubmit="ExtraValidation">
  <DataAnnotationsValidator />
  <ValidationSummary />
  <InputText id="name" @bind-Value="customer.CompanyName" />
  <button type="submit">Submit</button>
</EditForm>

@code {
  private Customer customer = new Customer();

  private void ExtraValidation()
  {
    // выполнить проверку
  }
}
```

Для отображения сообщения рядом с отдельным элементом формы в качестве альтернативы компоненту `ValidationSummary` можно использовать компонент `ValidationMessage`.

Подробнее об использовании NavigationManager с маршрутами Blazor можно узнать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/blazor/fundamentals/routing#uri-and-navigation-state-helpers.

Создание и использование компонента формы клиента

Теперь мы можем создать собственный компонент для введения или редактирования информации о клиенте.

1. В папке Pages создайте новый файл `CustomerDetail.razor` и измените его содержимое, чтобы определить форму для редактирования свойств клиента, как показано ниже:

```
<EditForm Model="@Customer" OnValidSubmit="@OnValidSubmit">
  <DataAnnotationsValidator />
  <div class="form-group">
    <div>
      <label>Customer ID</label>
      <div>
        <InputText @bind-Value="@Customer.CustomerID" />
        <ValidationMessage
          For="@(() => Customer.CustomerID)" />
      </div>
    </div>
  </div>
  <div class="form-group ">
    <div>
      <label>Company Name</label>
      <div>
        <InputText @bind-Value="@Customer.CompanyName" />
        <ValidationMessage
          For="@(() => Customer.CompanyName)" />
      </div>
    </div>
  </div>
  <div class="form-group ">
    <div>
      <label>Address</label>
      <div>
        <InputText @bind-Value="@Customer.Address" />
        <ValidationMessage
          For="@(() => Customer.Address)" />
      </div>
    </div>
```

```
    </div>
    <div class="form-group ">
      <div>
        <label>Country</label>
        <div>
          <InputText @bind-Value="@Customer.Country" />
          <ValidationMessage
            For="@(() => Customer.Country)" />
        </div>
      </div>
    </div>
    <button type="submit" class="btn btn-@ButtonStyle">
      @ButtonText
    </button>
</EditForm>

@code {
  [Parameter]
  public Customer Customer { get; set; }

  [Parameter]
  public string ButtonText { get; set; } = "Save Changes";

  [Parameter]
  public string ButtonStyle { get; set; } = "info";

  [Parameter]
  public EventCallback OnValidSubmit { get; set; }
}
```

2. В папке Pages создайте новый файл `CreateCustomer.razor` и измените его содержимое, чтобы использовать компонент сведений о клиенте для создания нового клиента:

```
@page "/createcustomer"
@inject INorthwindService service
@inject NavigationManager navigation

<h3>Create Customer</h3>
<CustomerDetail ButtonText="Create Customer"
                Customer="@customer"
                OnValidSubmit="@Create" />

@code {
  private Customer customer = new Customer();

  private async Task Create()
  {
    await service.CreateCustomerAsync(customer);
    navigation.NavigateTo("customers");
  }
}
```

3. В папке Pages найдите и откройте файл Customers.razor и после элемента <h1> добавьте элемент <div> с кнопкой для перехода к компоненту создания клиента, как показано ниже:

```
<div class="form-group">
  <a class="btn btn-info" href="createcustomer">
    <i class="oi oi-plus"></i> Create New</a>
</div>
```

4. В папке Pages создайте новый файл EditCustomer.razor и измените его содержимое, чтобы использовать компонент сведений о клиенте для редактирования и сохранения изменений для существующего клиента:

```
@page "/editcustomer/{customerid}"
@inject INorthwindService service
@inject NavigationManager navigation
<h3>Edit Customer</h3>
<CustomerDetail ButtonText="Update"
                Customer="@customer"
                OnValidSubmit="@Update" />

@code {
  [Parameter]
  public string CustomerID { get; set; }

  private Customer customer = new Customer();

  protected async override Task OnParametersSetAsync()
  {
    customer = await service.GetCustomerAsync(CustomerID);
  }

  private async Task Update()
  {
    await service.UpdateCustomerAsync(customer);
    navigation.NavigateTo("customers");
  }
}
```

5. В папке Pages создайте новый файл DeleteCustomer.razor и измените его содержимое, чтобы использовать компонент сведений о клиенте. Так можно показать клиента, которого мы собираемся удалить:

```
@page "/deletecustomer/{customerid}"
@inject INorthwindService service
@inject NavigationManager navigation
<h3>Delete Customer</h3>
<div class="alert alert-danger">
  Warning! This action cannot be undone!
</div>
<CustomerDetail ButtonText="Delete Customer"
```

```
                 ButtonStyle="danger"
                 Customer="@customer"
                 OnValidSubmit="@Delete" />

@code {
  [Parameter]
  public string CustomerID { get; set; }

  private Customer customer = new Customer();

  protected async override Task OnParametersSetAsync()
  {
    customer = await service.GetCustomerAsync(CustomerID);
  }

  private async Task Delete()
  {
    await service.DeleteCustomerAsync(CustomerID);
    navigation.NavigateTo("customers");
  }
}
```

6. Запустите проект сайта и перейдите по адресу https://localhost:5001/.
7. Перейдите к разделу Customers (Клиенты) и нажмите кнопку + Create New (+ Создать).
8. Введите недопустимый идентификатор клиента, например `ABCDEF`. Обратите внимание на сообщение проверки (рис. 20.5).

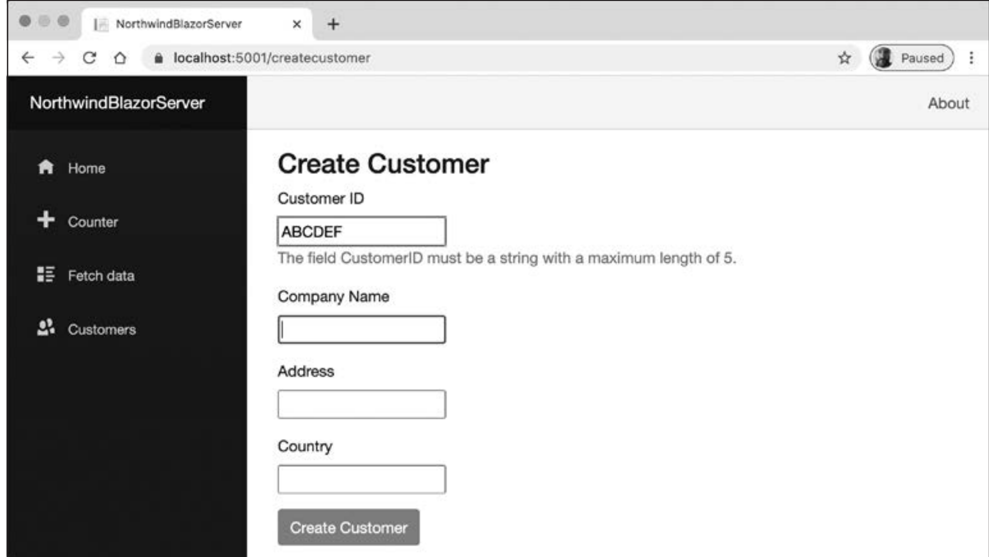


Рис. 20.5. Создание нового клиента и ввод неверного идентификатора клиента

9. Теперь измените идентификатор клиента на ABCDE, введите значения в других текстовых полях и нажмите кнопку Create Customer (Создать клиента) (рис. 20.6).

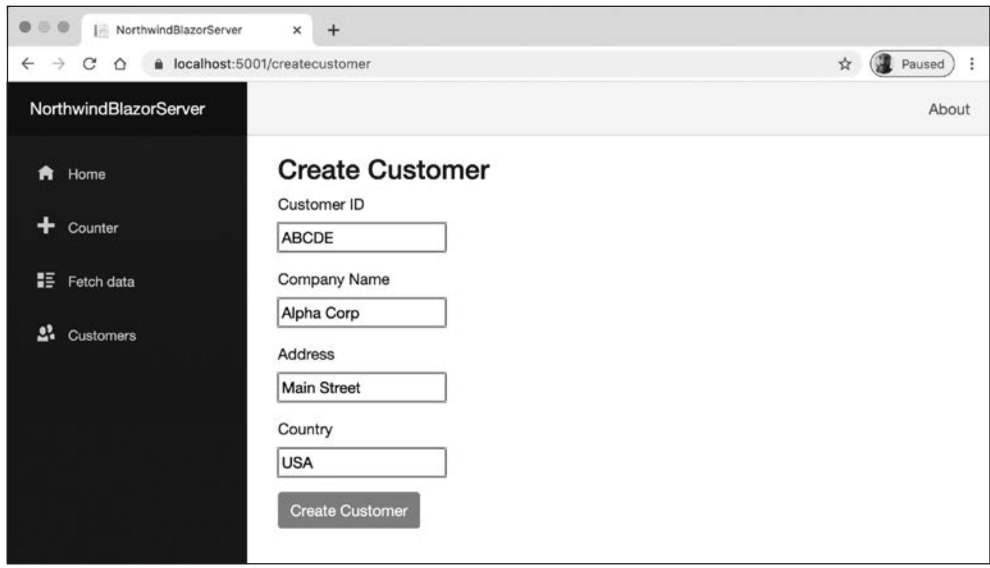


Рис. 20.6. Информация о новом клиенте, которая успешно проверяется

10. Когда появится список клиентов, прокрутите страницу вниз, чтобы увидеть нового клиента (рис. 20.7).

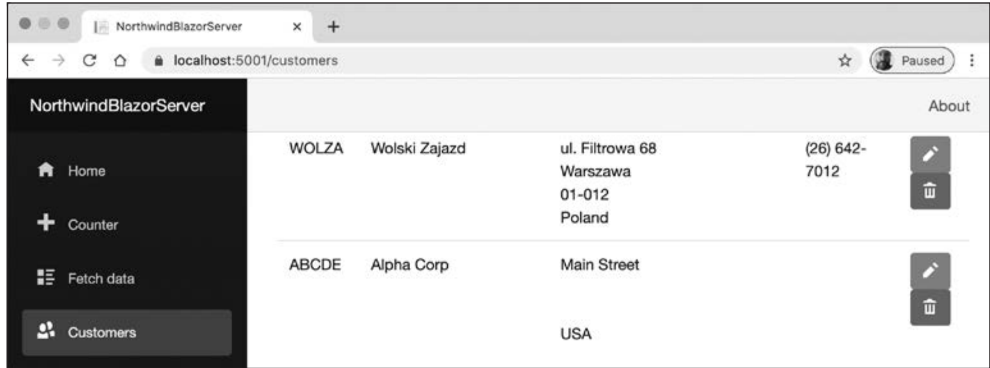


Рис. 20.7. Просмотр информации о новом клиенте

11. В строке ввода ABCDE нажмите кнопку со значком Edit (Изменить). Измените адрес, нажмите кнопку Update (Обновить) и обратите внимание, что выбранная запись обновлена.

12. В строке ввода ABCDE нажмите кнопку со значком Delete (Удалить). Обратите внимание на предупреждение, затем нажмите кнопку Delete Customer (Удалить клиента). Заметьте, что выбранная запись удалена (рис. 20.8).

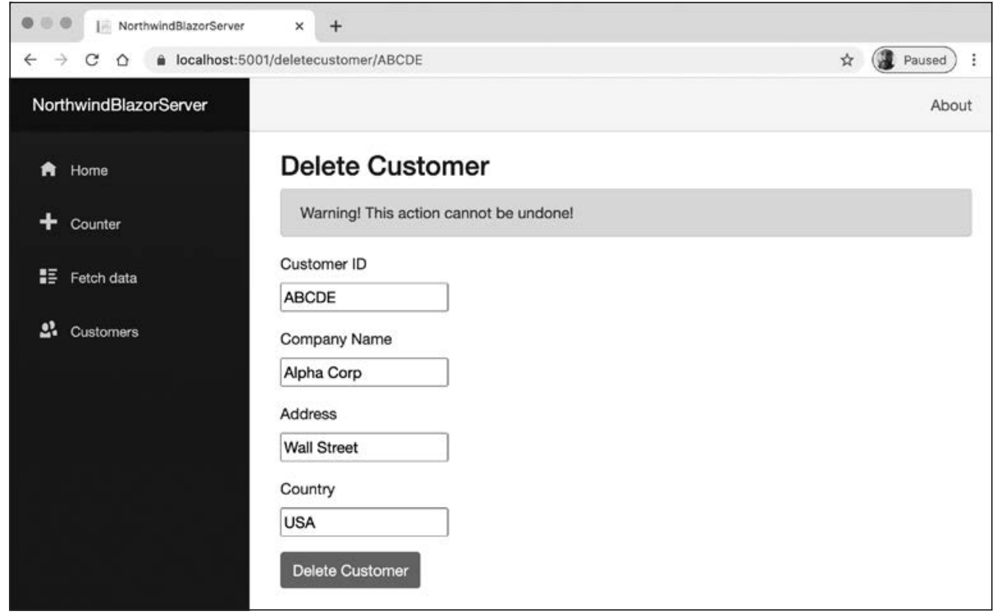


Рис. 20.8. Удаление клиента

13. Закройте браузер Google Chrome.
14. Чтобы остановить веб-сервер, в программе Visual Studio Code на панели TERMINAL (Терминал) нажмите сочетание клавиш Ctrl+C.

Создание компонентов с использованием Blazor WebAssembly

Чтобы четко увидеть основные различия, создадим ту же функциональность с помощью Blazor WebAssembly.

Поскольку мы абстрагировали локальную службу зависимостей в интерфейсе `INorthwindService`, мы сможем повторно использовать все компоненты и этот интерфейс, а также классы модели сущностей и просто переписать реализацию класса `NorthwindService` и создать клиентский контроллер для его реализации, чтобы вызвать Blazor WebAssembly (рис. 20.9)

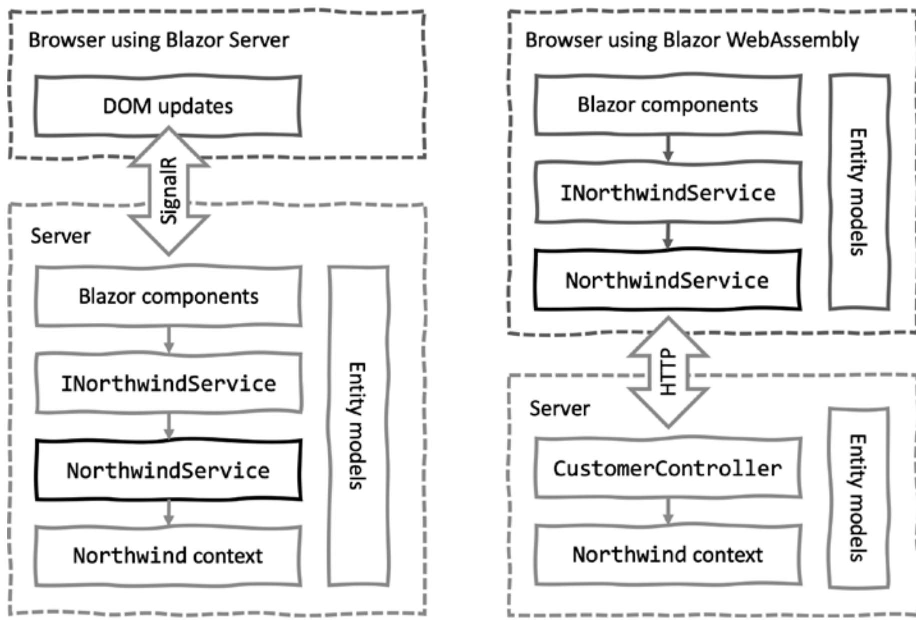


Рис. 20.9. Сравнение реализаций Blazor с использованием Server и WebAssembly

Настройки сервера для Blazor WebAssembly

Во-первых, необходимо создать службу, которую клиентское приложение может вызывать по HTTP.

Все ссылки на относительные пути для проектов и базы данных находятся на два уровня выше, например «.. \ .. \».

1. В проекте Server найдите и откройте файл `NorthwindBlazorWasm.Server.csproj` и добавьте операторы для ссылки на проект контекста базы данных `Northwind`, как показано ниже:

```
<ItemGroup>
  <ProjectReference Include=
    "..\..\NorthwindContextLib\NorthwindContextLib.csproj" />
</ItemGroup>
```

2. На панели TERMINAL (Терминал) в папке Server восстановите пакеты и скомпилируйте проект:

```
dotnet build
```

3. В проекте Server найдите и откройте файл `Startup.cs` и добавьте операторы для импорта некоторых пространств имен:

```
using Packt.Shared;
using Microsoft.EntityFrameworkCore;
using System.IO;
```

4. Для регистрации контекста базы данных `Northwind` в методе `ConfigureServices` добавьте следующие операторы:

```
string databasePath = Path.Combine(
  "..", "..", "Northwind.db");

services.AddDbContext<Northwind>(options =>
  options.UseSqlite($"Data Source={databasePath}"));
```

5. В проекте Server в папке `Controllers` создайте файл `CustomersController.cs` и добавьте операторы для определения класса контроллера веб-API с такими же методами CRUD, как и ранее:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Packt.Shared;

namespace NorthwindBlazorWasm.Server.Controllers
{
  [ApiController]
  [Route("api/[controller]")]
  public class CustomersController : ControllerBase
  {

    private readonly Northwind db;

  public CustomersController(Northwind db)
  {
    this.db = db;
  }

  [HttpGet]
  public async Task<List<Customer>> GetCustomersAsync()
  {
    return await db.Customers.ToListAsync();
  }

  [HttpGet("{id}")]
  public async Task<Customer> GetCustomerAsync(string id)
  {
    return await db.Customers.FirstOrDefaultAsync
```

```
        (c => c.CustomerID == id);
    }

    [HttpPost]
    public async Task<Customer>CreateCustomerAsync
      (Customer customerToAdd)
    {
      Customer existing = await db.Customers
        .FirstOrDefaultAsync
        (c => c.CustomerID == customerToAdd.CustomerID);

      if (existing == null)
      {
        db.Customers.Add(customerToAdd);
        int affected = await db.SaveChangesAsync();
        if (affected == 1)
        {
          return customerToAdd;
        }
      }
      return existing;
    }

    [HttpPut]
    public async Task<Customer> UpdateCustomerAsync
      (Customer c)
    {
      db.Entry(c).State = EntityState.Modified;
      int affected = await db.SaveChangesAsync();
      if (affected == 1)
      {
        return c;
      }
      return null;
    }

    [HttpDelete("{id}")]
    public async Task<int> DeleteCustomerAsync(string id)
    {
      Customer c = await db.Customers.FirstOrDefaultAsync
        (c => c.CustomerID == id);
        if (c != null)
        {
          db.Customers.Remove(c);
          int affected = await db.SaveChangesAsync();
          return affected;
        }
        return 0;
      }
    }
  }
```

Настройка клиента для Blazor WebAssembly

Также мы можем повторно использовать компоненты из проекта Blazor Server. Поскольку компоненты будут идентичными, мы можем их скопировать, и нам необходимо будет только внести изменения в локальную реализацию абстрактной службы Northwind.

1. В проекте Client найдите и откройте файл NorthwindBlazorWasm.Client.csproj и добавьте операторы для ссылки на проект библиотеки сущностей Northwind, как показано ниже:

```
<ItemGroup>
  <ProjectReference Include=
"..\..\NorthwindEntitiesLib\NorthwindEntitiesLib.csproj" />
</ItemGroup>
```

2. На панели TERMINAL (Терминал) в папке Client восстановите пакеты и скомпилируйте проект:

```
cd ..
cd Client
dotnet build
```

3. В проекте Client найдите и откройте файл _Imports.razor. Затем импортируйте пространство имен Packt.Shared, чтобы сделать типы модели сущностей Northwind доступными во всех компонентах Blazor:

```
@using Packt.Shared
```

4. В проекте Client в папке Shared найдите и откройте файл NavMenu.razor. Для клиентов добавьте элемент NavLink:

```
<li class="nav-item px-3">
  <NavLink class="nav-link" href="customers">
    <span class="oi oi-people" aria-hidden="true">
      </span> Customers
  </NavLink>
</li>
```

5. Скопируйте следующие пять компонентов из папки Pages проекта Northwind-BlazorServer в папку Pages клиентского проекта NorthwindBlazorWasm:

- CreateCustomer.razor;
- CustomerDetail.razor;
- Customers.razor;
- DeleteCustomer.razor;
- EditCustomer.razor.

6. В проекте Client создайте папку `Data`.
7. Скопируйте файл `INorthwindService.cs` из папки данных проекта `NorthwindBlazorServer` в папку данных клиентского проекта.
8. В папке `Data` добавьте новый файл `NorthwindService.cs`. Измените его содержимое, чтобы реализовать интерфейс `INorthwindService`, используя `HttpClient` для вызова службы клиентов веб-API:

```
using System.Collections.Generic;
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;
using Packt.Shared;

namespace NorthwindBlazorWasm.Client.Data
{
  public class NorthwindService : INorthwindService
  {
    private readonly HttpClient http;

    public NorthwindService(HttpClient http)
    {
      this.http = http;
    }

    public Task<List<Customer>> GetCustomersAsync()
    {
      return http.GetFromJsonAsync
        <List<Customer>>("api/customers");
    }

    public Task<Customer> GetCustomerAsync(string id)
    {
      return http.GetFromJsonAsync
        <Customer>($"api/customers/{id}");
    }

    public async Task<Customer> CreateCustomerAsync
      (Customer c)
    {
      HttpResponseMessage response = await
        http.PostAsJsonAsync<Customer>
          ("api/customers", c);

      return await response.Content
        .ReadFromJsonAsync<Customer>();
    }

    public async Task<Customer> UpdateCustomerAsync
      (Customer c)
    {
```

```
      HttpResponseMessage response = await
        http.PutAsJsonAsync<Customer>
        ("api/customers", c);

      return await response.Content
        .ReadFromJsonAsync<Customer>();
    }

    public async Task DeleteCustomerAsync(string id)
    {
      HttpResponseMessage response = await
        http.DeleteAsync($"api/customers/{id}");
    }
  }
}
```

9. Найдите и откройте файл `Program.cs`. Импортируйте пространства имен `Packt.Shared` и `NorthwindBlazorWasm.Client.Data`.
10. Для регистрации службы зависимостей Northwind в метод `ConfigureServices` добавьте следующие операторы:

```
builder.Services.AddTransient
  <INorthwindService, NorthwindService>();
```

11. На панели TERMINAL (Терминал) в папке Server скомпилируйте проект, как показано ниже:

```
cd ..
cd Server
dotnet run
```

12. Запустите Google Chrome. Перейдите в Developer Tools (Инструменты разработчика) и выберите вкладку Network (Сеть).
13. В адресной строке введите следующее: https://localhost:5001/.
14. Выберите вкладку Console (Консоль). Обратите внимание, что Blazor WebAssembly загрузил сборки .NET 5 в кеш браузера (рис. 20.10).
15. Выберите вкладку Network (Сеть).
16. Щелкните по ссылке Customers (Клиенты). Обратите внимание на HTTP-запрос GET с ответом в формате JSON, содержащим все записи клиентов (рис. 20.11).
17. Нажмите кнопку + Create New (+ Создать), чтобы добавить нового клиента, заполните форму и обратите внимание на выполненный HTTP-запрос POST (рис. 20.12).

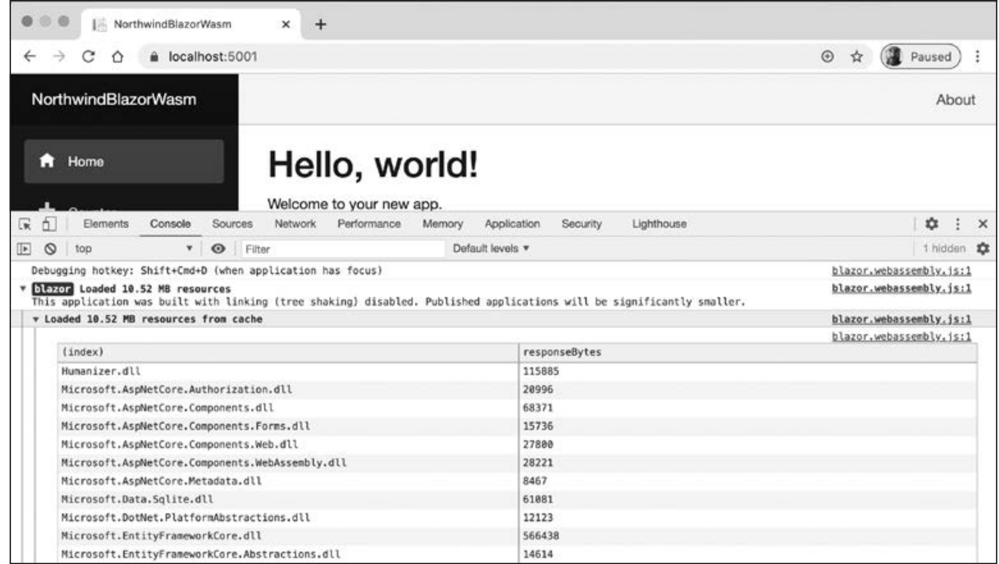


Рис. 20.10. Blazor WebAssembly загружает сборки .NET 5 в кеш браузера

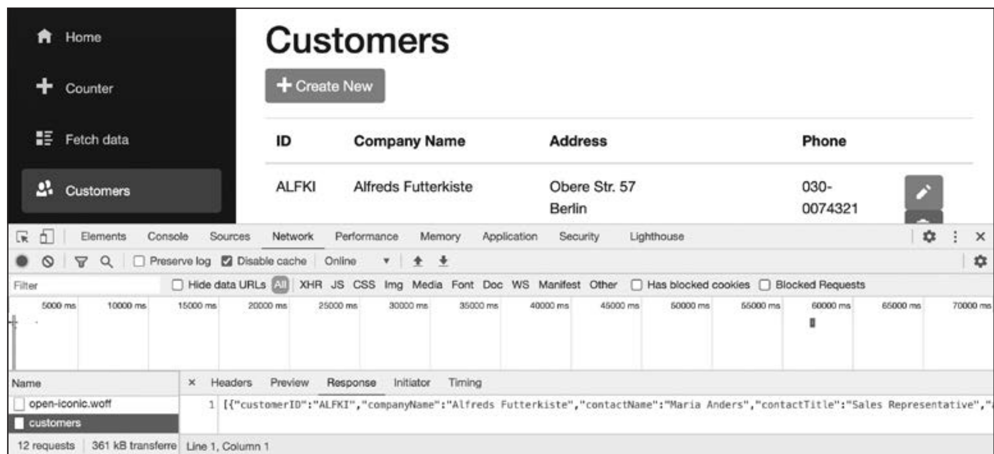


Рис. 20.11. HTTP-запрос GET с ответом в формате JSON (для клиентов)

Рис. 20.12. HTTP-запрос POST для создания нового клиента

Поддержка прогрессивных веб-приложений

Поддержка *прогрессивного веб-приложения (PWA)* в проектах Blazor WebAssembly означает, что веб-приложение получает следующие преимущества.

- Приложение работает как обычная веб-страница, пока пользователь явно не решит перейти к полноценной работе с ним.
- После установки приложения запустите его из меню Start (Пуск) или используя значок на Рабочем столе.
- Приложение визуально отображается в собственном окне вместо вкладки браузера.
- Приложение работает в офлайн-режиме (если разработчик предусмотрел, чтобы оно работало хорошо).
- Приложение обновляется автоматически.

Давайте разберемся, как работает поддержка PWA.

1. В адресной строке Google Chrome нажмите расположенную справа кнопку с плюсом в кружке и с всплывающей подсказкой Install NorthwindBlazorWasm (Установить Northwind Blazor Wasm). Затем нажмите кнопку Install (Установить) (рис. 20.13)

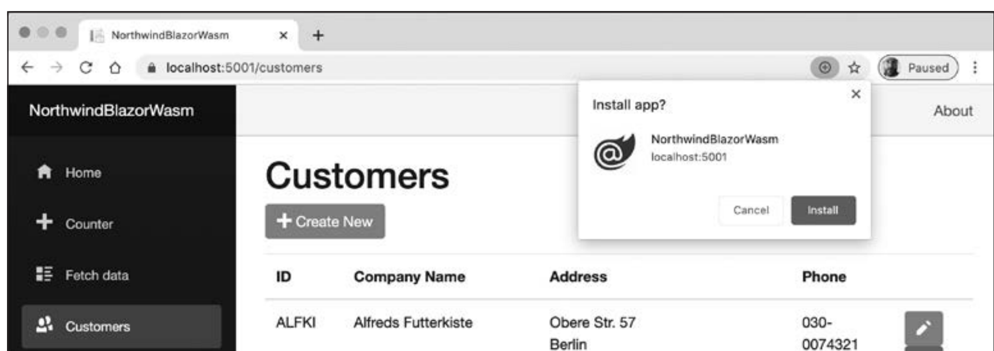


Рис. 20.13. Установка NorthwindBlazorWasm в качестве приложения

2. Закройте браузер Google Chrome.
3. Запустите приложение NorthwindBlazorWasm, используя панель запуска macOS или меню Start (Пуск) операционной системы Windows. Обратите внимание, что в нем имеются все возможности приложения.
4. Справа от строки заголовка щелкните на меню с тремя точками. Обратите внимание, что вы можете удалить приложение. Однако пока удалять его не следует (рис. 20.14).

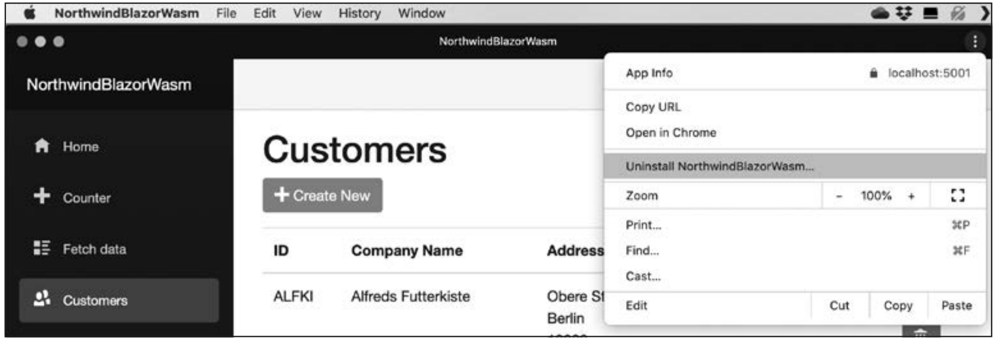


Рис. 20.14. Удаление NorthwindBlazorWasm

5. Выберите команду меню View ▸ Developer ▸ Developer Tools (Вид ▸ Разработчик ▸ Инструменты разработчика) или в Windows нажмите клавишу F12.
6. Выберите вкладку Network (Сеть), в раскрывающемся списке Throttling (Регулирование) выберите пункт Offline (Офлайн). Затем в приложении выберите Customers (Клиенты) и обратите внимание на то, что не удалось загрузить ни одного клиента, а вместо этого в нижней части окна приложения отображено сообщение об ошибке (рис. 20.15).

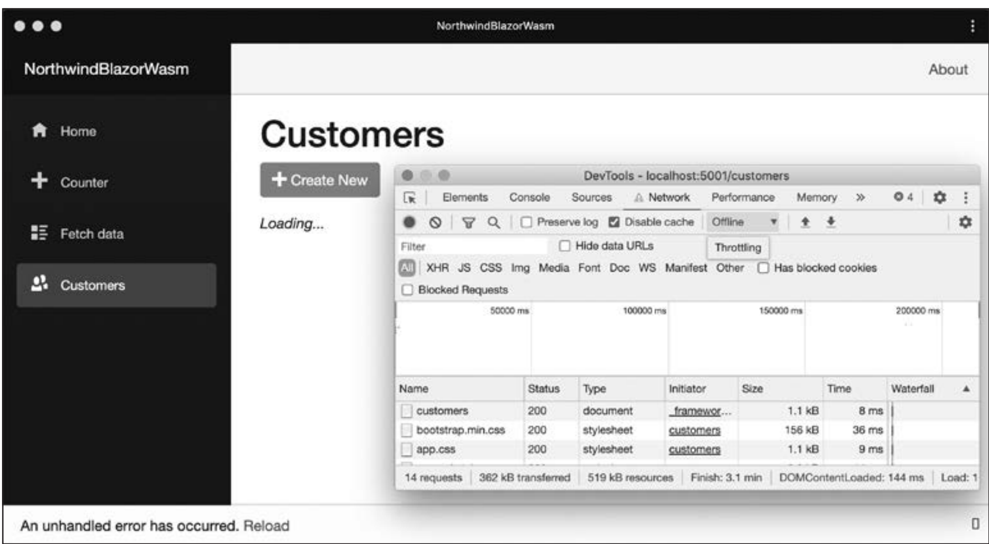


Рис. 20.15. Невозможно загрузить записи клиентов, когда сеть отключена

7. В Developer Tools (Инструменты разработчика) снова установите для параметра Throttling (Регулирование) значение Online (Онлайн).

8. Щелкните на ссылке Reload (Перезапуск), расположенной на желтой строке ошибок в нижней части приложения и обратите внимание, что функциональность возвращается.
9. Закройте приложение.

Мы могли бы усовершенствовать свой опыт, кэшируя ответы HTTP GET от службы веб-API локально и сохраняя новых клиентов и измененных или удаленных клиентов локально, а затем синхронизируя с сервером, выполняя запросы HTTP после восстановления сетевого подключения. Однако для хорошей реализации требуется много усилий.

Более подробно о реализации внутренней поддержки для проектов Blazor WebAssembly вы можете узнать на сайте https://docs.microsoft.com/ru-ru/aspnet/core/blazor/progressive-web-app#offline-support.

Еще один способ усовершенствовать проекты Blazor WebAssembly — использовать отложенную загрузку сборок.

Информацию, касающуюся ленивой загрузки сборок, вы можете получить на сайте https://docs.microsoft.com/ru-ru/aspnet/core/blazor/webassembly-lazy-load-assemblies?view=aspnetcore-5.0.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительну информацию.

Упражнение 20.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Какие две основные модели хостинга существуют в Blazor и чем они отличаются?
2. Какая дополнительная конфигурация требуется в классе `Startup` в проекте сайта Blazor Server по сравнению с проектом сайта ASP.NET Core MVC?
3. Одним из преимуществ Blazor является возможность реализации компонентов пользовательского интерфейса с использованием C# и .NET вместо JavaScript. Необходима ли Blazor какая-либо реализация JavaScript?
4. Какова роль файла `App.razor` в проекте Blazor?

5. В чем преимущество использования компонента `<NavLink>`?
6. Каким образом осуществляется передача значения компоненту?
7. В чем преимущество использования компонента `<EditForm>`?
8. Каким образом выполняются некоторые операторы при установленных параметрах?
9. Каким образом выполняются некоторые операторы при появлении компонента?
10. Назовите два ключевых различия в классе `Program` между сервером Blazor и проектом Blazor WebAssembly.

Упражнение 20.2. Упражнения по созданию компонента

Создайте компонент, отображающий таблицу умножения на основе параметра с именем `Number`, а затем протестируйте свой компонент двумя способами.

Сначала в файл `Index.razor` добавьте экземпляр компонента:

```
<timestable Number="6" />
```

Затем в адресной строке браузера введите следующий путь:

```
https://localhost:5001/timestable/6
```

Упражнение 20.3. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- Awesome Blazor: коллекция познавательных ресурсов Blazor: https://github.com/AdrienTorris/awesome-blazor;
- Blazor University: новая платформа .NET SPA от Microsoft: https://blazor-university.com;
- Blazor — семинар по созданию приложений: на этом семинаре мы создадим полное приложение Blazor и попутно узнаем о различных функциях платформы Blazor: https://github.com/dotnet-presentations/blazor-workshop/;
- «Поезд Blazor Карла Франклина»: https://www.youtube.com/playlist?list=PL8h4jt35t1wjvwFnvcB2LlYL4jLRzRmoz;
- Маршрутизация в приложениях Blazor: сравнение маршрутизации популярных веб-фреймворков, таких как React и Angular, с Blazor: https://devblogs.microsoft.com/premier-developer/routing-in-blazor-apps/;
- «Добро пожаловать в PACMAN, разработанный на C# и работающий на Blazor WebAssembly»: https://github.com/SteveDunn/PacManBlazor.

Резюме

Вы узнали, как создавать компоненты Blazor и для сервера, и для WebAssembly. Вы изучили некоторые ключевые различия между двумя моделями хостинга, например, как следует управлять данными с помощью служб зависимостей.

Далее вы узнаете, как создавать мобильные приложения с помощью Xamarin.Forms.

21 Разработка кросс-платформенных мобильных приложений

Данная глава посвящена разработке на языке C# кросс-платформенных мобильных приложений для операционных систем iOS и Android. Мобильное приложение, которое мы создадим далее, позволит просматривать список клиентов и управлять сведениями о них из базы данных `Northwind`.

Вы узнаете, как *расширяемый язык разметки приложений (XAML)* упрощает определение пользовательского интерфейса для графического приложения.

Помимо UWP-приложений, описанных в приложении Б, это единственный раздел, в котором не используется .NET Core 5. Однако в 2021 году в связи с выпуском .NET 6 все модели приложений, включая мобильные, будут использовать одну и ту же унифицированную платформу .NET.

Тему разработки мобильных приложений нельзя осветить в одной главе, но, как и веб-разработка, она сегодня настолько важна, что я хотел бы дать вам хотя бы основные сведения о ней. Представьте, что эта глава — бонус. В ней вы познакомитесь с основами разработки мобильных приложений, а более подробную информацию сможете найти в книге, посвященной разработке мобильных приложений.

Издательство Packt выпустило две книги, касающиеся Xamarin.Forms, с рейтингом 4,5 звезды на Amazon — это Xamarin.Forms Projects, доступная по следующей ссылке: https://www.packtpub.com/product/xamarin-forms-projects-second-edition/9781839210051, и Mastering Xamarin.Forms, доступная по следующей ссылке: https://www.packtpub.com/product/mastering-xamarin-forms-third-edition/9781839213380.

Мобильное приложение будет обращаться к веб-сервису Northwind, созданному с помощью ASP.NET Core Web API в главе 18. Если вы еще не создали этот сервис, то вернитесь и создайте сейчас или скачайте из репозитория GitHub на сайте https://github.com/markjprice/cs9dotnet5.

Для выполнения упражнений из этой главы вам понадобится компьютер, работающий под управлением операционной системы macOS, Xcode и среда разработки Visual Studio для Mac.

В этой главе:

- знакомство с XAML;
- знакомство с Xamarin и Xamarin.Forms;
- разработка мобильных приложений с помощью Xamarin.Forms;
- взаимодействие мобильных приложений с веб-сервисами.

Знакомство с XAML

В 2006 году корпорация Microsoft выпустила *Windows Presentation Foundation (WPF)*, которая стала первой технологией, использующей XAML. В дальнейшем быстро последовало развитие Silverlight для веб- и мобильных приложений, но Silverlight больше не поддерживается Microsoft. WPF до настоящего времени используется для создания настольных приложений Windows; например, Microsoft Visual Studio 2019 частично построена с использованием WPF.

XAML можно использовать для создания частей следующих приложений:

- UWP-приложения для устройств с Windows 10, Xbox One и Mixed Reality;
- WPF-приложения для Рабочего стола Windows, включая версии Windows 7 и более новые;
- приложения Xamarin.Forms для мобильных и настольных устройств, включая Android, iOS, Windows и macOS. В 2021 году с выпуском .NET 6 он превратится в .NET MAUI (многоплатформенный пользовательский интерфейс приложений).

Упрощение кода с помощью XAML

XAML упрощает код C#, особенно при создании пользовательского интерфейса.

Представьте, что вам для создания панели инструментов необходимо установить две и более кнопки, расположенные горизонтально.

В C# вы можете написать следующий код:

```
var toolbar = new StackPanel();
toolbar.Orientation = Orientation.Horizontal;
var newButton = new Button();
```

```
newButton.Content = "New";
newButton.Background = new SolidColorBrush(Colors.Pink);
toolbar.Children.Add(newButton);
var openButton = new Button();
openButton.Content = "Open";
openButton.Background = new SolidColorBrush(Colors.Pink);
toolbar.Children.Add(openButton);
```

В XAML данный код можно упростить. Когда приведенный XAML обрабатывается, устанавливаются эквивалентные свойства и вызываются методы для достижения той же цели, что и предыдущий код C#:

```
<StackPanel Name="toolbar" Orientation="Horizontal">
  <Button Name="newButton" Background="Pink">New</Button>
  <Button Name="openButton" Background="Pink">Open</Button>
</StackPanel>
```

XAML — это альтернативный и лучший способ объявления и создания экземпляров типов .NET для использования их в пользовательском интерфейсе.

Выбор общих элементов управления

Существует множество предопределенных элементов управления, из которых вы можете выбирать для создания общих сценариев пользовательского интерфейса. Почти все диалекты XAML поддерживают данные элементы управления (табл. 21.1.).

Таблица 21.1

Элементы управления	**Описание**
Button, Menu, Toolbar	Выполнение определенных действий
CheckBox, RadioButton	Выбор вариантов
Calendar, DatePicker	Выбор дат
ComboBox, ListBox, ListView, TreeView	Выбор элементов из списков и иерархических деревьев
Canvas, DockPanel, Grid, StackPanel, WrapPanel	Контейнеры макета, которые по-разному влияют на свои дочерние элементы
Label, TextBlock	Отображение текста только для чтения
RichTextBox, TextBox	Редактирование текста
Image, MediaElement	Встраивание изображений, видео и аудиофайлов
DataGrid	Просмотр и редактирование данных
Scrollbar, Slider, StatusBar	Различные элементы пользовательского интерфейса

Расширения разметки

Для поддержки некоторых дополнительных функций XAML использует расширения разметки. Ниже в списке приведены некоторые из наиболее важных элементов включения и привязки данных, а также повторное использование ресурсов.

- `{Binding}` связывает элемент со значением из другого элемента или источника данных.
- `{StaticResource}` связывает элемент с общим ресурсом.
- `{ThemeResource}` связывает элемент с общим ресурсом, определенным в теме.

В данной главе вы изучите несколько практических примеров расширений разметки.

Знакомство с Xamarin и Xamarin.Forms

Чтобы разработать мобильное приложение, работающее только на iPhone, вы можете задействовать Objective-C или Swift и UIKit, используя Xcode.

Разработка мобильного приложения, работающего только на телефонах с Android, возможна с помощью Java или Kotlin и Android SDK с использованием Android Studio.

В 2020 году доля iPhone и Android на мировом рынке смартфонов составляла 99,6 %. А как насчет остальных 0,4 %? Xamarin поддерживает создание мобильных приложений Tizen для устройств Samsung. Информацию, касающуюся Tizen .NET вы можете получить на сайте https://docs.microsoft.com/ru-ru/xamarin/xamarin-forms/platform/other/tizen.

Но что, если вам необходимо создать мобильное приложение, которое может работать на iPhone и телефонах с Android? И вы хотите создать это мобильное приложение только один раз, используя знакомый вам язык программирования и платформу разработки?

Платформа *Xamarin* позволяет разработчикам создавать на языке C# кросс-платформенные мобильные приложения для операционных систем Apple iOS (iPhone и iPad), macOS и Google Android с помощью .NET, которые затем компилируются в собственные API и запускаются на собственных телефонных платформах. Она основана на открытой реализации платформы .NET под названием *Mono*.

Уровень бизнес-логики может быть написан один раз и разделен между всеми мобильными платформами. Взаимодействие с пользовательским интерфейсом и API различается на разных мобильных платформах, поэтому уровень UI часто настраивается для каждой платформы. Но даже здесь существует технология, которая может облегчить разработку.

Xamarin.Forms в качестве расширения Xamarin

Инструментарий *Xamarin.Forms* расширяет возможности Xamarin, упрощая кросс-платформенную мобильную разработку благодаря возможности создавать общий код для большей части пользовательского интерфейса, а не только для бизнес-логики.

Подобно WPF- и UWP-приложениям, в Xamarin.Forms применяется язык разметки XAML в целях однократного определения пользовательского интерфейса для всех платформ сразу благодаря абстрагированию компонентов UI, специфичных для той или иной платформы. Приложения, созданные с помощью Xamarin.Forms, формируют UI на основе собственных виджетов платформы, поэтому приложения удобны и привлекательны, а также идеально подходят для целевой мобильной платформы.

UI, созданный с помощью Xamarin.Forms, никогда не будет идеально подходить для конкретной платформы, по сравнению с созданным непосредственно под эту платформу в Xamarin, но для корпоративных мобильных приложений этого более чем достаточно.

Мобильные стратегии

Мобильные приложения часто взаимодействуют с облачными сервисами.

Сатья Наделла, генеральный исполнительный директор Microsoft, как-то отлично выразился: *«Для меня стратегия mobile first означает не мобильность устройств, а мобильность индивидуального опыта. [...] Единственный способ, с помощью которого вы сможете организовать мобильность этих приложений и данных, — использовать облако»*.

Как и раньше, воспользуемся Visual Studio Code для создания сервиса ASP.NET Core Web API для поддержки мобильного приложения. Создавать приложения Xamarin.Forms разработчики могут, используя программу Visual Studio 2019 или Visual Studio для Mac. Для компиляции iOS-приложений вам потребуется компьютер Mac и среда разработки Xcode.

Если вы планируете создать мобильное приложение с помощью среды разработки Visual Studio 2019, то можете прочитать, как подключиться к серверу сборки Mac, на сайте https://docs.microsoft.com/ru-ru/xamarin/ios/get-started/installation/windows/connecting-to-mac/.

Сводная информация, содержащая сведения о том, какие среды разработки могут использоваться для создания тех или иных приложений, приведена в табл. 21.2.

Таблица 21.2

	iOS	**Android**	**ASP.NET Core Web API**
Visual Studio Code	Нет	Нет	Да
Visual Studio for Mac	Да	Да	Да
Visual Studio 2019	Нет	Да	Да

Доля рынка мобильных платформ

Данные о доле рынка следует рассматривать в контексте того, что пользователи iOS гораздо больше взаимодействуют со своими устройствами. А это важно для монетизации мобильных приложений за счет авансовых продаж, покупок в приложении или рекламы. Согласно недавним отчетам аналитиков приложения для iPhone, как правило, приносят как минимум на 60 % больше дохода, чем приложения для Android. Однако вы можете провести свое собственное исследование, поскольку мобильный мир быстро развивается.

Информацию о преимуществах и недостатках двух основных мобильных платформ, основанных на таких аспектах, как получение дохода и участие пользователей, вы можете получить на сайте https://fueled.com/blog/app-store-vs-google-play/.

Дополнительная функциональность

Создадим мобильное приложение, используя множество навыков и знаний из предыдущих глав. Кроме того, применим некоторые незнакомые функции.

Интерфейс INotificationPropertyChanged

Интерфейс `INotifyPropertyChanged` позволяет классу модели поддерживать двустороннее связывание данных. Он требует от класса реализации события `PropertyChanged`:

```
using System.ComponentModel;

public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}
```

Внутри каждого свойства в классе при установке нового значения вам необходимо инициировать событие (если оно не равно `null`) с экземпляром `PropertyChangedEventArgs`, содержащим имя свойства в виде строкового значения, как показано ниже:

```
private string companyName;

public string CompanyName
{
  get => companyName;
  set
  {
    companyName = value; // сохранение нового установленного значения

    PropertyChanged?.Invoke(this,
      new PropertyChangedEventArgs(nameof(CompanyName)));
  }
}
```

Будучи привязанным к свойству данных, элемент управления пользовательского интерфейса автоматически обновится, чтобы показать новое значение при его изменении.

Этот интерфейс подходит не только для мобильных приложений. Он может применяться и в других графических пользовательских интерфейсах, таких как настольные Windows-приложения.

Сервисы зависимостей

Мобильные платформы, такие как iOS и Android, по-разному реализуют общие функции, и потому необходим способ реализовать общие функции на уровне платформы. Мы можем сделать это с помощью сервиса зависимостей. Это работает следующим образом:

- определите интерфейс для общей функции, например `IDialer` для компонента набора телефонного номера;
- реализуйте интерфейс для всех мобильных платформ, которые вам необходимо поддерживать, например iOS и Android, и зарегистрируйте реализации атрибутом:

```
[assembly: Dependency(typeof(PhoneDialer))]
namespace NorthwindMobile.iOS
{
  public class PhoneDialer : IDialer
```

- в общем мобильном проекте получите платформозависимую реализацию интерфейса, используя сервис зависимостей:

```
var dialer = DependencyService.Get<IDialer>();
```

Больше информации о сервисе зависимостей Xamarin можно найти на сайте https://docs.microsoft.com/ru-ru/xamarin/xamarin-forms/app-fundamentals/dependency-service/introduction.

Компоненты пользовательского интерфейса Xamarin.Forms

Xamarin.Forms содержит некоторые специализированные элементы управления, предназначенные для создания мобильных пользовательских интерфейсов. Эти элементы делятся на четыре категории:

- *страницы* представляют кросс-платформенные экраны мобильных приложений, например `ContentPage`, `NavigationPage` и `CarouselPage`;
- *макеты* представляют структуру комбинации других компонентов пользовательского интерфейса, например `StackLayout`, `RelativeLayout` и `FlexLayout`;
- *представления* обозначают отдельный компонент пользовательского интерфейса, например `Label`, `Entry`, `Editor` и `Button`;
- *ячейки* представляют отдельный элемент в списке или таблице, например `TextCell`, `ImageCell`, `SwitchCell` и `EntryCell`.

Представление ContentPage

Представление `ContentPage` предназначено для простых пользовательских интерфейсов.

Оно содержит свойство `ToolbarItems`, отображающее действия, которые пользователь может выполнять на платформе. Каждый параметр `ToolbarItem` может содержать значок и текст.

```
<ContentPage.ToolbarItems>
  <ToolbarItem Text="Add" Activated="Add_Activated"
    Order="Primary" Priority="0" />
</ContentPage.ToolbarItems>
```

Больше информации о страницах в Xamarin.Forms можно найти на сайте https://docs.microsoft.com/ru-ru/xamarin/xamarin-forms/user-interface/controls/pages.

Элементы управления Entry и Editor

Элементы управления `Entry` и `Editor` применяются для редактирования текстовых значений и часто привязаны по данным к свойству модели объекта:

```
<Editor Text="{Binding CompanyName, Mode=TwoWay}" />
```

Используйте элемент `Entry` для одной строки текста.

Информацию об элементе управления Entry можно получить на сайте https://docs.microsoft.com/ru-ru/xamarin/xamarin-forms/user-interface/text/entry.

Задействуйте элемент `Editor` для нескольких строк текста.

Информацию об элементе управления Editor можно получить на сайте https://docs.microsoft.com/ru-ru/xamarin/xamarin-forms/user-interface/text/editor.

Элемент управления ListView

Элемент управления `ListView` используется для длинных списков связанных по данным значений одного типа. Может содержать верхние и нижние колонтитулы, а его элементы могут быть сгруппированы.

Элемент управления `ListView` содержит ячейки для каждого элемента списка. Существует два встроенных типа ячеек: текст и изображение. Разработчики могут определять пользовательские типы ячеек.

Для ячеек могут быть определены действия контекста, которые появляются, когда ячейка пролистывается смахиванием влево на iPhone или долгим нажатием в Android. Деструктивное действие может быть отображено красным цветом, как показано ниже:

```
<TextCell Text="{Binding CompanyName}" Detail="{Binding Location}">
  <TextCell.ContextActions>
    <MenuItem Clicked="Customer_Phoned" Text="Phone" />
    <MenuItem Clicked="Customer_Deleted" Text="Delete" IsDestructive="True" />
  </TextCell.ContextActions>
</TextCell>
```

Информацию об элементе управления ListView можно получить на сайте https://docs.microsoft.com/ru-ru/xamarin/xamarin-forms/user-interface/listview/.

Разработка мобильных приложений с помощью Xamarin.Forms

Создадим мобильное приложение для операционных систем iOS или Android для управления клиентами в `Northwind`.

Если вы никогда не запускали среду разработки Xcode, то запустите ее сейчас, чтобы увидеть окно Start (Пуск) и убедиться, что все необходимые компоненты установлены и зарегистрированы. Если не запускаете Xcode, то можете позже получить ошибки в среде Visual Studio для Mac.

Добавление Android SDK

Для разработки под Android необходимо установить хотя бы один Android SDK. Установка Visual Studio для Mac по умолчанию уже включает в себя один Android SDK, но часто это довольно старая версия для поддержки как можно большего количества устройств с Android.

Чтобы использовать последние функции Xamarin.Forms, вам необходимо установить более позднюю версию Android SDK. Возможно, вам также придется указать путь на вкладке Locations (Расположение).

1. Запустите программу Visual Studio для Mac и выберите Visual Studio ▸ Preferences (Visual Studio ▸ Настройки).
2. В окне Preferences (Настройки) выберите Projects ▸ SDK Locations ▸ Android (Проекты ▸ Расположение SDK ▸ Android) и необходимые Android Platform SDK и System Image (Образ системного диска), например Android 11.0 — R. При установке Android SDK следует выбрать хотя бы один System Image (Образ системного диска), который будет использоваться в качестве эмулятора виртуальной машины для тестирования.
3. Установите или снимите флажки, чтобы указать, что следует устанавливать либо удалять, или нажмите кнопку Updates Available (Доступные обновления) для установки обновлений, а затем нажмите кнопку Install Updates (Установить обновления).

Создание решения Xamarin.Forms

Создадим решение с проектами для кросс-платформенного мобильного приложения.

1. Нажмите кнопку New (Создать) в окне Start (Пуск), выберите File ▸ New Solution (Файл ▸ Создать решение) или нажмите сочетание клавиш Shift+Cmd+N.
2. В диалоговом окне New Project (Новый проект) в левой колонке выберите Multiplatform ▸ App (Кросс-платформенные проекты ▸ Приложение). В разделе Xamarin.Forms выберите пункт Blank Forms App (Пустое приложение Forms), используя C# в среднем столбце, а затем нажмите кнопку Next (Далее).
3. В поле App Name (Имя приложения) введите текст `NorthwindMobile`, а в поле Organization Identifier (Идентификатор организации) — значение `com.packt`, установите флажки в Target Platforms (Целевые платформы) как для Android, так и для iOS.
4. Нажмите кнопку Next (Далее).
5. Оставьте Project Name (Имя проекта) и Solution Name (Имя решения) без изменений. Измените расположение на `/Users/[`*`папка_пользователя`*`]/Code/`

`PracticalApps` и нажмите кнопку Create (Создать). Через несколько секунд будут созданы решение *`NorthwindMobile`* и три проекта:

1) `NorthwindMobile` — компоненты, совместно используемые несколькими устройствами, включая файлы XAML, определяющие пользовательский интерфейс;
2) `NorthwindMobile.Android` — компоненты, специфичные для Android;
3) `NorthwindMobile.iOS` — компоненты, специфичные для iOS.

6. Пакеты NuGet должны быть автоматически восстановлены. Если нет, то щелкните правой кнопкой мыши на решении `NorthwindMobile`, выберите Update NuGet Packages (Обновить пакеты NuGet) и примите все лицензионные соглашения.
7. Выберите Build ▸ Build All (Сборка ▸ Собрать все) и дождитесь, пока решение соберет проекты.
8. На панели управления справа от кнопки Run (Запуск) выберите проект NorthwindMobile.iOS, выберите Debug (Отладка) и выберите iPhone 11 iOS 13.6 (или более поздние версии).
9. На панели управления нажмите кнопку Run (Запуск) и дождитесь, пока симулятор запустит операционную систему iOS и ваше мобильное приложение.
10. На панели управления Simulator нажмите кнопку поворота, чтобы установить приложение iPhone в горизонтальное положение (рис. 21.1).

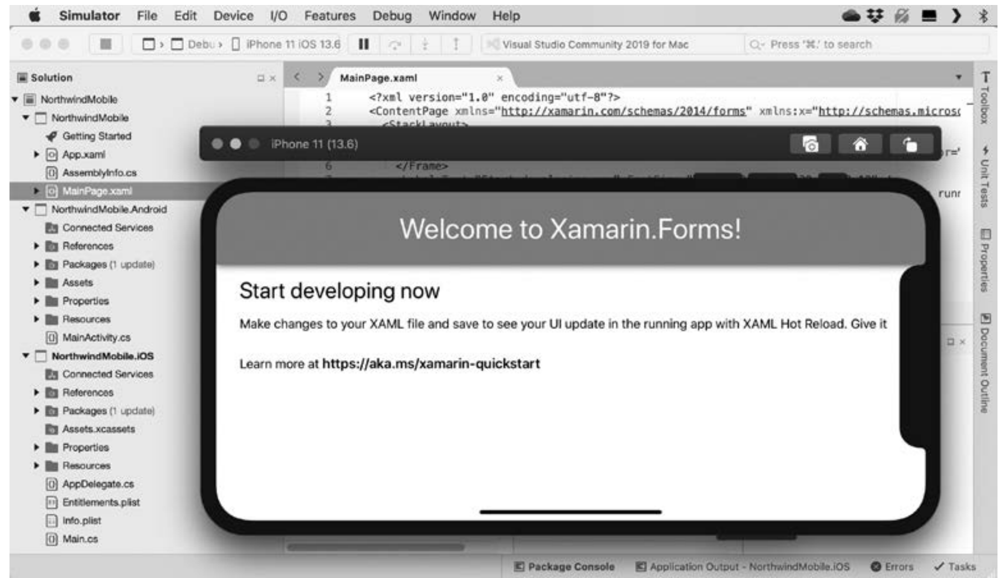


Рис. 21.1. Поворот макета iPhone

11. Выйдите из симулятора.
12. На панели управления справа от кнопки Run (Запуск) выберите проект NorthwindMobile.Android, выберите Debug (Отладка) и pixel_3a_xl_pie_9.0 _-_ api_28 (или как вы назвали свое устройство).
13. На панели управления нажмите кнопку Run (Запуск) и дождитесь, пока эмулятор устройства запустит операционную систему Android и ваше мобильное приложение (рис. 21.2).

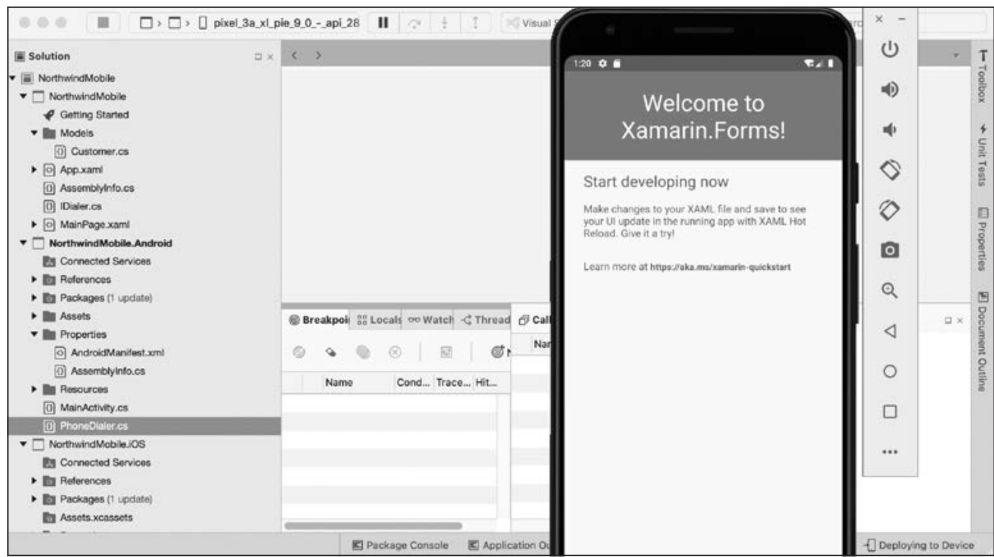


Рис. 21.2. Запуск мобильного приложения Android

14. Закройте эмулятор устройства Android.

Создание модели объекта с двусторонней привязкой данных

После ноября 2021 года, с выпуском .NET 6, мы сможем повторно использовать библиотеку сущностных моделей данных .NET Standard, созданную в главе 14. Нам необходимы объекты для реализации двусторонней привязки данных. Поэтому мы создадим сущностный класс `Customer`, который можно поместить в общий проект для операционных систем iOS и Android.

1. Щелкните правой кнопкой мыши на проекте `NorthwindMobile`, выберите Add ▸ New Folder (Добавить ▸ Новая папка) в контекстном меню и присвойте созданной папке имя `Models`.

2. Щелкните правой кнопкой мыши на папке Models и выберите Add ▸ New File (Добавить ▸ Создать файл).
3. В диалоговом окне New File (Создать файл) выберите пункт General ▸ Empty Class (Общее ▸ Пустой класс), присвойте классу имя `Customer` и нажмите кнопку New (Создать).
4. Измените операторы так, чтобы определить класс `Customer`, который реализует интерфейс `INotifyPropertyChanged` и содержит шесть свойств:

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace NorthwindMobile.Models
{
  public class Customer : INotifyPropertyChanged
  {
    public static IList<Customer> Customers;

    static Customer()
    {
      Customers = new ObservableCollection<Customer>();
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private string customerID;
    private string companyName;
    private string contactName;
    private string city;
    private string country;
    private string phone;

    // атрибут устанавливает значение параметра propertyName,
    // используя контекст, в котором этот метод вызывается
    private void NotifyPropertyChanged(
      [CallerMemberName] string propertyName = "")
    {
      // если был установлен обработчик события,
      // то вызываем делегат и передаем имя свойства
      PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyName));
    }

    public string CustomerID
    {
      get => customerID;
      set
      {
        customerID = value;
```

```
    NotifyPropertyChanged();
  }
}

public string CompanyName
{
  get => companyName;
  set
  {
    companyName = value;
    NotifyPropertyChanged();
  }
}

public string ContactName
{
  get => contactName;
  set
  {
    contactName = value;
    NotifyPropertyChanged();
  }
}

public string City
{
  get => city;
  set
  {
    city = value;
    NotifyPropertyChanged();
  }
}

public string Country
{
  get => country;
  set
  {
    country = value;
    NotifyPropertyChanged();
  }
}

public string Phone
{
  get => phone;
  set
  {
    phone = value;
    NotifyPropertyChanged();
  }
}
```

```
      public string Location
      {
        get => $"{City}, {Country}";
      }

      // для тестирования перед вызовом веб-сервиса
      public static void AddSampleData()
      {
        Customers.Add(new Customer
        {
          CustomerID = "ALFKI",
          CompanyName = "Alfreds Futterkiste",
          ContactName = "Maria Anders",
          City = "Berlin",
          Country = "Germany",
          Phone = "030-0074321"
        });

        Customers.Add(new Customer
        {
          CustomerID = "FRANK",
          CompanyName = "Frankenversand",
          ContactName = "Peter Franken",
          City = "München",
          Country = "Germany",
          Phone = "089-0877310"
        });

        Customers.Add(new Customer
        {
          CustomerID = "SEVES",
          CompanyName = "Seven Seas Imports",
          ContactName = "Hari Kumar",
          City = "London",
          Country = "UK",
          Phone = "(171) 555-1717"
        });
      }
    }
  }
```

Обратите внимание на следующие моменты.

- Класс реализует `INotifyPropertyChanged`, в связи с чем компоненты пользовательского интерфейса с двухсторонней привязкой, такие как `Editor`, будут обновлять свойство и наоборот. Реализовано событие `PropertyChanged`, которое возникает при изменении одного из свойств. Метод `NotifyPropertyChanged` упрощает реализацию этого механизма.
- После загрузки из сервиса данные о клиентах будут локально кэшироваться в мобильном приложении с помощью `ObservableCollection`. Так обеспечивается поддержка уведомлений для любых связанных компонентов пользовательского

интерфейса, таких как ListView, поэтому пользовательский интерфейс может обновить себя, когда базовые данные добавляют или удаляют элементы из коллекции.

- Помимо свойств для сохранения значений, полученных из HTTP-сервиса, класс определяет свойство Location, установленное только для чтения. Оно будет использоваться для привязки в сводном списке клиентов в целях отображения местоположения каждого из них.
- В целях тестирования, когда HTTP-сервис недоступен, используется метод, выдающий данные трех демоклиентов.

Создание компонента для набора телефонных номеров

В качестве примера компонента, специфичного для операционных систем Android и iOS, мы определим, а затем реализуем компонент набора телефонного номера.

1. Щелкните правой кнопкой мыши на папке NorthwindMobile и выберите команду Add ▸ New File (Добавить ▸ Создать файл) в контекстном меню.
2. Выберите General ▸ Empty Interface (Общее ▸ Пустой интерфейс), присвойте интерфейсу имя IDialer и нажмите кнопку New (Создать).
3. Измените код интерфейса IDialer:

```
namespace NorthwindMobile
{
  public interface IDialer
  {
    bool Dial(string number);
  }
}
```

4. Щелкните правой кнопкой мыши на папке NorthwindMobile.iOS и выберите команду Add ▸ New File (Добавить ▸ Создать файл) в контекстном меню.
5. Выберите General ▸ Empty Class (Общее ▸ Пустой класс), присвойте классу имя PhoneDialer и нажмите кнопку New (Создать).
6. Измените его содержимое:

```
using Foundation;
using NorthwindMobile.iOS;
using UIKit;
using Xamarin.Forms;

[assembly: Dependency(typeof(PhoneDialer))]

namespace NorthwindMobile.iOS
{
  public class PhoneDialer : IDialer
  {
```

```
    public bool Dial(string number)
    {
      return UIApplication.SharedApplication.OpenUrl(
        new NSUrl("tel:" + number));
    }
  }
}
```

7. Щелкните правой кнопкой мыши на папке Packages в проекте NorthwindMobile.Android и выберите Add NuGet Packages (Добавить пакеты NuGet).
8. Найдите Plugin.CurrentActivity и нажмите кнопку Add Package (Добавить пакет).
9. Откройте файл MainActivity.cs и импортируйте пространство имен Plugin.CurrentActivity:

```
using Plugin.CurrentActivity;
```

10. В начале метода OnCreate добавьте оператор для инициализации текущего действия:

```
CrossCurrentActivity.Current.Init(this, savedInstanceState);
```

11. Щелкните правой кнопкой мыши на папке NorthwindMobile.Android и выберите команду Add ▸ New File (Добавить ▸ Создать файл) в контекстном меню.
12. Выберите пункт General ▸ Empty Class (Общее ▸ Пустой класс), присвойте классу имя PhoneDialer и нажмите кнопку New (Создать).
13. Измените его содержимое:

```
using Android.Content;
using Android.Telephony;
using NorthwindMobile.Droid;
using Plugin.CurrentActivity;
using System.Linq;
using Xamarin.Forms;
using Uri = Android.Net.Uri;

[assembly: Dependency(typeof(PhoneDialer))]

namespace NorthwindMobile.Droid
{
  public class PhoneDialer : IDialer
  {
    public bool Dial(string number)
    {
      var context = CrossCurrentActivity.Current.Activity;

      if (context == null) return false;

      var intent = new Intent(Intent.ActionCall);
      intent.SetData(Uri.Parse("tel:" + number));
```

```
      if (IsIntentAvailable(context, intent))
      {
        context.StartActivity(intent); return true;
      }
      return false;
    }

    public static bool IsIntentAvailable(Context context, Intent intent)
    {
      var packageManager = context.PackageManager;

      var list = packageManager
        .QueryIntentServices(intent, 0)
        .Union(packageManager
        .QueryIntentActivities(intent, 0));

      if (list.Any()) return true;

      var manager = TelephonyManager.FromContext(context);
      return manager.PhoneType != PhoneType.None;
    }
  }
}
```

14. В проекте `NorthwindMobile.Android` раскройте папку Properties и откройте файл `AndroidManifest.xml`.
15. В области Required permissions (Требуемые разрешения) установите флажок CallPhone (рис. 21.3).

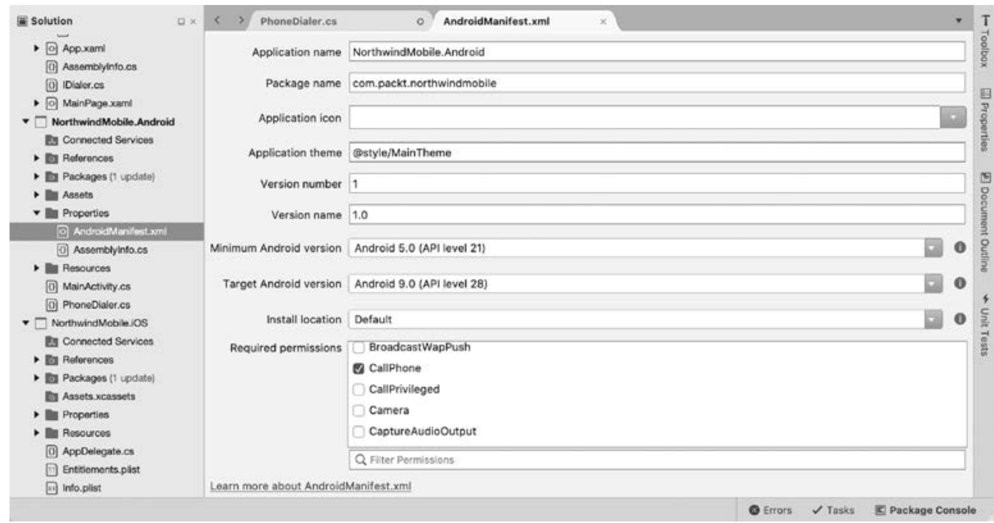


Рис. 21.3. Проверка разрешения CallPhone для устройств Android

Создание представлений для списка клиентов и подробной информации о клиенте

Заменим существующий код в файле MainPage на представления для отображения списка клиентов и сведений о клиенте.

1. В проекте NorthwindMobile щелкните правой кнопкой мыши на файле MainPage.xaml, выберите Delete (Удалить), а затем щелкните в диалоговом окне Delete (Удалить).
2. Щелкните правой кнопкой мыши на проекте NorthwindMobile, выберите команду Add ▸ New Folder (Добавить ▸ Создать папку) в контекстном меню и присвойте созданной папке имя Views.
3. Щелкните правой кнопкой мыши на папке Views и выберите команду Add ▸ New File (Добавить ▸ Создать файл) в контекстном меню. Затем выберите пункт Forms ▸ Forms ContentPage XAML (Формы ▸ XAML Forms ContentPage).
4. Присвойте файлу имя CustomersList и нажмите кнопку New (Создать).
5. Щелкните правой кнопкой мыши на папке Views и выберите команду Add ▸ New File (Добавить ▸ Создать файл) в контекстном меню. Затем выберите пункт Forms ▸ Forms ContentPage XAML (Формы ▸ XAML Forms ContentPage).
6. Присвойте файлу имя CustomerDetails и нажмите кнопку New (Создать).

Реализация представления списка клиентов

В первую очередь реализуем список клиентов.

1. Найдите и откройте файл CustomersList.xaml и измените его содержимое, как показано ниже:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage
  xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="NorthwindMobile.Views.CustomersList"
  Title="List">

  <ContentPage.Content>
    <ListView ItemsSource="{Binding .}"
              VerticalOptions="Center"
              HorizontalOptions="Center"
              IsPullToRefreshEnabled="True"
              ItemTapped="Customer_Tapped"
              Refreshing="Customers_Refreshing">
      <ListView.Header>
        <Label Text="Northwind Customers" BackgroundColor="Silver" />
      </ListView.Header>
```

```
      <ListView.ItemTemplate>
        <DataTemplate>
          <TextCell Text="{Binding CompanyName}"
                    Detail="{Binding Location}">
            <TextCell.ContextActions>
              <MenuItem Clicked="Customer_Phoned" Text="Phone" />
              <MenuItem Clicked="Customer_Deleted" Text="Delete"
                        IsDestructive="True" />
            </TextCell.ContextActions>
          </TextCell>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
  </ContentPage.Content>
  <ContentPage.ToolbarItems>
    <ToolbarItem Text="Add" Activated="Add_Activated"
                 Order="Primary" Priority="0" />
  </ContentPage.ToolbarItems>
</ContentPage>
```

Обратите внимание на следующие моменты:

- класс `ContentPage` содержит атрибут `Title` со значением `List`;
- класс `ListView` содержит `IsPullToRefreshEnabled` со значением `true`;
- были написаны обработчики для следующих событий:
  - `Customer_Tapped` — производится касание записи клиента чтобы посмотреть сведения о нем;
  - `Customers_Refreshing` — список тянется вниз для обновления элементов;
  - `Customer_Phoned` — ячейка смахивается влево на iPhone или долго нажимается в Android и далее производится касание кнопки Phone (Телефон).
  - `Customer_Deleted` — ячейка смахивается влево на iPhone или долго нажимается в Android и далее производится касание кнопки Delete (Удалить);
  - `Add_Activation` — производится касание кнопки Add (Добавить);
- шаблон данных определяет способ отображения сведений о каждом клиенте: крупный шрифт для названия компании и мелкий — для ее адреса;
- доступна кнопка Add (Добавить), позволяющая пользователям перейти к подробному представлению для добавления нового клиента.

2. Найдите и откройте файл `CustomersList.xaml.cs` и измените его содержимое, как показано ниже:

```
using System;
using System.Threading.Tasks;
using NorthwindMobile.Models;
using Xamarin.Forms;
```

```
namespace NorthwindMobile.Views
{
  public partial class CustomersList : ContentPage
  {
    public CustomersList()
    {
      InitializeComponent();

      Customer.Customers.Clear();
      Customer.AddSampleData();
      BindingContext = Customer.Customers;
    }

    async void Customer_Tapped(
      object sender, ItemTappedEventArgs e)
    {
      var c = e.Item as Customer;

      if (c == null) return;
      // переход к подробному представлению и вывод сведений
      // о выбранном клиенте
      await Navigation.PushAsync(new CustomerDetails(c));
    }

    async void Customers_Refreshing(object sender, EventArgs e)
    {
      var listView = sender as ListView;
      listView.IsRefreshing = true;
      // имитация обновления
      await Task.Delay(1500);
      listView.IsRefreshing = false;
    }

    void Customer_Deleted(object sender, EventArgs e)
    {
      var menuItem = sender as MenuItem;
      Customer c = menuItem.BindingContext as Customer;
      Customer.Customers.Remove(c);
    }

    async void Customer_Phoned(object sender, EventArgs e)
    {
      var menuItem = sender as MenuItem;
      var c = menuItem.BindingContext as Customer;

      if (await this.DisplayAlert("Dial a Number",
        "Would you like to call " + c.Phone + "?",
        "Yes", "No"))
      {
        var dialer = DependencyService.Get<IDialer>();
```

```
        if (dialer != null) dialer.Dial(c.Phone);
      }
    }

    async void Add_Activated(object sender, EventArgs e)
    {
      await Navigation.PushAsync(new CustomerDetails());
    }
  }
}
```

Обратите внимание на следующие моменты:

- значением BindingContext в конструкторе страницы устанавливается демо-список Customers;
- при касании записи о клиенте в представлении списка пользователь попадает в представление сведений (которое вы создадите на следующем шаге);
- опускаясь, представление списка запускает смоделированное обновление, которое занимает 1,5 секунды;
- при удалении из представления списка клиент удаляется из связанной коллекции клиентов;
- когда запись о пользователе в представлении списка смахивается, а кнопка Phone (Телефон) нажата, в диалоговом окне пользователю будет предложено набрать номер. Если он согласится, то встроенная в платформу реализация будет получена с помощью разрешения зависимостей, а затем использована для набора номера;
- при нажатии кнопки Add (Добавить) пользователь перенаправляется на страницу сведений о клиенте для ввода сведений о новом клиенте.

Реализация представления сведений о клиенте

Далее мы реализуем представление сведений о клиенте.

1. Откройте файл CustomerDetails.xaml и измените его содержимое, как показано в коде ниже. Обратите внимание на следующие моменты:
    - элемент Title класса ContentPage имеет значение Edit;
    - макет построен на элементе Grid для вывода клиентов, который содержит два столбца и шесть строк. Это тот же самый элемент Grid, который был описан в главе 20;
    - представление Entry двусторонне связано по данным со свойствами класса Customer;

- для `InsertButton` определен обработчик событий для выполнения кода, добавляющего нового клиента.

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage
  xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="NorthwindMobile.Views.CustomerDetails"
  Title="Edit">

  <ContentPage.Content>
    <StackLayout VerticalOptions="Fill" HorizontalOptions="Fill">
      <Grid BackgroundColor="Silver">
        <Grid.ColumnDefinitions>
          <ColumnDefinition/>
          <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
          <RowDefinition/>
          <RowDefinition/>
          <RowDefinition/>
          <RowDefinition/>
          <RowDefinition/>
          <RowDefinition/>
        </Grid.RowDefinitions>
        <Label Text="Customer ID" VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding CustomerID, Mode=TwoWay}"
               Grid.Column="1" />
        <Label Text="Company Name" Grid.Row="1"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding CompanyName, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="1" />
        <Label Text="Contact Name" Grid.Row="2"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding ContactName, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="2" />
        <Label Text="City" Grid.Row="3"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding City, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="3" />
        <Label Text="Country" Grid.Row="4"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding Country, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="4" />
        <Label Text="Phone" Grid.Row="5"
               VerticalOptions="Center" Margin="6" />
        <Entry Text="{Binding Phone, Mode=TwoWay}"
               Grid.Column="1" Grid.Row="5" />
      </Grid>
      <Button x:Name="InsertButton" Text="Insert Customer"
             Clicked="InsertButton_Clicked" />
```

```
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

2. Откройте файл `CustomerDetail.xaml.cs` и измените его содержимое, как показано в коде ниже:

```
using System;
using NorthwindMobile.Models;
using Xamarin.Forms;

namespace NorthwindMobile.Views
{
  public partial class CustomerDetails : ContentPage
  {
    public CustomerDetails()
    {
      InitializeComponent();

      BindingContext = new Customer();
      Title = "Add Customer";
    }

    public CustomerDetails(Customer customer)
    {
      InitializeComponent();

      BindingContext = customer;
      InsertButton.IsVisible = false;
    }

    async void InsertButton_Clicked(object sender, EventArgs e)
    {
      Customer.Customers.Add((Customer)BindingContext);
      await Navigation.PopAsync(animated: true);
    }
  }
}
```

Обратите внимание на следующие моменты:

- конструктор по умолчанию устанавливает контекст привязки для нового экземпляра `customer`, а заголовок представления изменяется на `Add Customer`;
- конструктор с параметром `customer` устанавливает контекст привязки для этого экземпляра и скрывает кнопку Insert (Вставить), поскольку из-за двусторонней привязки данных она не нужна при редактировании существующего клиента;
- при нажатии кнопки Insert (Вставить) новый клиент добавляется в коллекцию и навигация асинхронно возвращается к предыдущему виду.

Настройка главной страницы для мобильного приложения

Наконец, необходимо отредактировать код мобильного приложения, чтобы использовать список клиентов, обернутый в `NavigationPage`, в качестве главной страницы вместо удаленной, которая была создана с помощью шаблона проекта.

1. Откройте файл `App.xaml.cs`.
2. Импортируйте пространство имен `NorthwindMobile.Views`.
3. Измените оператор установки значения для `MainPage`, чтобы создать экземпляр `CustomersList`, обернутый в `NavigationPage`:

```
MainPage = new NavigationPage(new CustomersList());
```

Тестирование мобильного приложения в среде iOS

Протестируем мобильное приложение с помощью эмулятора iPhone.

1. В программе Visual Studio для Mac справа от кнопки Run (Запуск) на панели инструментов выберите проект NorthwindMobile.iOS. Выберите пункт меню Debug (Отладка) и iPhone 11 iOS 13.6 (или более поздней версии).
2. Нажмите кнопку Run (Запуск) на панели управления или выберите Run ▸ Start Debugging (Запуск ▸ Начать отладку). Через несколько секунд вы увидите окно, имитирующее работу вашего мобильного приложения (рис. 21.4).

Рис. 21.4. Симулятор под управлением мобильного приложения iOS

3. Щелкните кнопкой мыши на Seven Seas Imports и измените Company Name (Название компании) (рис. 21.5).

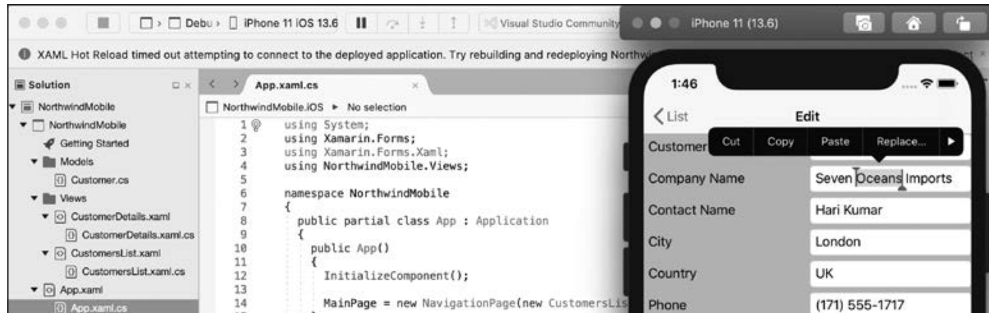


Рис. 21.5. Страница сведений о клиенте

4. Щелкните кнопкой мыши на ссылке List (Список), чтобы вернуться к списку клиентов. Обратите внимание: название компании было обновлено из-за двусторонней привязки данных.
5. Щелкните на ссылке Add (Добавить). Заполните поля ввода для добавления записи о новом клиенте (рис. 21.6).

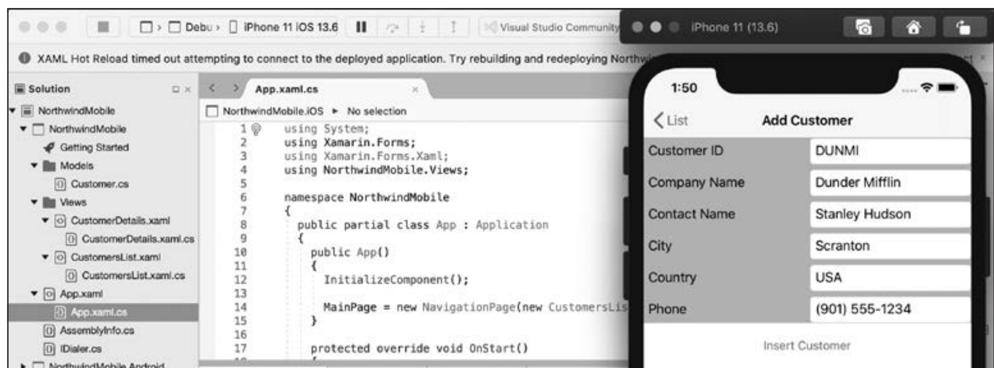


Рис. 21.6. Добавление нового клиента

6. Щелкните кнопкой мыши на ссылке Insert Customer (Вставить) и обратите внимание, что новый клиент был внесен в список.
7. Смахните строку с именем одного из клиентов влево, чтобы открыть кнопки двух действий: Phone (Позвонить) и Delete (Удалить) (рис. 21.7).

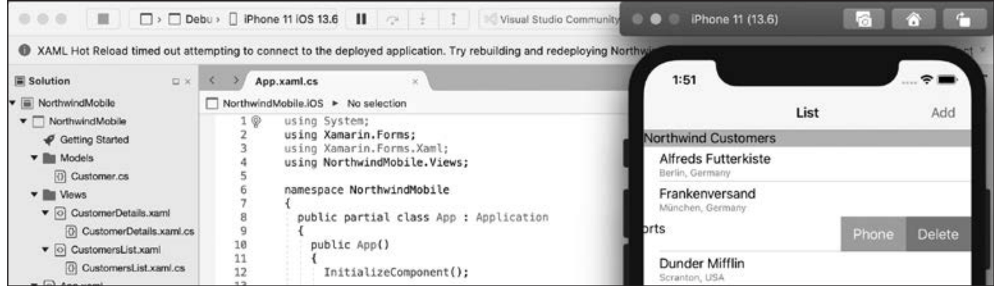


Рис. 21.7. Удаление клиента

8. Нажмите кнопку No (Нет).
9. Щелкните по кнопке Phone (Позвонить) и обратите внимание на появившееся уведомление с подтверждением телефонного вызова клиента (кнопки Yes (Да) и No (Нет)).
10. Смахните строку с именем одного из клиентов влево, чтобы открыть кнопки двух действий: Phone (Позвонить) и Delete (Удалить). Щелкните кнопкой мыши на ссылке Delete (Удалить) и обратите внимание, что сведения о клиенте были удалены.
11. Нажмите, удерживайте и перетащите список вниз, а затем отпустите и обратите внимание на эффект анимации для обновления списка. Помните, что мы не реализовали данную функцию, поэтому список не изменяется.
12. Выберите Simulator ▸ Quit Simulator (Симулятор ▸ Выйти из симулятора) или нажмите сочетание клавиш Cmd+Q.
13. Перейдите на эмулятор устройства Android и повторите те же шаги, чтобы проверить его работоспособность (рис. 21.8).

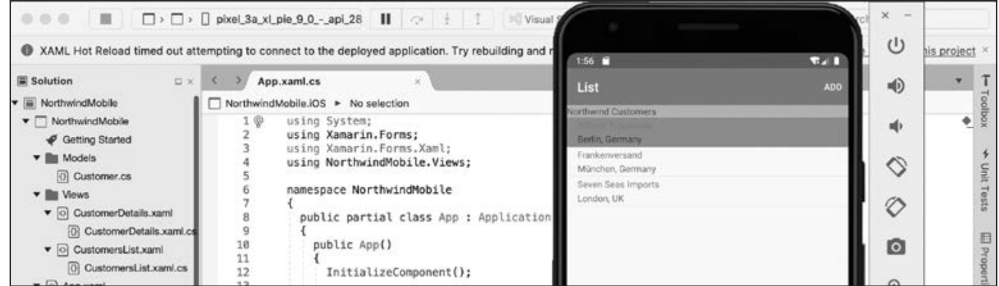


Рис. 21.8. Тестирование приложения на эмуляторе Android-устройства

Обратите внимание, что вместо того, чтобы смахнуть пальцем влево по экрану, вам необходимо нажать и удерживать, чтобы отобразить кнопки Phone (Телефон) и Delete (Удалить), которые затем появятся в верхней части пользовательского интерфейса.

Теперь мы научим мобильное приложение вызывать `NorthwindService` для получения списка клиентов.

Взаимодействие мобильных приложений с веб-сервисами

Функция платформ Apple *App Transport Security (ATS)* заставляет разработчиков использовать передовой опыт, включая безопасные соединения между приложением и веб-сервисом. Она включена по умолчанию, и при небезопасном подключении мобильные приложения будут выдавать ошибку.

Более подробную информацию об ATS можно найти на сайте https://docs.microsoft.com/ru-ru/xamarin/ios/app-fundamentals/ats.

Вызвать веб-сервис, который защищен самоподписанным сертификатом, таким как наш `NorthwindService`, возможно, однако не слишком легко.

Более подробную информацию о работе с самоподписанными сертификатами можно прочитать на сайте https://docs.remotingsdk.com/Clients/Tasks/HandlingSelfSignedCertificates/NET/.

Для простоты мы разрешим небезопасные подключения к веб-сервису и отключим проверки безопасности в мобильном приложении.

Настройка веб-сервиса в целях разрешения небезопасных запросов

В первую очередь разрешим веб-сервису обрабатывать небезопасные соединения по новому URL.

1. Запустите программу Visual Studio Code и откройте проект `NorthwindService`.
2. Откройте файл `Startup.cs` и в методе `Configure` закомментируйте перенаправление HTTPS, как показано ниже (выделено полужирным шрифтом):

```
public void Configure(IApplicationBuilder app,
  IWebHostEnvironment env)
```

```
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
  // закомментировано для мобильного приложения в главе 21
  // app.UseHttpsRedirection();

app.UseRouting();
```

3. Откройте файл `Program.cs` и в метод `CreateHostBuilder` добавьте небезопасный URL:

```
public static IHostBuilder CreateHostBuilder(
  string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
      webBuilder.UseStartup<Startup>();
      webBuilder.UseUrls(
        "https://localhost:5001", // для MVC
        "http://localhost:5003"   // для мобильного клиента
      );
    });
```

4. Выберите Terminal ▸ New Terminal (Терминал ▸ Новый терминал) и выберите проект `NorthwindService`.
5. На панели TERMINAL (Терминал) запустите веб-сервис с помощью следующей команды:

```
dotnet run
```

6. Запустите браузер Google Chrome. В адресной строке введите следующий URL: http://localhost:5003/api/customers/. Проверьте, что веб-сервис возвращает клиентов в формате JSON.
7. Закройте браузер Google Chrome.

Настройка приложения для iOS в целях разрешения небезопасных подключений

Настроим проект `NorthwindMobile.iOS`, чтобы отключить функцию ATS и разрешить незащищенные HTTP-запросы к веб-сервису.

1. В проекте `NorthwindMobile.iOS` откройте файл `Info.plist`.
2. Перейдите на вкладку Source (Источник), добавьте запись `NSAppTransportSecurity` и установите для Type (Тип) значение Dictionary (Словарь).

3. В словаре добавьте запись `NSAllowsArbitraryLoads` со значением Yes (рис. 21.9).

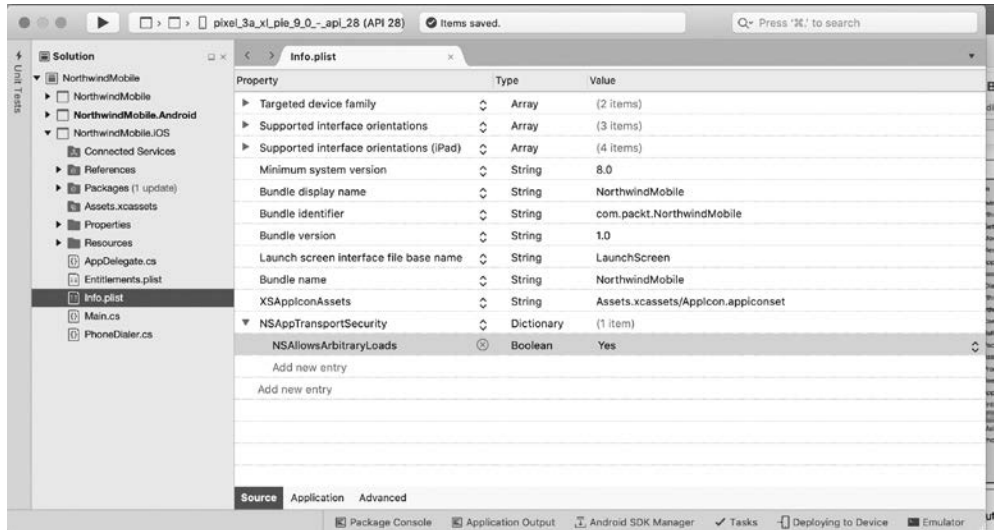


Рис. 21.9. Вкладка info.plist, отключающая требование шифрования SSL

Настройка приложения для Android в целях разрешения небезопасных подключений

Аналогично для Apple и ATS в Android 9 (уровень API 28) поддержка открытого текста (то есть без HTTPS) по умолчанию отключена.

Более подробную информацию о поддержке Android и открытого текста вы можете найти на сайте https://devblogs.microsoft.com/xamarin/cleartext-http-android-network-security/.

Настроим проект `NorthwindMobile.Android`, чтобы включить функцию открытого текста и разрешить незащищенные HTTP-запросы к веб-сервису.

1. В проекте `NorthwindMobile.Android` в папке Properties найдите и откройте файл `AssemblyInfo.cs`.
2. В конце файла добавьте атрибут для включения открытого текста, как показано ниже.

```
[assembly: Application(UsesCleartextTraffic = true)]
```

Добавление NuGet-пакетов для потребления веб-сервиса

Далее необходимо добавить несколько пакетов NuGet в каждый из проектов, специфичных для платформы, чтобы мы смогли делать HTTP-запросы и обработать ответы в формате JSON.

1. В проекте `NorthwindMobile.iOS` щелкните правой кнопкой мыши на папке Packages и выберите команду Manage NuGet Packages (Управление пакетами NuGet).
2. В диалоговом окне Manage NuGet Packages (Управление пакетами NuGet) в поле поиска введите запрос `System.Net.Http`.
3. Выберите пакет System.Net.Http и нажмите кнопку Add Package (Добавить пакет).
4. В диалоговом окне License Acceptance (Лицензионное соглашение) нажмите кнопку Accept (Принимаю).
5. В проекте `NorthwindMobile.iOS` щелкните правой кнопкой мыши на папке Packages и выберите команду Manage NuGet Packages (Управление пакетами NuGet).
6. В диалоговом окне Manage Packages (Управление пакетами) в поле поиска введите запрос `Newtonsoft.Json`.
7. Выберите пакет Newtonsoft.Json и нажмите кнопку Add Package (Добавить пакет).
8. Повторите шаги с 1 по 7, чтобы добавить те же два пакета NuGet в проект `NorthwindMobile.Android`.

Получение данных о клиентах с помощью сервиса

Теперь мы можем изменить страницу со списком клиентов, чтобы получать ее список из веб-сервиса вместо того, чтобы использовать образцы данных.

1. В проекте `NorthwindMobile` найдите и откройте файл `Views\CustomersList.xaml.cs`.
2. Импортируйте следующие пространства имен:

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

```
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading.Tasks;
using Newtonsoft.Json;
using NorthwindMobile.Models;
using Xamarin.Forms;
```

3. Измените код конструктора `CustomersList` так, чтобы загружать список клиентов через прокси сервиса вместо метода `AddSampleData`, как показано в коде, приведенном ниже:

```
public CustomersList()
{
  InitializeComponent();
  Customer.Customers.Clear();

  try
  {
    var client = new HttpClient
    {
      BaseAddress = new Uri("http://localhost:5003/")
    };

    client.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
        "application/json"));

    HttpResponseMessage response = client
      .GetAsync("api/customers").Result;

    response.EnsureSuccessStatusCode();

    string content = response.Content
      .ReadAsStringAsync().Result;

    var customersFromService = JsonConvert
      .DeserializeObject<IEnumerable<Customer>>(content);

    foreach (Customer c in customersFromService
      .OrderBy(customer => customer.CompanyName))
    {
      Customer.Customers.Add(c);
    }
  }
  catch (Exception ex)
  {
    DisplayAlert(title: "Exception",
      message: $"App will use sample data due to: {ex.Message}",
      cancel: "OK");
```

```
      Customer.AddSampleData();
    }

    BindingContext = Customer.Customers;
  }
```

4. Выберите Build ▶ Clean All (Сборка ▶ Очистить все). Изменения в файле `Info.plist`, такие как разрешение небезопасных соединений, иногда требуют чистой сборки.
5. Выберите Build ▶ Build All (Сборка ▶ Собрать все).
6. Запустите проект `NorthwindMobile` и обратите внимание, что сведения о 91 клиенте загружены с помощью веб-сервиса (рис. 21.10).

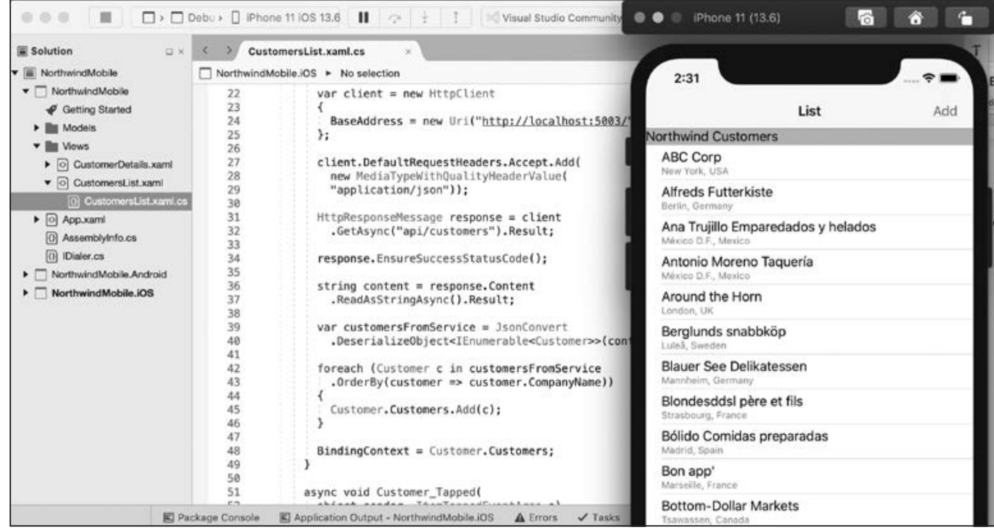


Рис. 21.10. Проект NorthwindMobile в симуляторе iPhone

7. Выберите Simulator ▶ Quit Simulator (Симулятор ▶ Выйти из симулятора) или нажмите сочетание клавиш Cmd+Q.

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение 21.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. В чем разница между Xamarin и Xamarin.Forms?
2. Каковы четыре категории компонентов пользовательского интерфейса Xamarin.Forms и что они собой представляют?
3. Перечислите четыре типа ячеек.
4. Каким образом вы можете разрешить пользователю выполнять действия с ячейками в `ListView`?
5. Как определить сервис зависимостей в целях реализации функциональности, специфичной для платформы?
6. В каком случае элемент `Entry` используется вместо элемента `Editor`?
7. В чем заключается эффект установки `IsDestructive` в значение `true` для пункта меню в контекстных действиях ячейки?
8. Когда выполняется вызов методов `PushAsync` и `PopAsync` в мобильных приложениях Xamarin.Forms?
9. Как показать всплывающее модальное сообщение с простым выбором кнопок, таких как Yes (Да) или No (Нет)?
10. В чем заключается функция ATS от Apple и почему она важна?

Упражнение 21.2. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- документация Xamarin.Forms: https://docs.microsoft.com/ru-ru/xamarin/xamarin-forms/;
- Xamarin.Essentials предоставляет разработчикам кросс-платформенные API для мобильных приложений: https://docs.microsoft.com/ru-ru/xamarin/essentials/;
- самоподписанные сертификаты iOS и закрепление сертификатов в приложении Xamarin.Forms: https://nicksnettravels.builttoroam.com/ios-certificate/;
- защита пользователей с помощью закрепления сертификатов: https://www.basdecort.com/protecting-your-users-with-certificate-pinning/;
- селектор реализации HttpClient и SSL/TLS для операционных систем iOS/macOS: https://docs.microsoft.com/ru-ru/xamarin/cross-platform/macios/http-stack.

Резюме

Вы узнали, как создать кросс-платформенное мобильное приложение с помощью Xamarin.Forms для операционных систем iOS и Android (и, возможно, других платформ), используя пространства имен пакета NuGet `System.Net.Http` и `Newtonsoft.Json`.

Из приложения Б вы узнаете, как создавать настольные приложения для Windows с помощью *Windows Presentation Foundation (WPF)* и *универсальной платформы Windows (UWP)*, а также как переносить приложения Windows Forms на .NET 5. Приложение Б доступно по следующему адресу: https://clck.ru/XtWkK.

Послесловие

Я хотел, чтобы данная книга отличалась от других, присутствующих на рынке. Надеюсь, что она вам понравилась и вы оценили обилие пошаговых инструкций в каждой теме.

Если у вас есть какие-либо предложения по интересующим вас вопросам или вы обнаружили ошибки, требующие исправления в книге или в коде, то, пожалуйста, свяжитесь со мной через мою учетную запись GitHub на сайте https://github.com/markjprice/cs9dotnet5.

Желаю вам удачи в разработке всех ваших проектов на языке C# и .NET!

Приложения

A Ответы на проверочные вопросы

Это приложение содержит ответы на *проверочные вопросы*, приведенные в конце каждой главы.

Глава 1. Привет, C#! Здравствуй, .NET Core!

1. Почему на платформе .NET Core для разработки приложений программисты могут использовать разные языки, например C# и F#?

   Ответ: платформа .NET Core поддерживает разные языки, потому что для каждого из них есть компилятор, преобразующий исходный код в промежуточный язык (IL). Затем этот IL-код преобразуется в машинные инструкции для процессора CPU с помощью общеязыковой исполняющей среды (CLR).

2. Какие команды нужно ввести для создания консольного приложения?

   Ответ: `dotnet new console`.

3. Какие команды нужно ввести в окне консоли для сборки и запуска исходного кода на языке C#?

   Ответ: в папке с файлом `ProjectName.csproj` необходимо ввести команду `dotnet run`.

4. Какое сочетание клавиш используется для открытия в программе Visual Studio Code панели TERMINAL (Терминал)?

   Ответ: Ctrl+` (обратная кавычка) — для операционной системы macOS. Ctrl+' (одинарная кавычка) — для операционной системы Windows.

5. Среда разработки Visual Studio 2019 круче, чем Visual Studio Code?

   Ответ: нет. Каждая из них оптимизирована под разные задачи. Visual Studio 2019 мощна, многофункциональна и позволяет создавать программы с графическим пользовательским интерфейсом, к примеру мобильные приложения

Windows Forms, WPF-, UWP-приложения и Xamarin, но доступна только для операционной системы Windows. Visual Studio Code — меньше, оптимизирована под набор кода в командой строке и доступна для разных операционных систем. В 2021 году с выпуском .NET 6 и .NET *Multi-Platform App User Interface (MAUI)* среда разработки Visual Studio Code получит расширение, позволяющее создавать пользовательские интерфейсы для настольных и мобильных приложений.

6. Платформа .NET Core лучше .NET Framework?

   Ответ: зависит от того, что вам нужно. .NET Core — это сокращенная, кросс-платформенная версия полнофункциональной устаревшей версии платформы .NET Framework. NET Core периодически совершенствуется. Платформа .NET Framework стабильна и лучше поддерживает устаревшие приложения. Однако текущая версия 4.8 была последней основной версией. .NET Framework никогда не будет поддерживать некоторые языковые функции C# 8 и 9.

7. Что такое .NET Standard и почему эта технология так важна?

   Ответ: .NET Standard определяет API, который может реализовывать платформа .NET.

   Последние версии .NET Framework, .NET Core, NET 5 и Xamarin реализуют .NET Standard 2.0 для предоставления единого стандартного API, который разработчики смогут изучить и настроить. Платформы .NET Core 3.0 или более поздние версии, включая .NET 5, и Xamarin реализуют .NET Standard 2.1, который содержит некоторые новые функции, не поддерживаемые .NET Framework. Если необходимо создать новую библиотеку классов, поддерживающую все платформы .NET, вам потребуется, чтобы она была совместима с .NET Standard 2.0.

8. Как называется метод точки входа консольного приложения .NET и как его объявить?

   Ответ: точка входа консольного приложения .NET — метод `Main`. Рекомендуется использовать массив `string` для аргументов командной строки и тип возвращаемого значения `int`, хоть это и не обязательно. Это может быть объявлено так, как показано далее:

   ```
   public static void Main() // минимум
   public static int Main(string[] args) // рекомендуется
   ```

9. Как найти справочную информацию по ключевому слову C#?

   Ответ: на сайте Microsoft Docs. Ключевые слова C# также приведены на сайте https://docs.microsoft.com/ru-ru/dotnet/articles/csharp/language-reference/keywords/.

10. Как найти решения общих проблем программирования?

    Ответ: https://stackoverflow.com/.

Глава 2. Говорим на языке C#

Упражнение 2.1. Проверочные вопросы

Какой тип следует выбрать для каждого указанного ниже числа?

1. Телефонный номер.

   Ответ: `string`.

2. Рост.

   Ответ: `float` или `double`.

3. Возраст.

   Ответ: `int` для характеристики или `byte` (0–255) для размера.

4. Размер оклада.

   Ответ: `decimal`.

5. Артикул книги.

   Ответ: `string`.

6. Цена книги.

   Ответ: `decimal`.

7. Вес книги.

   Ответ: `float` или `double`.

8. Размер населения страны.

   Ответ: `uint` (от 0 до примерно 4 миллиардов).

9. Количество звезд во Вселенной.

   Ответ: `ulong` (от 0 до примерно 18 квадриллионов) или `System.Numerics.BigInteger` (допускает произвольно большие целые числа).

10. Количество сотрудников на каждом из предприятий малого или среднего бизнеса (примерно до 50 000 сотрудников на каждом предприятии).

    Ответ: поскольку существуют сотни тысяч малых и средних предприятий, нам необходимо использовать размер памяти в качестве определяющего фактора, поэтому выберите `ushort`, так как этот тип занимает всего 2 байта, в отличие от `int`, который занимает 4 байта.

Глава 3. Управление потоками и преобразование типов

Упражнение 3.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Что произойдет, если разделить переменную `int` на `0`?

   Ответ: вызов исключения `DivideByZeroException` при делении целого или дробного числа.

2. Что произойдет, если разделить переменную `double` на `0`?

   Ответ: тип `double` содержит особое значение `Infinity`. Экземпляры чисел с плавающей запятой могут иметь специальные значения — `NaN` (не число), `PositiveInfinity` и `NegativeInfinity`.

3. Что происходит при переполнении переменной `int`, то есть вы присваиваете ей значение, выходящие за пределы допустимого диапазона?

   Ответ: он будет работать в цикле, если вы не поместите оператор в блок `checked`. В последнем случае будет вызвано исключение `OverflowException`.

4. В чем разница между операторами `x = y++;` и `x = ++y;`?

   Ответ: в случае с операторами `x = y++;` будет присвоено значение `x`, а затем инкрементировано значение `y`. В случае с операторами `x = ++y;` сначала будет инкрементировано значение `y`, а затем результат присвоен переменной `x`.

5. В чем разница между ключевыми словами `break`, `continue` и `return` при использовании в операторах цикла?

   Ответ: оператор `break` завершает весь цикл и продолжает выполнение кода после цикла, оператор `continue` завершит текущую итерацию цикла и продолжает выполнение с начала цикла (следующая итерация), а оператор `return` завершит текущий вызов метода и продолжает выполнение после вызова метода.

6. Из каких трех частей состоит оператор `for` и какие из них обязательны?

   Ответ: три части оператора `for` — это инициализатор, условие и инкремент. Условие требуется обязательно и должно быть логическим выражением, которое возвращает значение `true` или `false`, а две другие части опциональны.

7. В чем разница между операторами `=` и `==`?

   Ответ: `=` — оператор присваивания для назначения значений переменным, а `==` — оператор равенства, возвращающий значение `true` или `false`.

8. Будет ли скомпилирован следующий оператор: `for ( ; true; ) ;`?

Ответ: Да. Для оператора `for` требуется только логическое выражение. Выражения `initializer` и `incrementer` опциональны и могут быть опущены. Оператор `for` будет выполняться непрерывно. Это пример бесконечного цикла.

9. Что означает символ подчеркивания `_` в выражении `switch`?

   Ответ: символ подчеркивания `_` представляет возвращаемое по умолчанию значение.

10. Какой интерфейс должен реализовать объект для перечисления с помощью оператора `foreach`?

    Ответ: объект должен реализовывать интерфейс `IEnumerable`.

Упражнение 3.2. Циклы и переполнение

Что произойдет при выполнении кода, приведенного ниже?

```
int max = 500;
for (byte i = 0; i < max; i++)
{
  WriteLine(i);
}
```

Ответ: код будет циклически работать без остановки, так как значение переменной `i` может быть только в диапазоне от `0` до `255`, поэтому, когда оно инкрементируется с шагом `255`, то возвращается к `0` и поэтому всегда будет меньше, чем `max (500)`.

Чтобы предотвратить непрерывное выполнение цикла, вы можете обернуть код оператором `checked`. Это приведет к тому, что исключение будет вызываться после достижения значения `255`, например:

```
254
255
System.OverflowException says Arithmetic operation resulted in an
overflow.
```

Упражнение 3.5. Проверка знаний операторов

Чему будут равны значения `x` и `y` после выполнения следующих операторов?

1. Чему равны значения `x` и `y` после выполнения следующих операторов?

   ```
   x = 3;
   y = 2 + ++x;
   ```

   Ответ: `x` равен `4`; `y` равен `6`.

2. Чему равны значения `x` и `y` после выполнения следующих операторов?

```
x = 3 << 2;
y = 10 >> 1;
```

Ответ: `x` равен `12`; `y` равен `5`.

3. Чему равны значения `x` и `y` после выполнения следующих операторов?

```
x = 10 & 8;
y = 10 | 7;
```

Ответ: `x` равен `8`; `y` равен `15`.

Глава 4. Разработка, отладка и тестирование функций

1. Что означает в языке C# ключевое слово `void`?

   Ответ: ключевое слово `void` указывает на то, что метод не имеет возвращаемого значения.

2. В чем разница между императивным и функциональным стилями программирования?

   Ответ: императивный стиль программирования — это процесс, который описывает процесс вычисления в виде инструкций. Написанный код сообщает среде выполнения, как именно выполнять задачу. Напримар, сначала необходимо выполнить шаг 1, затем шаг 2. Это парадигма программирования, в которой используются утверждения, означающие, что состояние программы может измениться в любой момент, в том числе вне текущей функции. Императивное программирование вызывает побочные эффекты, изменяя значение некоторого состояния вашей программы. Побочные эффекты сложно отладить. Стиль функционального программирования описывает то, чего вы хотите достичь, а не способ достижения результата. Данный стиль также можно назвать декларативным. Наиболее важный момент — это то, что во избежание побочных эффектов языки функционального программирования по умолчанию делают все состояния неизменяемыми.

3. В программе Visual Studio Code какова разница между сочетаниями клавиш F5, Ctrl или Cmd+F5, Shift+F5 и Ctrl или Cmd+Shift+F5?

   Ответ: нажатие клавиши F5 сохраняет, компилирует, запускает и присоединяет отладчик, сочетания клавиш Ctrl или Cmd+F5 — сохраняет, компилирует и запускает отладчик, сочетания клавиш Shift+F5 — останавливает отладчик, а сочетания клавиш Ctrl или Cmd+Shift+F5 — перезапускает отладчик.

4. Куда записывает выходные данные метод `Trace.WriteLine`?

   Ответ: метод `Trace.WriteLine` записывает свои выходные данные в любые настроенные прослушиватели трассировки. По умолчанию это включает терминал Visual Studio Code или командную строку, но его можно настроить как текстовый файл или любой пользовательский прослушиватель.

5. Каковы пять уровней трассировки?

   Ответ: `0` = `None`, `1` = `Error` (Ошибка), `2` = `Warning` (Предупреждение), `3` = `Info` (Информация) и `4` = `Verbose` (Подробная информация).

6. В чем разница между классами `Debug` и `Trace`?

   Ответ: класс `Debug` активен только во время разработки. Класс `Trace` активен во время разработки и после выпуска в рабочую среду.

7. Как называются три А модульного теста?

   Ответ: Arrange, Act, Assert — размещение, действие, утверждение.

8. При написании модульного теста с использованием `xUnit` каким атрибутом вы должны дополнять методы тестирования?

   Ответ: `[Fact]`

9. Какая команда `dotnet` выполняет тесты `xUnit`?

   Ответ: `dotnet test`

10. Что такое TDD?

    Ответ: Test Driven Development — разработка через тестирование.

Вы можете прочитать о TDD на сайте https://docs.microsoft.com/ru-ru/dotnet/core/testing/.

Глава 5. Создание пользовательских типов с помощью объектно-ориентированного программирования

1. Какие шесть модификаторов доступа вы знаете и для чего они используются?

   Ответ: ниже в списке перечислены шесть модификаторов доступа и их значение:

   - `private`: доступ ограничен содержащим типом;
   - `internal`: доступ ограничен содержащим типом и любым другим типом в текущей сборке;
   - `protected`: доступ ограничен содержащим классом или типами, которые являются производными от содержащего класса;

- `internal protected`: доступ ограничен содержащим типом, производным классом или другим типом в текущей сборке;
- `private protected`: доступ ограничен содержащим типом или производным классом и другим типом в текущей сборке;
- `public`: неограниченный доступ.

2. В чем разница между разными ключевыми словами — `static`, `const` и `readonly`?

   Ответ: ниже в списке описана разница между ключевыми словами `static`, `const` и `readonly` применительно к элементу типа:

   - `static`: создает элемент, общий для всех экземпляров и с доступом из типа;
   - `const`: устанавливает поле как фиксированное литеральное значение, которое никогда не должно меняться, так как во время компиляции сборки операторы, использующие это поле, копируют литеральное значение;
   - `readonly`: создает поле, которое может быть назначено только во время выполнения с помощью конструктора.

3. Для чего используется конструктор?

   Ответ: конструктор служит для выделения памяти и инициализации значений полей.

4. Зачем с ключевым словом `enum` используется атрибут `[Flags]`, если требуется хранить комбинированные значения?

   Ответ: если не применить атрибут `[Flags]` к типу `enum` при сохранении скомбинированных значений, то сохраненное значение `enum`, представляющее собой комбинацию, будет возвращаться в качестве сохраненного целочисленного значения вместо списка текстовых значений, разделенных запятыми.

5. В чем польза ключевого слова `partial`?

   Ответ: вы можете использовать ключевое слово `partial` для разделения определения типа на несколько файлов.

6. Что вы знаете о кортежах?

   Ответ: это структура данных, состоящая из нескольких частей. Они используются при необходимости сохранить несколько значений, при этом не определяя тип.

7. Для чего служит ключевое слово C# `record`?

   Ответ: ключевое слово `record` определяет неизменяемую структуру данных для обеспечения более функционального стиля программирования. Как и класс, `record` может содержать свойства и методы, но значения свойств могут быть установлены только во время инициализации.

8. Что такое перегрузка?

   Ответ: перегрузка — это когда вы определяете более одного метода с одинаковым именем метода и разными входными параметрами.

9. В чем разница между полем и свойством?

   Ответ: поле — это место хранения данных, на которое можно ссылаться. Свойство — это один или пара методов, которые получают и/или устанавливают значение. Значение свойства часто хранится в закрытом поле.

10. Как сделать параметр метода необязательным?

    Ответ: вы можете сделать параметр метода необязательным, присвоив ему значение по умолчанию в сигнатуре метода.

Глава 6. Реализация интерфейсов и наследование классов

1. Что такое делегат?

   Ответ: делегат — это типобезопасный указатель на метод. Его можно использовать для выполнения любого метода с соответствующей сигнатурой.

2. Что такое событие?

   Ответ: событие — это поле, представляющее собой делегат, к которому применено ключевое слово `event`. Оно гарантирует, что используются только операторы `+=` и `-=` для безопасного объединения нескольких делегатов без замены каких-либо существующих обработчиков событий.

3. Как связаны базовый и производный классы и как производный класс может получить доступ к базовому классу?

   Ответ: производный класс (подкласс) — это класс, унаследованный от базового класса (суперкласса). Для доступа к классу, от которого наследуется подкласс, внутри производного класса необходимо использовать ключевое слово `base`.

4. В чем разница между ключевыми словами `is` и `as`?

   Ответ: оператор `is` возвращает значение `true`, если объект может быть приведен к типу; в противном случае возвращает значение `false`. Оператор `as` возвращает указатель, если объект может быть приведен к типу. В противном случае возвращается значение `null`.

5. Какое ключевое слово используется для предотвращения наследования класса и переопределения метода?

   Ответ: `sealed`.

Дополнительную информацию можно найти на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/sealed.

6. Какое ключевое слово используется для предотвращения создания экземпляра класса с помощью нового ключевого слова `new`?

   Ответ: `abstract`.

Дополнительную информацию можно найти на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/abstract.

7. Какое ключевое слово используется для переопределения члена?

   Ответ: `virtual`.

Дополнительную информацию можно найти на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/virtual.

8. Чем деструктор отличается от деконструктора?

   Ответ: *деструктор*, также известный как *финализатор*, должен использоваться для выпуска ресурсов, принадлежащих этому объекту. *Деконструктор* — новая функция C# 7 или более поздних версий, позволяющая разбивать сложный объект на более мелкие части. Это особенно полезно при работе с кортежами.

Дополнительную информацию можно найти на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/deconstruct.

9. Как выглядят сигнатуры конструкторов, которые должны иметь все исключения?

   Ответ: ниже приведены сигнатуры конструкторов, которые должны содержать все исключения:

   - конструктор без параметров;
   - конструктор со строковым параметром, обычно называемым `message`;
   - конструктор со строковым параметром, обычно называемым `message`, и параметром исключения, обычно называемым `innerException`.

10. Что такое метод расширения и как его определить?

    Ответ: метод расширения — это способ действия компилятора, который делает статический метод статического класса одним из членов типа. Расширяемый тип определяется с помощью префикса `this`.

Глава 7. Описание и упаковка типов .NET

1. В чем разница между пространством имен и сборкой?

   Ответ: *пространство имен* — это логический контейнер типа. *Сборка* — это физический контейнер типа. Чтобы использовать тип, разработчик должен ссылаться на его сборку. При желании разработчик может импортировать свое пространство имен или указать пространство имен при именовании типа.

2. Как вы ссылаетесь на другой проект в файле `.csproj`?

```
<ItemGroup>
  <ProjectReference Include="..\Calculator\Calculator.csproj" />
</ItemGroup>
```

3. В чем преимущество такого инструмента, как ILSpy?

   Ответ: преимущество такого инструмента, как ILSpy, заключается в том, чтобы научиться писать код на C# для платформы .NET, наблюдая за созданием других пакетов. Конечно, не используйте их интеллектуальную собственность. Но особенно полезно ознакомиться с тем, как разработчики Microsoft реализовали ключевые компоненты библиотек базовых классов.

4. Какой тип .NET представлен псевдонимом `float` в C#?

   Ответ: `System.Single`.

5. Какой инструмент следует использовать перед переносом приложения с .NET Framework на .NET 5?

   Ответ: перед переносом приложения с .NET Framework на .NET 5 следует использовать .NET Portability Analyzer.

6. В чем разница между платформозависимым и автономным развертыванием приложений .NET?

   Ответ: зависящие от платформы приложения .NET Core требуют наличия .NET Core для выполнения операционной системой. Автономные приложения .NET Core включают в себя все необходимое для самостоятельного выполнения.

7. Что означает RID?

   Ответ: RID — сокращение от Runtime Identifier (идентификатор среды выполнения). Значения RID используются для определения целевых платформ, на которых выполняется приложение .NET Core.

8. В чем разница между командами `dotnet pack` и `dotnet publish`?

   Ответ: с помощью команды `dotnet pack` создается пакет NuGet. С помощью команды `dotnet publish` приложение и его зависимости помещаются в папку для развертывания в хост-систему.

9. Какие типы приложений, написанных для .NET Framework, можно перенести на .NET Core?

Ответ: консоль, ASP.NET MVC, ASP.NET Web API, Windows Forms и Windows Presentation Foundation (WPF).

10. Можете ли вы использовать пакеты, написанные для .NET Framework, с .NET 5?

    Ответ: да, если они вызывают только API в .NET Standard 2.0.

Глава 8. Работа с распространенными типами .NET

1. Какое максимальное количество символов может быть сохранено в переменной типа `string`?

   Ответ: максимальный размер переменной `string` может быть равен 2 Гбайт, или примерно 1 миллиарду символов, потому что каждая переменная `char` использует 2 байта из-за внутреннего использования для символов кодировки Unicode (UTF-16).

2. В каких случаях и почему нужно использовать тип `SecureString`?

   Ответ: строковый тип хранит текстовые данные в памяти слишком долго, и при этом они не защищены. Тип `SecureString` шифрует текст и гарантирует немедленное освобождение памяти. Например, элемент `PasswordBox` сохраняет пароль в виде переменной `SecureString`, а при запуске нового процесса параметр `Password` должен быть переменной `SecureString`.

Для получения более подробной информации посетите сайт https://stackoverflow.com/questions/141203/when-would-i-need-a-securestring-in-net.

3. В каких ситуациях целесообразно применить тип `StringBuilder`?

   Ответ: при конкатенации свыше трех переменных `string` с помощью типа `StringBuilder` можно снизить потребление ресурсов памяти и улучшить производительность в сравнении со способами, предусматривающими использование метода `string.Concat` и оператора `+`.

4. В каких случаях следует задействовать `LinkedList<T>`?

   Ответ: каждый элемент в связанном списке содержит ссылку на предыдущие и следующие одноуровневые элементы, а также и на сам список, поэтому его следует использовать, когда элементы необходимо вставлять и удалять из позиций в списке, не перемещая элементы в памяти.

5. Когда класс `SortedDictionary<T>` нужно использовать вместо класса `SortedList<T>`?

   Ответ: класс `SortedList<T>` использует меньше памяти, чем `SortedDictionary<T>`, а `SortedDictionary<T>` быстрее выполняет операции добавления и удаления

неотсортированных данных. Если список заполняется с помощью уже отсортированных данных, `SortedList<T>` работает быстрее, чем `SortedDictionary<T>`.

Для получения более подробной информации посетите сайт https://stackoverflow.com/questions/935621/whats-the-difference-between-sortedlist-and-sorteddictionary.

6. Каков ISO-код языковых и региональных параметров ISO для валлийского языка?

   Ответ: `cy-GB`.

Полный список кодов языковых и региональных параметров можно найти по адресу https://lonewolfonline.net/list-net-culture-country-codes/.

7. В чем разница между локализацией, глобализацией и интернационализацией?

   Ответ

   - *Локализация* оказывает влияние на пользовательский интерфейс приложения. Локализация определяется нейтральными (только языковыми) или специфичными (языковыми и региональными) настройками. Вы предоставляете текст и другие значения в нескольких языковых версиях. К примеру, метка текстового поля с именем может быть отображена как *First name* на английском языке и *Prénom* на французском языке.
   - *Глобализация* определяет данные вашего приложения. Глобализация определяется языковыми и региональными настройками, к примеру `en-GB` для британской версии английского языка или `fr-CA` — для канадской версии французского языка. Эти настройки должны быть определены для правильного форматирования десятичных значений денежных единиц, например канадских долларов вместо французских евро.
   - *Интернационализация* — это сочетание локализации и глобализации.

8. Что означает символ `$` в регулярных выражениях?

   Ответ: в регулярных выражениях символ `$` представляет конец ввода.

9. Как в регулярных выражениях представить цифры?

   Ответ: в регулярном выражении вы можете представлять цифры с помощью `\d` или `[0-9]`.

10. Почему *нельзя* использовать официальный стандарт для адресов электронной почты при создании регулярного выражения для проверки адреса электронной почты пользователя?

Ответ: результат того не стоит. Проверка адреса электронной почты с использованием официальной спецификации не позволяет убедиться, действительно ли этот адрес существует или является ли человек, указавший адрес, его владельцем.

Для получения более подробной информации посетите сайты https://davidcel.is/posts/stop-validating-email-addresses-with-regex/ и https://stackoverflow.com/questions/201323/how-to-validate-an-email-address-using-a-regular-expression.

Глава 9. Работа с файлами, потоками и сериализация

1. Чем применение класса `File` отличается от использования класса `FileInfo`?

   Ответ: класс `File` содержит статические методы, поэтому его экземпляр не может быть создан. Он лучше всего подходит для одноразовых задач, таких как копирование файла. Класс `FileInfo` требует создания экземпляра объекта, представляющего файл. Его лучше всего использовать, когда нужно выполнить несколько операций с одним и тем же файлом.

2. В чем разница между методами потока `ReadByte` и `Read`?

   Ответ: метод `ReadByte` при каждом вызове возвращает один байт, а метод `Read` заполняет временный массив байтами до указанной длины. Обычно рекомендуется использовать метод `Read` для обработки сразу последовательности байтов.

3. В каких случаях применяются классы `StringReader`, `TextReader` и `StreamReader`?

   Ответ

   - Класс `StringReader` используется для эффективного чтения из строки, хранящейся в памяти.
   - `TextReader` — это абстрактный класс, который наследуют классы `StringReader` и `StreamReader` для совместной функциональности.
   - Класс `StreamReader` используется для чтения строк из потока, который может быть текстовым файлом любого типа, включая форматы XML и JSON.

4. Для чего предназначен тип `DeflateStream`?

   Ответ: тип `DeflateStream` реализует тот же алгоритм сжатия, что и GZIP, но без циклического избыточного кода, поэтому, хотя и создает меньшие по размеру сжатые файлы, он не может выполнять проверку целостности данных при распаковке.

5. Сколько байтов на символ затрачивается при использовании кодировки UTF-8?

   Ответ: количество байтов на символ, используемое кодировкой UTF-8, зависит от символа. Большинство символов латинского алфавита хранятся с использованием одного байта. Другим символам может потребоваться два и более байта для хранения.

6. Что такое граф объектов?

   Ответ: графом объектов является любой экземпляр классов в памяти, которые ссылаются друг на друга, тем самым формируя набор связанных объектов. Например, объект `Customer` может иметь свойство, которое ссылается на набор экземпляров `Order`.

7. Какой формат сериализации лучше всего подходит для минимизации затрат памяти?

   Ответ: объектная нотация JavaScript (JSON) имеет хороший баланс между требованиями к пространству и практическими факторами, такими как удобочитаемость. Однако буферы протокола лучше всего подходят для минимизации требований к пространству.

Дополнительную информацию вы можете найти на сайте https://stackoverflow.com/questions/52409579/protocol-buffer-vs-json-when-to-choose-one-over-another

8. Какой формат сериализации лучше всего подходит для кросс-платформенной совместимости?

   Ответ: расширяемый язык разметки (XML), хотя сегодня JSON еще более эффективен, особенно если вам необходимо интегрироваться с веб-системами или буферами протокола для лучшей производительности и использования минимальной полосы пропускания.

9. Почему не рекомендуется использовать строковое значение типа `"\Code\Chapter01"` для представления пути и что необходимо выполнить вместо этого?

   Ответ: неправильно использовать строковое значение, например `"\Code\Chapter01"`, для представления пути, так как предполагается, что символ «обратный слеш» используется в качестве разделителя папок во всех операционных системах. Вместо этого вы должны применять метод `Path.Combine` и передавать отдельные строковые значения для каждой папки, как показано в коде ниже:

   ```
   string path = Path.Combine(new[] { "Code", "Chapter01" });
   ```

10. Где вы можете найти информацию о пакетах NuGet и их зависимостях?

    Ответ: дополнительную информацию о пакетах NuGet и их зависимостях можно найти на сайте https://www.nuget.org/.

Глава 10. Защита данных и приложений

1. Какой из алгоритмов шифрования, доступных на платформе .NET, лучше всего подойдет для симметричного шифрования?

   Ответ: для симметричного шифрования лучше всего подойдет алгоритм AES.

2. Какой из алгоритмов шифрования, доступных на платформе .NET, лучше всего подойдет для асимметричного шифрования?

   Ответ: для асимметричного шифрования лучше всего подойдет алгоритм RSA.

3. Что такое радужная атака?

   Ответ: радужная атака использует таблицу предварительно рассчитанных хешей паролей. Когда база данных хешей паролей украдена, злоумышленник может быстро сравнить хеши радужной таблицы и определить оригинальные пароли.

Дополнительную информацию о радужных таблицах вы можете найти на сайте https://learncryptography.com/hash-functions/rainbow-tables.

4. При использовании алгоритмов шифрования лучше использовать блоки большого или малого размера?

   Ответ: при использовании алгоритмов шифрования лучше использовать блоки малого размера.

5. Что означает хеширование данных?

   Ответ: хеш — это вывод фиксированного размера, который получается в результате ввода произвольного размера, обрабатываемого хеш-функцией. Хеш-функции — односторонние, это означает, что единственный способ воссоздать исходные данные — перебор всех возможных входных данных и сравнение результатов.

6. Что означает подписывание данных?

   Ответ: подпись — это значение, добавляемое к цифровому документу для подтверждения его подлинности. Действительная подпись сообщает получателю, что документ был создан известным отправителем.

7. В чем разница между симметричным и асимметричным шифрованием?

   Ответ: симметричное шифрование использует секретный общий ключ для шифрования и дешифрования. Асимметричное шифрование использует открытый ключ для шифрования и закрытый ключ для дешифрования.

8. Что означает RSA?

   Ответ: Rivest — Shamir — Adleman (Ривест — Шамир — Адлеман), фамилии трех создателей криптографического алгоритма, созданного в 1978 году.

9. Почему пароли должны быть засолены перед сохранением?

   Ответ: для замедления радужных словарных атак.

10. SHA1 — это алгоритм хеширования, разработанный Национальным агентством безопасности США. Почему не следует использовать данный алгоритм?

    Ответ: алгоритм SHA-1 — небезопасный. Все современные браузеры перестали принимать SSL-сертификаты алгоритма SHA-1.

Глава 11. Работа с базами данных с помощью Entity Framework Core

1. Какой тип вы бы использовали для свойства, представляющего таблицу, например для свойства `Products` контекста базы данных?

   Ответ: `DbSet<T>`, где `T` — тип объекта, например `Product`.

2. Какой тип вы бы применили для свойства, которое представляет отношение «один ко многим», скажем, свойство `Products` объекта `Category`?

   Ответ: `ICollection<T>`, где `T` — тип объекта, например `Product`.

3. Какое соглашение, касающееся первичных ключей, действует в EF?

   Ответ: предполагается, что свойство с именем `ID` или `ClassNameID` является первичным ключом. Если тип этого свойства является одним из следующих, то свойство также обозначается как столбец `IDENTITY`: `tinyint`, `smallint`, `int`, `bigint`, `guid`.

4. Когда бы вы воспользовались атрибутом аннотаций в классе сущности?

   Ответ: вы должны в классе элемента использовать атрибут аннотации, если соглашения не могут определить правильное сопоставление между классами и таблицами. Например, если имя класса не соответствует имени таблицы или имя свойства не соответствует имени столбца. Вы также можете определить ограничения, такие как максимальная длина символов в текстовом значении или диапазон числовых значений, добавив атрибуты проверки.

5. Почему вы предпочли бы использовать Fluent API, а не атрибуты аннотации?

   Ответ: вы можете выбрать Fluent API, а не атрибуты аннотации, если соглашения не позволяют определить правильное сопоставление между классами и таблицами или если требуется, чтобы классы элементов были чистыми и свободными от постороннего кода. К примеру, при создании библиотеки классов .NET Standard 2.0 для классов сущностей вы можете использовать только атрибуты проверки сущностей, чтобы эти метаданные поддерживались такими технологиями, как Entity Framework Core и проверка привязки модели ASP.NET Core, а также настольными Windows- и мобильными приложениями. Однако вы можете использовать Fluent API для определения функциональных

возможностей Entity Framework Core, таких как сопоставление с другой таблицей или именем столбца.

6. Что означает уровень изолированности транзакции Serializable?

   Ответ: максимальные блокировки применяются для обеспечения полной изоляции от любых других процессов, работающих с затронутыми данными.

7. Что возвращает в результате метод `DbContext.SaveChanges()`?

   Ответ: значение `int` для числа затронутых объектов.

8. В чем разница между жадной и явной загрузкой?

   Ответ: жадная загрузка означает, что связанные объекты включены в исходный запрос к базе данных, поэтому их не нужно загружать позже. Явная загрузка означает, что связанные объекты не включены в исходный запрос к базе данных и должны быть явно загружены непосредственно перед тем, как они понадобятся.

9. Как определить сущностный класс EF Core для соответствия следующей таблице?

```
CREATE TABLE Employees(
  EmpID INT IDENTITY,
  FirstName NVARCHAR(40) NOT NULL,
  Salary MONEY
)
```

   Ответ: используйте следующий класс:

```
public class Employee
{
  [Column("EmpID")]
  public int EmployeeID { get; set; }
  [Required] [StringLength(40)]
  public string FirstName { get; set; }
  [Column(TypeName = "money")]
  public decimal? Salary { get; set; }
}
```

10. Какую выгоду вы получаете от объявления свойств навигации как `virtual`?

    Ответ: вы можете включить ленивую загрузку, если объявите свойства навигации сущностей в качестве виртуальных.

Глава 12. Создание запросов и управление данными с помощью LINQ

1. Каковы две обязательные составные части LINQ?

   Ответ: поставщик данных LINQ и методы расширения LINQ. Для доступа к методам расширения LINQ вы должны импортировать пространство имен `System.`

Linq, а затем обращаться к сборке поставщика того типа данных, с которыми вы хотите работать.

2. Какой метод расширения LINQ вы использовали бы для возврата из типа набора свойств?

   Ответ: метод Select позволяет осуществлять проекцию (выбор) свойств.

3. Какой метод расширения LINQ вы применили бы для фильтрации последовательности?

   Ответ: метод Where позволяет выполнить фильтрование путем предоставления делегата (или лямбда-выражения), который возвращает логическое значение, показывающее, должно ли значение быть включено в результаты.

4. Перечислите пять методов расширения LINQ, выполняющих агрегацию данных.

   Ответ: любые пять из следующих: Max, Min, Count, LongCount, Average, Sum и Aggregate.

5. Чем различаются методы расширения Select и SelectMany?

   Ответ: метод Select возвращает именно то, что вы указали для возврата. Метод SelectMany проверяет, не являются ли выбранные вами элементы IEnumerable<T>, а затем разбивает их на более мелкие части. Например, если выбранный вами тип — строковое значение (то есть IEnumerable<char>), то метод SelectMany разобьет каждое возвращенное строковое значение на соответствующие отдельные значения char.

6. В чем разница между интерфейсом IEnumerable<T> и IQueryable<T>? и как вы переключаетесь между ними.

   Ответ: интерфейс IEnumerable<T> указывает поставщика LINQ, который будет выполнять запрос локально, как LINQ to Objects. Эти поставщики не имеют ограничений, но могут быть менее эффективными. Интерфейс IQueryable<T> указывает поставщика LINQ, который сначала создает дерево выражений для представления запроса, а затем преобразует его в другой синтаксис запроса перед его выполнением, подобно тому как Entity Framework Core преобразует LINQ to SQL. Эти поставщики иногда имеют ограничения и создают исключения. Вы можете преобразовать из поставщика IQueryable<T> в поставщика IEnumerable<T>, вызвав метод AsEnumerable.

7. Что представляет собой последний параметр типа в делегатах-дженериках Func?

   Ответ: последний параметр типа в общих делегатах Func представляет тип возвращаемого значения. Например, для Func<string, int, bool> используемая функция делегата или лямбда-функции должны возвращать логическое значение.

8. В чем преимущество метода расширения LINQ, который заканчивается оператором OrDefault?

Ответ: преимущество метода расширения LINQ, который заканчивается оператором `OrDefault`, заключается в том, что он возвращает значение по умолчанию, а не выдает исключение, если не может вернуть значение. Например, вызов метода `First` для последовательности значений `int` вызовет исключение, если коллекция пуста, но метод `FirstOrDefault` вернет в результате `0`.

9. Почему понятный синтаксис запросов необязателен?

   Ответ: синтаксис понимания запросов — необязательный, так как это просто синтаксический сахар. Он облегчает чтение кода разработчиками, но не добавляет никакой дополнительной функциональности.

Более подробно о ключевом слове let вы можете прочитать на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/let-clause.

10. Каким образом вы можете создать свои собственные методы расширения LINQ?

    Ответ: создайте класс `static` со статическим методом с параметром `IEnumerable<T>` с префиксом `this`.

```
namespace System.LINQ
{
  public static class MyLINQExtensionMethods
  {
    public static IEnumerable<T> MyChainableExtensionMethod<T>(
      this IEnumerable<T> sequence)
    {
    // вернуть что-нибудь IEnumerable<T>
    }
    public static int? MyAggregateExtensionMethod<T>(
      this IEnumerable<T> sequence)
    {
    // вернуть некоторое значение int
    }
  }
}
```

Глава 13. Улучшение производительности и масштабируемости с помощью многозадачности

1. Какую информацию о классе `Process` вы можете найти?

   Ответ: класс `Process` содержит ряд свойств, включая `ExitCode`, `ExitTime`, `Id`, `MachineName`, `PagedMemorySize64`, `ProcessorAffinity`, `StandardInput`, `StandardOutput`, `StartTime`, `Threads` и `TotalProcessorTime`.

Дополнительную информацию о классе Process вы можете найти на сайте https://docs.microsoft.com/ru-ru/dotnet/api/system.diagnostics.process.

2. Насколько точен класс `Stopwatch`?

   Ответ: класс `Stopwatch` может быть точным до наносекунды (миллиардной доли секунды), но все равно может быть погрешность.

Дополнительную информацию о том, как повысить точность, установив привязку процессора вы можете найти на сайте https://www.codeproject.com/Articles/61964/Performance-Tests-Precise-Run-Time-Measurements-wi.

3. По соглашению какой суффикс должен быть применен к методу, если он возвращает `Task` или `Task<T>`?

   Ответ: `Async`. Например, `OpenAsync` при использовании метода с именем `Open`.

4. Какое ключевое слово следует применить к объявлению метода, чтобы в нем можно было использовать ключевое слово `await`?

   Ответ: к объявлению метода следует применить ключевое слово `async`.

5. Как создать дочернюю задачу?

   Ответ: для создания дочерней задачи необходимо вызвать метод `Task.Factory.StartNew` с параметром `TaskCreationOptions.AttachToParent`.

6. Почему не следует использовать ключевое слово `lock`?

   Ответ: ключевое слово `lock` не позволяет указать время ожидания, что может привести к взаимоблокировке. Вместо этого используйте метод `Monitor.Enter` со структурой `TimeSpan` и метод `Monitor.Exit`.

7. В каких случаях нужно применять класс `Interlocked`?

   Ответ: класс `Interlocked` нужно использовать, если в коде используются целые числа и числа с плавающей запятой, распределяемые между несколькими потоками.

8. Когда класс `Mutex` следует задействовать вместо класса `Monitor`?

   Ответ: используйте класс `Mutex`, когда вам необходимо поделиться ресурсом через границы процесса. Класс `Monitor` работает только с ресурсами внутри текущего процесса.

9. В чем преимущество использования на сайте или в веб-сервисе методов `async` и `await`?

   Ответ: на сайте или в веб-сервисе использование методов `async` и `await` улучшает масштабируемость, но не производительность конкретного запроса, поскольку требуется дополнительная работа по обработке потоков.

10. Вы можете отменить задачу? Каким образом?

    Ответ: да, можно отменить задачу. Как это сделать, описано на сайте https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/async/cancel-an-async-task-or-a-list-of-tasks.

Глава 15. Разработка сайтов с использованием ASP.NET Core Razor Pages

1. Перечислите шесть методов, которые могут быть указаны в HTTP-запросе.

   Ответ: GET, HEAD, POST, PUT, PATCH, DELETE. Другие включают TRACE, OPTIONS и CONNECT.

Дополнительную информацию об определении HTTP-методов вы можете найти на сайте https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

2. Перечислите шесть кодов состояния и их описания, которые могут быть возвращены в HTTP-ответе.

   Ответ: 200 OK, 201 Created, 301 Moved Permanently, 400 Bad Request, 404 Not Found (отсутствует ресурс), 500 Internal Server Error. Другие включают 101 Switching Protocols (например, с HTTP на WebSocket), 202 Accepted, 204 No Content, 304 Not Modified, 401 Unauthorized, 403 Forbidden, 406 Not Acceptable (например, запрашивается формат ответа, который не поддерживается сайтом), 503 Service Unavailable.

Дополнительную информацию об определениях кода состояния вы можете найти на сайте https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

3. Для чего в ASP.NET Core используется класс `Startup`?

   Ответ: в ASP.NET Core класс `Startup` используется для добавления и настройки служб в конвейере запросов и ответов, таких как обработка ошибок, параметры безопасности, статические файлы, файлы по умолчанию, маршрутизация конечных точек, Razor Pages и MVC и контексты данных Entity Framework Core.

4. Что такое HSTS и для чего он предназначен?

   Ответ: HTTP Strict Transport Security (HSTS) — дополнительный механизм усовершенствования безопасности. Если сайт указывает его и браузер его поддерживает, то он заставляет все коммуникации по HTTPS и не позволяет пользователю задействовать недоверенные или недействительные сертификаты.

5. Как включить статические HTML-страницы для сайта?

   Ответ: чтобы включить статические HTML-страницы для сайта, необходимо добавить операторы в метод `Configure` класса `Startup`, чтобы использовать файлы по умолчанию, а затем — статические файлы (этот порядок важен!), как показано ниже:

```
app.UseDefaultFiles(); // index.html, default.html и т. д.
app.UseStaticFiles();
```

6. Как вставить код C# в HTML, чтобы создать динамическую страницу?

   Ответ: чтобы смешать код языка C# с HTML и создать динамическую страницу, вы можете создать файл Razor с расширением `.cshtml`. Затем вы можете добавить префикс любых выражений C# с символом `@`, а операторы C# заключить в фигурные скобки или создать раздел `@functions`, как показано ниже:

```
@page
@functions
{
  public string[] DaysOfTheWeek
  {
    get => System.Threading.Thread.CurrentThread
      .CurrentCulture.DateTimeFormat.DayNames;
  }

  public string WhatDayIsIt
  {
    get => System.DateTime.Now.ToString("dddd");
  }
}
<html>
<head>
  <title>Today is @WhatDayIsIt</title>
</head>
<body>
  <h1>Days of the week in your culture</h1>
  <ul>
@{
// чтобы добавить блок операторов, используйте фигурные скобки
foreach (string dayName in DaysOfTheWeek)
  {
  <li>@dayName</li>
  }
}
</ul>
</body>
</html>
```

7. Как можно определить общие макеты для Razor Pages?

   Ответ: чтобы определить общие макеты для Razor Pages, создайте как минимум два файла: `_Layout.cshtml` для определения разметки для общего макета и `_ViewStart.cshtml` для установки макета по умолчанию:

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

8. Как можно отделить разметку от кода на странице Razor?

   Ответ: чтобы отделить разметку от кода на странице Razor, создайте два файла: `MyPage.cshtml`, который содержит разметку, и `MyPage.cshtml.cs`, содержащий класс, наследуемый от `PageModel`. В файле `MyPage.cshtml` установите модель для использования класса:

```
@page
@model MyProject.Pages.MyPageModel
```

9. Как настроить контекст данных Entity Framework Core для использования с сайтом ASP.NET Core?

   Ответ: чтобы настроить контекст данных Entity Framework Core для использования с сайтом ASP.NET Core, необходимо выполнить следующее:

   - в файле проекта укажите ссылку на сборку, определяющую класс контекста данных;
   - в классе `Startup` импортируйте пространства имен для `Microsoft.EntityFrameworkCore` и класс контекста данных;
   - в методе `ConfigureServices` добавьте следующий оператор, который настраивает контекст данных со строкой подключения к базе данных для использования с указанным поставщиком базы данных, таким как SQLite, как показано ниже:

```
services.AddDbContext<MyDataContext>(options => options.UseSqlite("my
database connection string"));
```

   - в классе модели Razor Page или в разделе `@functions` объявите скрытое поле для хранения контекста данных, а затем установите его в конструкторе, как показано ниже:

```
private MyDataContext db;

public SuppliersModel(MyDataContext injectedContext)
{
  db = injectedContext;
}
```

10. Как повторно использовать Razor Pages в ASP.NET Core 2.2 или более поздней версии?

    Ответ: чтобы повторно использовать Razor Pages в ASP.NET 2.2 или более поздней версии, все, что связано со страницей Razor, можно скомпилировать в библиотеку классов, используя команду `dotnet new razorclasslib -s`.

Глава 16. Разработка сайтов с использованием паттерна MVC

1. Зачем нужны файлы со специальными именами `_ViewStart` и `_ViewImports`, созданные в папке Views?

   Ответ

   - Файл `_ViewStart` содержит блок операторов, которые выполняются при вызове метода `View`, когда метод действия контроллера передает модель в представление, например, для установки макета по умолчанию.
   - Файл `_ViewImports` содержит операторы `@using` для импорта пространств имен для всех представлений, чтобы избежать необходимости добавлять одни и те же операторы импорта в верхней части всех представлений.

2. Что представляют собой имена трех сегментов, определенных по умолчанию в ASP. NET Core MVC, и какие из них необязательны?

   Ответ

   - `{controller}`: например, `/home` представляет класс контроллера для создания экземпляра `HomeController`. Это необязательно, так как он может использовать значение по умолчанию: `Home`.
   - `{action}`: например, `/index` представляет метод действия для выполнения `Index`. Это не обязательно, так как он может использовать значение по умолчанию: `Index`.
   - `{id}`: например, `/5` представляет параметр в методе действия `int id`. Это не обязательно, так как к нему добавляется суффикс `?`.

3. Что такое привязка моделей и какие типы данных она может обрабатывать?

   Ответ: связыватель модели по умолчанию устанавливает параметры в методе действия. Он может обрабатывать следующие типы данных:

   - простые типы, такие как `int`, `string`, `DateTime`, включая типы `nullable`;
   - сложные типы, такие как `Person`, `Product`;
   - типы коллекций, такие как `IEnumerable<T>`.

4. Как выводится содержимое текущего представления в общем файле макета, таком как `_Layout.cshtml`?

Ответ: чтобы вывести содержимое текущего представления в общем макете, вызовите метод `RenderBody`:

```
@RenderBody()
```

5. Как в общем файле макета, таком как `_Layout.cshtml`, задать секцию, для которой содержимое может быть отображено текущим представлением, и как это представление выдает содержимое для этой секции?

   Ответ: чтобы вывести содержимое раздела в общем макете, вызовите метод `RenderSection`, указав при необходимости имя раздела:

```
@RenderSection("Scripts", required: false)
```

   Чтобы определить содержимое раздела в представлении, создайте именованный раздел:

```
@section Scripts
{
  <script> alert('hello');
  </script>
}
```

6. Какие пути ищутся для представления по соглашению при вызове метода `View` внутри метода действия контроллера?

   Ответ: при вызове метода `View` внутри метода действия контроллера выполняется поиск трех путей для представления по умолчанию на основе сочетаний имен контроллера, метода действия и специальной общей папки, как показано в следующем выводе:

```
InvalidOperationException: The view 'Index' was not found. The following
locations were searched:
/Views/Home/Index.cshtml
/Views/Shared/Index.cshtml
/Pages/Shared/Index.cshtml
```

   Это можно обобщить следующим образом:

   - `/Views/[controller]/[action].cshtml`;
   - `/Views/Shared/[action].cshtml`;
   - `/Pages/Shared/[action].cshtml`.

7. Как вы можете указать браузеру пользователя кэшировать ответ в течение 24 часов?

   Ответ: чтобы настроить браузер пользователя для кэширования ответа в течение 24 часов, дополните класс контроллера или метод действия атрибутом `[ResponseCache]` и установите для параметра `Duration` значение `86400` секунд, а для параметра `Location` — `ResponseCacheLocation.Client`.

8. Почему вы можете решить подключить Razor Pages, даже если сами их не создаете?

   Ответ: если вы использовали такие функции, как ASP.NET Core Identity UI, то для этого требуются Razor Pages.

9. Как ASP.NET Core MVC идентифицирует классы, которые могут действовать как контроллеры?

   Ответ: ASP.NET Core MVC идентифицирует классы, которые могут действовать как контроллеры, проверяя, не дополнен ли класс (или класс, из которого он происходит) атрибутом `[Controller]`.

10. Каким образом ASP.NET Core MVC облегчает тестирование сайта?

    Ответ: шаблон проектирования Model-View-Controller (MVC) разделяет технические аспекты формы данных (модели), исполняемых операторов для обработки входящего запроса и исходящего ответа, а затем генерирует ответ в формате, запрошенном пользовательским агентом, например HTML или JSON. Это облегчает написание модульных тестов. ASP.NET Core также упрощает реализацию шаблонов проектирования Inversion-of-Control (IoC) и Dependency Injection (DI) для удаления зависимостей при тестировании компонента, такого как контроллер.

Глава 17. Разработка сайтов с помощью системы управления контентом (CMS)

1. Каковы преимущества использования системы управления контентом для разработки сайта по сравнению с применением только ASP.NET Core?

   Ответ: система управления контентом CMS отделяет контент (значения данных) от шаблонов (макет, формат и стиль). CMS предоставляет пользовательский интерфейс управления контентом, чтобы нетехнические пользователи могли быстро и легко управлять контентом, сохраняя при этом профессионально выглядящий сайт.

   CMS уровня предприятия также будет обеспечивать такие функции, как SEO-URL, точный контроль аутентификации и авторизации, управление медиаресурсами, различные способы повторного использования контента, дизайнер визуальных форм, различные способы персонализации контента и поддержка нескольких версий и языков для контента.

2. Каков специальный относительный URL для доступа к UI управления Piranha CMS и какие имя пользователя и пароль настроены по умолчанию?

   Ответ: специальный относительный URL-путь для доступа к пользовательскому интерфейсу управления Piranha CMS — `/manager`, а имя пользователя и пароль, настроенные по умолчанию, — `admin` и `password`.

3. Что такое слаг?

   Ответ: слаг — это относительный URL-путь для элемента контента, такого как страница или сообщение.

4. В чем разница между сохранением контента и его публикацией?

   Ответ: разница между сохранением и публикацией контента заключается в том, что при сохранении сохраняются изменения в базе данных CMS, а публикация также делает эти изменения доступными для пользователей.

5. Каковы три типа контента Piranha CMS и для чего они используются?

   Ответ: существует три типа контента Piranha CMS: сайты, страницы и сообщения. Сайты предназначены для значений свойств, которые должны быть общими для всех страниц. Страницы предназначены для обычных веб-страниц. Сообщения предназначены для специальных страниц, которые могут отображаться только на странице архива с возможностью сортировки и фильтрации.

6. Каковы три типа компонентов Piranha CMS и для чего они применяются?

   Ответ: существует три типа компонентов Piranha CMS — это поля, регионы и блоки. Поля предназначены для простых значений данных, таких как строки и числа. Регионы предназначены для сложных значений данных, которые отображаются в фиксированном месте с типом контента, таким как страница. Блоки предназначены для сложных значений данных, которые отображаются в любом порядке и комбинации, определенной менеджером контента.

7. Перечислите три свойства, которые тип `Page` наследует от своих базовых классов, и объясните, для чего они используются.

   Ответ: три свойства, которые тип `Page` наследует от своих базовых классов, включают:

   - `ParentId` — значение `Guid`, которое ссылается на свой родительский элемент с «деревом контента»;
   - `Blocks` — упорядоченная коллекция ссылок на экземпляры блоков;
   - `Published` — значение `DateTime`, которое показывает, когда страница была опубликована для просмотра пользователями.

   Ниже приведен полный список свойств, унаследованных от базовых классов:

   - `GenericPage<T>`: `IsStartPage`;
   - `PageBase`: `SiteId`, `ParentId`, `Blocks`, `ContentType`, `SortOrder`, `NavigationTitle`, `IsHidden`, `RedirectUrl`, `RedirectType`, `OriginalPageId`;
   - `RoutedContent`: `Slug`, `Permalink`, `MetaKeywords`, `MetaDescription`, `Route`, `Published`;
   - `Content`: `Id`, `TypeId`, `Title`, `Created`, `LastModified`.

8. Каким образом определить пользовательский тип региона?

   Ответ: вы определяете пользовательский регион для типа страницы, создавая класс со свойствами, дополненными атрибутом `[Field]`, а затем добавляете свойство к типу контента с классом региона в качестве его типа и дополняете его с помощью атрибутов `[Region]` и `[RegionDescription]`:

```
namespace NorthwindCms.Models
{
  [PageType(Title = "Category Page", UseBlocks = false)]
  [PageTypeRoute(Title = "Default", Route = "/category")]
  public class CategoryPage : Page<CategoryPage>
  {
    [Region(Title = "Category detail")]
    [RegionDescription("The details for this category.")]
    public CategoryRegion CategoryDetail { get; set; }
```

9. Как определить маршруты для Piranha CMS?

   Ответ: чтобы определить маршруты для Piranha CMS, в классе `CmsController` добавьте метод действия и дополните его атрибутом `[Route]`:

```
[Route("category")]
public IActionResult Category(Guid id)
```

10. Каким образом получить страницу из базы данных Piranha CMS?

    Ответ: вы извлекаете страницу из базы данных Piranha CMS с помощью зависимостей службы `IApi`:

```
var model = await _api.Pages.GetByIdAsync<Models.CategoryPage>(id);
```

Глава 18. Разработка и использование веб-сервисов

1. Какой базовый класс вы должны унаследовать, чтобы создать класс контроллера для сервиса ASP.NET Core Web API?

   Ответ: чтобы создать класс контроллера для службы Web API ASP.NET Core, вы должны наследовать от класса `ControllerBase`. Не наследуйте от `Controller`, как в MVC, так как этот класс включает такие методы, как `View`, которые используют файлы Razor для визуализации HTML, которые не нужны для веб-службы.

2. Если вы дополнили свой класс `Controller` атрибутом `[ApiController]`, чтобы получить поведение по умолчанию, например автоматический ответ со статусом 400 для недопустимых моделей, то что еще должны сделать?

   Ответ: если вы дополните свой класс контроллера с помощью атрибута `[ApiController]`, то вы также должны вызвать метод `SetCompatibilityVersion` в классе `Startup`.

3. Что вы должны сделать, чтобы указать, какой метод действия контроллера будет выполняться в ответ на HTTP-запрос?

   Ответ: чтобы указать, какой метод действия контроллера будет выполняться в ответ на запрос, вы должны дополнить метод действия атрибутом. Например, чтобы ответить на HTTP-запрос `POST`, дополните метод действия атрибутом `[HttpPost]`.

4. Что вы должны сделать, чтобы указать, какие ответы следует ожидать при вызове метода действия?

   Ответ: чтобы указать, какие ответы следует ожидать при вызове метода действия, дополните метод действия атрибутом `[ProducesResponseType]`, как показано в коде ниже:

```
// GET: api/customers/[id]
[HttpGet("{id}", Name = nameof(Get))] // именованный маршрут
[ProducesResponseType(200, Type = typeof(Customer))]
[ProducesResponseType(404)]
public IActionResult Get(string id)
{
```

5. Перечислите три метода, которые можно вызывать для возврата ответов с разными кодами состояния.

   Ответ: три метода, которые можно вызвать для возврата ответов с разными кодами состояния, включают в себя следующие коды состояния:

   - `Ok` — возвращает код состояния 200 и объект, переданный в тело кода `Ok`;
   - `CreatedAtRoute` — возвращает код состояния 201 и объект, переданный этому методу в теле кода;
   - `NoContentResult` — возвращает код состояния 204 и пустое тело кода;
   - `BadRequest` — возвращает код состояния 400 и необязательное сообщение об ошибке;
   - `NotFound` — возвращает код состояния 404 и необязательное сообщение об ошибке.

6. Перечислите четыре способа тестирования веб-сервиса.

   Ответ: четыре способа тестирования веб-службы включают в себя следующее.

   - Использование браузера для проверки простых HTTP-запросов `GET`.
   - Установку клиентского расширения REST для Visual Studio Code.
   - Установку пакета Swagger NuGet в свой проект веб-службы, включение Swagger и применение пользовательского интерфейса тестирования Swagger.
   - Установку инструмента `Postman` по следующей ссылке: https://www.getpostman.com.

7. Почему не стоит оборачивать `HttpClient` в оператор `using`, чтобы очистить его ресурсы, когда закончите с ним работать, даже если он реализует интерфейс `IDisposable`, и что вы должны задействовать вместо этого?

   Ответ: HttpClient является общим, реентерабельным и частично поточно-ориентированным, поэтому его сложно использовать во многих сценариях. Следует использовать `HttpClientFactory`, представленный в .NET Core 2.1.

8. Что такое CORS и почему важно включить его в веб-сервис?

   Ответ: аббревиатура CORS расшифровывается как Cross-Origin Resource Sharing (совместное использование ресурсов между разными источниками). Это систему важно включить для веб-службы, поскольку для повышения безопасности по умолчанию политика для одного и того же браузера не позволяет коду, загруженному из одного источника, получать доступ к ресурсам, загруженным из другого источника.

9. Как вы можете разрешить клиентам определять, исправен ли ваш веб-сервис, в ASP.NET Core 2.2 и более поздних версиях?

   Ответ: чтобы клиенты могли определить, исправна ли ваша веб-служба, вы можете установить API проверки работоспособности, включая проверки работоспособности базы данных для контекстов данных Entity Framework Core. Чтобы сообщить клиенту подробную информацию, проверка работоспособности может быть расширена.

10. Какие преимущества обеспечивает маршрутизация на основе конечных точек?

    Ответ: маршрутизация в конечной точке обеспечивает улучшенную производительность при маршрутизации и выборе метода действия, а также службу генерации канала.

Глава 19. Разработка интеллектуальных приложений с помощью алгоритмов машинного обучения

1. Каковы четыре основных этапа жизненного цикла машинного обучения?

   Ответ: четыре основных этапа жизненного цикла машинного обучения — анализ проблем, сбор и обработка данных, моделирование и развертывание модели.

2. Каковы три подэтапа шага моделирования?

   Ответ: три подэтапа шага моделирования — идентификация функций, обучение модели и оценка модели.

3. Почему модели необходимо перетренировать после развертывания?

   Ответ: модели должны быть переобучены после развертывания, потому что их прогнозы могут изменяться и становиться хуже, так как данные могут со временем меняться.

4. Почему вы должны разделить свой набор данных на два: для обучения и для тестирования?

   Ответ: вам необходимо разделить свой набор данных на набор данных для обучения и набор данных для тестирования, так как, если вы используете весь набор данных для обучения, у вас не останется данных для тестирования вашей модели и вы не сможете использовать один и тот же набор данных для обучения и тестирования, потому что все будет работать идеально!

5. В чем разница между кластеризацией и классификацией задач машинного обучения?

   Ответ: некоторые из различий между задачами машинного обучения по кластеризации и классификации заключаются в следующем.

   - Кластеризация предназначена для группировки данных, когда вы еще не знаете, есть ли общие черты или какими должны быть метки. Данная техника машинного обучения — неконтролируемая.
   - Классификация предназначена для распределения данных по заранее определенным маркированным группам. Данная техника машинного обучения — контролируемая.

6. Какой класс вы должны создать, чтобы выполнить любую задачу машинного обучения?

   Ответ: для выполнения любой задачи машинного обучения необходимо создать экземпляр `MLContext`.

7. В чем разница между меткой и характеристикой?

   Ответ: характеристика представляет собой ввод, например токенизированный текст обзора Amazon. Метка — это значение, используемое для обучения модели, и может быть выводом, например положительным или отрицательным.

8. Что представляет собой интерфейс `IDataView`?

   Ответ: интерфейс `IDataView` представляет собой входные данные для задачи машинного обучения. Он неизменный, курсивный, лениво оцененный, разнородный и схематизированный.

9. Что представляет собой параметр `count` в атрибуте `[KeyType(count: 10)]`?

   Ответ: параметр `count` представляет собой максимально возможное значение, которое может быть сохранено в столбце.

10. Что означает оценка с матричной факторизацией?

    Ответ: оценка с матричной факторизацией означает вероятность быть положительным результатом, но это не означает вероятность в традиционном смысле. Чем больше значение оценки, тем выше вероятность.

Глава 20. Создание пользовательских веб-интерфейсов с помощью Blazor

1. Какие две основные модели хостинга существуют в Blazor и чем они отличаются?

   Ответ: две основные модели хостинга для Blazor — это Server и WebAssembly.

   Blazor Server выполняет код на стороне сервера. Это означает, что код имеет полный и простой доступ к ресурсам на стороне сервера, таким как базы данных. Это значительно упрощает реализацию функциональности. Обновления пользовательского интерфейса выполняются с использованием SignalR. Это означает, что между браузером и сервером необходимо постоянное соединение, что ограничивает масштабируемость.

   Blazor WebAssembly выполняет код на стороне клиента. Это означает, что код имеет доступ только к ресурсам в браузере, что значительно усложняет реализацию, так как обратный вызов на сервер должен выполняться всякий раз, когда требуются новые данные.

2. Какая дополнительная конфигурация требуется в классе Startup в проекте сайта Blazor Server по сравнению с проектом сайта ASP.NET Core MVC?

   Ответ: в классе `Startup` в методе `ConfigureServices` необходимо вызвать метод `AddServerSideBlazor`, а в методе `Configure` при настройке конечных точек необходимо вызвать `MapBlazorHub` и `MapFallbackToPage`.

3. Одним из преимуществ Blazor является возможность реализации компонентов пользовательского интерфейса с использованием C# и .NET вместо JavaScript. Необходима ли Blazor какая-либо реализация JavaScript?

   Ответ: да. Для компонентов Blazor требуется минимум кода JavaScript. Для Blazor Server этот вопрос можно решить, используя файл `_framework/blazor.server.js`. Для Blazor WebAssembly — файл `_framework/blazor.webassembly.js`. Blazor WebAssembly с PWA также использует файл JavaScript `service-worker.js`.

4. Какова роль файла `App.razor` в проекте Blazor?

   Ответ: файл `App.razor` настраивает маршрутизатор, используемый всеми компонентами Blazor в текущей сборке. Например, он устанавливает общий макет по умолчанию для компонентов, которые соответствуют маршруту, и представление, которое будет использоваться, если совпадения не обнаружены.

5. В чем преимущество использования компонента `<NavLink>`?

   Ответ: преимущество использования компонента `<NavLink>` заключается в том, что он интегрируется с системой маршрутизации Blazor. Затем NavigationManager может использоваться для программного перехода между компонентами.

6. Каким образом осуществляется передача значения компоненту?

   Ответ: вы можете передать значение в компонент, дополнив общедоступное свойство в компоненте атрибутом `[Parameter]`, а затем установив атрибут в компоненте при его использовании:

```
// определение компонента
@code {
  [Parameter]
  public string ButtonText { get; set; };

// использование компонента
<CustomerDetail ButtonText="Create Customer" />
```

7. В чем преимущество использования компонента `<EditForm>`?

   Ответ: преимущество использования компонента `<EditForm>` — сообщения автоматической проверки.

8. Каким образом выполняются некоторые операторы при установленных параметрах?

   Ответ: когда параметры установлены, вы можете выполнить некоторые операторы, определив метод `OnParametersSetAsync` для обработки события.

9. Каким образом выполняются некоторые операторы при появлении компонента?

   Ответ: при обнаружении компонента вы можете выполнить некоторые операторы, определив метод `OnInitializedAsync` для обработки события.

10. Назовите два ключевых различия в классе `Program` между сервером Blazor и проектом Blazor WebAssembly.

    Ответ: два ключевых различия в классе `Program` между Blazor Server и проектом Blazor WebAssembly: 1) использование `WebAssemblyHostBuilder` вместо `Host.CreateDefaultBuilder` и 2) регистрация `HttpClient` с базовым адресом серверной среды.

Глава 21. Разработка кросс-платформенных мобильных приложений

1. В чем разница между Xamarin и Xamarin.Forms?

   Ответ: платформа Xamarin позволяет разработчикам создавать настольные и мобильные приложения с использованием языка C# для Apple iOS (iPhone и iPad), *или* для MacOS, *или* для Google Android. Некоторой бизнес-логикой можно поделиться, но вы в основном будете создавать отдельные проекты для каждой платформы. Xamarin.Forms расширяет Xamarin, чтобы сделать еще более упростить кросс-платформенную разработку мобильных приложений,

разделяя большую часть уровня взаимодействия с пользователем, а также уровня бизнес-логики.

2. Каковы четыре категории компонентов пользовательского интерфейса Xamarin.Forms и что они собой представляют?

   Ответ: ниже перечислены четыре категории компонентов пользовательского интерфейса Xamarin.Forms.

   - *Страницы*: представляют кросс-платформенные экраны мобильных приложений.
   - *Макеты:* представляют структуру комбинации других компонентов пользовательского интерфейса.
   - *Представления*: представляют отдельный компонент пользовательского интерфейса.
   - *Ячейки*: представляют отдельный элемент в виде списка или таблицы.

3. Перечислите четыре типа ячеек.

   Ответ: `TextCell`, `SwitchCell`, `EntryCell` и `ImageCell`.

4. Каким образом вы можете разрешить пользователю выполнять действия с ячейками в `ListView`?

   Ответ: чтобы позволить пользователю выполнить действие с ячейкой в представлении списка, вы можете установить некоторые контекстные действия, представляющие собой пункты меню, которые вызывают событие, как показано в коде ниже:

```
<TextCell Text="{Binding CompanyName}" Detail="{Binding Location}">
  <TextCell.ContextActions>
    <MenuItem Clicked="Customer_Phoned" Text="Phone" />
    <MenuItem Clicked="Customer_Deleted" Text="Delete"
      IsDestructive="True" />
  </TextCell.ContextActions>
</TextCell>
```

5. Как определить сервис зависимостей в целях реализации функциональности, специфичной для платформы?

   Ответ: чтобы определить зависимости службы для реализации функциональности, специфичной для платформы, необходимо выполнить следующее.

   - Определите интерфейс для зависимостей службы.
   - Реализуйте зависимости службы для каждой платформы, например iOS и Android.
   - Дополните зависимости службы с помощью атрибута `[assembly: Dependency]` и передайте тип зависимостей службы в качестве параметра.

- В общем главном проекте получите зависимости службы:

```
var service = DependencyService.Get<IMyService>();
```

6. В каком случае элемент `Entry` используется вместо элемента `Editor`?

   Ответ: используйте `Entry` для одной строки текста и `Editor` — для нескольких строк текста.

7. В чем заключается эффект установки `IsDestructive` в значение `true` для пункта меню в контекстных действиях ячейки?

   Ответ: пункт меню окрашен в красный цвет в качестве предупреждения для пользователя.

8. Когда выполняется вызов методов `PushAsync` и `PopAsync` в мобильных приложениях Xamarin.Forms?

   Ответ: чтобы обеспечить переход между экранами со встроенной поддержкой для возврата к предыдущему экрану, при первом запуске приложения оберните первый экран оператором `NavigationPage`:

```
MainPage = new NavigationPage(new CustomersList());
```

   Чтобы перейти к следующему экрану, нажмите следующую страницу на объекте навигации:

```
await Navigation.PushAsync(new CustomerDetails(c));
```

   Чтобы вернуться к предыдущему экрану, откройте страницу из объекта `Navigation`:

```
await Navigation.PopAsync();
```

9. Как показать всплывающее модальное сообщение с простым выбором кнопок, таких как Yes (Да) или No (Нет)?

   Ответ: чтобы отобразить всплывающее модальное сообщение с простым выбором кнопок, например `Yes` (Да) или `No` (Нет), вы можете вызвать метод `DisplayAlert`:

```
bool response = await DisplayAlert("Apple Survey",
  "Do you like your iPhone 11 Pro?", "Yes", "No");
```

10. В чем заключается функция ATS от Apple и почему она важна?

    Ответ: функция ATS от Apple — это сокращение App Transport Security, и она важна, так как включена по умолчанию и призывает разработчиков использовать передовой опыт, включая безопасные соединения между приложением и веб-службой.

Приложение Б. Разработка настольных Windows-приложений

1. Платформа .NET 5 — кросс-платформенная. Приложения Windows Forms и WPF могут работать на .NET 5. Могут ли эти приложения работать для операционных систем macOS и Linux?

   Ответ: нет. Хотя приложения Windows Forms- и WPF-приложения могут работать на платформе .NET 5, им также необходимо выполнять вызовы Win32 API, поэтому они ограничены работой на основе операционной системы Windows.

2. Как приложение Windows Forms определяет свой пользовательский интерфейс и почему это сложная задача?

   Ответ: с помощью кода языка C#. Это сложно, поскольку приложение Windows Forms, предназначенное для .NET Framework, получает визуальный конструктор с набором инструментов и поддержкой перетаскивания в Visual Studio 2019. Однако приложение, ориентированное на .NET 5, в настоящее время не имеет визуальной поддержки.

3. Как WPF- или UWP-приложение может определять свой пользовательский интерфейс и почему это хорошо для разработчиков?

   Ответ: с помощью языка разметки XAML. Его понять легче, чем код на языке C#.

4. Назовите пять контейнеров макета и варианты расположения их дочерних элементов.

   Ответ

   - `StackPanel` — дочерние элементы размещаются путем их наложения по горизонтали или вертикали.
   - `Grid` — дочерние элементы располагаются по строкам и столбцам в пределах определенной сетки.
   - `DockPanel` — одиночный дочерний элемент размещается в его родительском контейнере путем выравнивания его с помощью значений `Top`, `Left`, `Bottom`, `Right` или заполнения оставшегося пространства.
   - `WrapPanel` — дочерние элементы размещаются путем их наложения горизонтально или вертикально, а затем при необходимости переносятся в другой столбец или строку.
   - `Canvas` — дочерние элементы позиционируются с помощью абсолютных значений `Top` и `Left` или `Bottom` и `Right`.

5. В чем разница между `Margin` и `Padding` для такого элемента, как `Button`?

   Ответ: `Margin` находится за пределами границы, а `Padding` — внутри границы.

6. Как обработчики событий присоединяются к объекту с помощью XAML?

   Ответ: устанавливая атрибут для имени события равным имени метода в классе с фоновым кодом:

```
<Button Clicked="SaveButton_Clicked">
```

7. Как работают стили XAML?

   Ответ: позволяют устанавливать одно или несколько свойств.

8. Где вы можете определить ресурсы?

   Ответ: в любом элементе в зависимости от того, где хотите поделиться этими ресурсами. Для совместного использования ресурсов в приложении определите ресурсы в элементе `<Application.Resources>`. Чтобы поделиться ресурсами только внутри страницы, определите ресурсы в ее элементе `<Page.Resources>`. Что касается одного элемента, такого как `Button`, определите ресурсы в его элементе `<Button.Resources>`.

9. Каким образом вы можете связать свойство одного элемента со свойством другого?

   Ответ: с помощью выражения привязки для свойства, которое ссылается на другой элемент и его свойство:

```
<Slider Value="18" Minimum="18" Maximum="65" Name="slider" />
<TextBlock Text="{Binding ElementName=slider, Path=Value}" />
```

10. Почему вам может потребоваться реализовать интерфейс `IValueConverter`?

    Ответ: это возможно при необходимости выполнить привязку данных между двумя несовместимыми типами, например для преобразования значения `int` в значение `string`, которое определяет путь к изображению, или между значением с плавающей запятой, представляющим индекс массы тела (ИМТ) и цвет, который указывает на хорошее или плохое здоровье.

Б Разработка настольных Windows-приложений

Данное приложение посвящено созданию настольных Windows-приложений с помощью трех технологий: *Windows Forms*, *Windows Presentation Foundation* (WPF) и *универсальной платформы Windows* (Universal Windows Platform, UWP).

Windows Forms и WPF поддерживают использование .NET Core 3.0 и более поздней версии в качестве среды выполнения. Однако текущая поддержка времени разработки ограничена, и потому я рекомендую эти технологии, только если у вас имеются приложения Windows Forms или WPF, которые необходимо перенести на современную платформу .NET.

Значительная часть этого приложения будет посвящена приложениям UWP, которые используют современную среду выполнения Windows Runtime и могут выполнять приложения, созданные с помощью адаптированной версии .NET, которая компилируется в инструкции для ЦП.

Хотя в большей части данного приложения технически не применяется .NET 5, в ноябре 2021 года корпорация Microsoft выпустит версию .NET 6, которая станет единой унифицированной платформой для .NET, используемой всеми моделями приложений, включая ASP.NET Core для веб-разработки, Windows Forms, WPF, а также кросс-платформенные настольные и мобильные приложения с использованием пользовательского интерфейса кросс-платформенных приложений .NET (MAUI).

Более подробную информацию о выборе платформы для создания настольных Windows-приложений можно найти на сайте https://docs.microsoft.com/ru-ru/windows/apps/desktop/choose-your-platform.

Вы познакомитесь с некоторыми новыми функциями пользовательского интерфейса Fluent Design, представленными в обновлении Windows 10 Fall Creators, выпущенном в октябре 2017 года.

В одном приложении я смогу лишь немного рассказать о том, что достижимо с помощью .NET для разработки настольных Windows-приложений. Однако я надеюсь, что у вас появится желание узнать больше о новой возможности миграции

устаревших Windows-приложений на современную платформу .NET и о новой современной технологии UWP с интерфейсом Fluent Design, в том числе о шаблонизируемых элементах управления, привязке данных и анимации!

Несколько важных слов об этом приложении

UWP-приложения хотя и не кросс-платформенные, но задействуют принцип «кросс-девайсной» разработки, если устройства, на которых они запускаются, работают под управлением современных версий операционной системы Windows, то есть они могут работать на настольных ПК, ноутбуках, планшетах и устройствах виртуальной реальности наподобие гарнитуры HP Windows Mixed Reality. Чтобы использовать новейшие функции, такие как XAML Islands, вам понадобится операционная система Windows 10 с обновлением от мая 2019 года и среда разработки Visual Studio 2019 версии 16.3 или новее. Хотя для разработки приложения из этого приложения вам достаточно более старой версии Windows 10, выпущенной в апреле 2018 года.

В этом приложении:

- общие сведения об устаревших платформах Windows-приложений;
- общие сведения о современной платформе Windows;
- разработка современного Windows-приложения;
- использование ресурсов и шаблонов;
- привязка данных.

Общие сведения об устаревших платформах Windows-приложений

С момента выпуска Microsoft Windows 1.0 в 1985 году Windows-приложения можно было разрабатывать только на языке C, вызывая функции одной из трех основных DLL-библиотек: kernel, user и GDI. Как только появилась 32-битная Windows 95, к названиям этих библиотек добавился суффикс 32 и они стали известны как Win32 API.

В 1991 году корпорация Microsoft представила среду разработки Visual Basic, которая предоставила разработчикам возможность визуального перетаскивания элементов управления из набора инструментов в целях создания пользовательского интерфейса для Windows-приложений. Эта возможность стала очень популярной, и среда выполнения Visual Basic по-прежнему входит в состав операционной системы Windows 10.

В 2002 году корпорация Microsoft представила платформу .NET Framework, включающую Windows Forms для разработки Windows-приложений. Программный код

стало можно писать на языках Visual Basic или C#. Технология Windows Forms содержала похожий инструмент визуального перетаскивания, хотя для определения пользовательского интерфейса генерировала программный код на языках C# или Visual Basic, что может усложнить понимание и редактирование напрямую.

В 2006 году корпорация Microsoft представила платформу .NET Framework 3.0, включающую технологию WPF для создания пользовательских интерфейсов с помощью XAML. Эта технология проста для понимания и редактирования.

Поддержка .NET 5 для устаревших Windows-платформ

Пакет SDK, входящий в состав .NET 5, для операционных систем Linux и macOS занимает на диске 332 Мбайт и 337 Мбайт соответственно. Пакет SDK, входящий в состав .NET 5 для операционной системы Windows, занимает на диске 441 Мбайт. Это связано с тем, что в состав входит среда выполнения Windows Desktop, позволяющая запускать устаревшие платформы Windows-приложений Windows Forms и WPF в .NET 5.

Установка Microsoft Visual Studio 2019 для операционной системы Windows

С октября 2014 года корпорация Microsoft разработала версию Visual Studio профессионального уровня, доступную для всех бесплатно, — Community Edition.

Установим ее прямо сейчас.

1. Скачайте среду разработки Microsoft Visual Studio 2019 версии 16.8 или более поздней для операционной системы Windows по следующей ссылке: https://visualstudio.microsoft.com/downloads/.
2. Запустите установщик.
3. На вкладке Workloads (Рабочие нагрузки) выберите следующее:
   - Разработка настольных приложений .NET;
   - Разработка универсальной платформы Windows;
   - Кросс-платформенная разработка на основе .NET Core.
4. Нажмите кнопку Install (Установить) и дождитесь, пока установщик загрузит выбранное программное обеспечение и установит его.
5. После завершения установки нажмите кнопку Launch (Запустить).
6. При первом запуске Visual Studio вам будет предложено войти в систему. Если у вас уже есть учетная запись Microsoft, то можете использовать ее. В противном

случае для регистрации учетной записи вам необходимо пройти по следующей ссылке: signup.live.com.

7. При первом запуске Visual Studio вам будет предложено настроить вашу среду разработки. Для Development Settings (Параметры разработки) выберите язык Visual C#. В качестве цветовой темы я выбрал Blue (Синий), но вы можете выбрать любую другую.

Работа с Windows Forms

Разработку нового приложения Windows Forms начнем с использования инструмента командной строки dotnet. Затем задействуем среду разработки Visual Studio, чтобы создать приложение Windows Forms для .NET Framework в целях имитации устаревшего приложения, а затем перенесем это приложение в .NET 5.

Разработка нового приложения Windows Forms

На следующем примере вы убедитесь, что не стоит создавать приложения Windows Forms для .NET 5. Как правило, в .NET следует переносить только существующие приложения Windows Forms.

Следующие инструкции начинаются с создания папок для хранения файла решения и проекта. Вы можете задействовать другой инструмент для этого вместо командной строки, но вам необходимо по крайней мере создать новое решение и проект Windows Forms, используя инструмент командной строки dotnet в правильной папке.

1. В операционной системе Windows в меню Start (Пуск) откройте Command Prompt (Командная строка).
2. В командной строке введите команды для выполнения следующих задач:
   - переход в папку Code;
   - создание папки WindowsDesktopApps с новым файлом решения;
   - создание подпапки BasicWinForms с новым проектом Windows Forms.

```
cd C:\Code\
mkdir WindowsDesktopApps
cd WindowsDesktopApps
dotnet new sln
mkdir BasicWinForms
cd BasicWinForms
dotnet new winforms
```

3. Запустите программу Visual Studio 2019 и нажмите кнопку Open a project or solution (Открыть проект или решение).

4. Выберите каталог C:\Code\WindowsDesktopApps. Затем выберите файл решения `WindowsDesktopApps.sln` и нажмите кнопку Open (Открыть).
5. Выберите File ▶ Add ▶ Existing Project (Файл ▶ Добавить ▶ Существующий проект). Выберите проект `BasicWinForms.csproj` и нажмите кнопку Open (Открыть).
6. Выберите Project ▶ Edit Project File (Проект ▶ Свойства файла проекта) или щелкните правой кнопкой мыши на панели Solution Explorer (Обозреватель решений) и выберите пункт меню Edit Project File (Свойства файла проекта). Затем обратите внимание, что Target framework (Целевая платформа) — net5.0-windows, используется технология Windows Forms, а Output type (Тип вывода) — Windows-приложение:

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

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>
  </PropertyGroup>

</Project>
```

7. Закройте вкладку.
8. Дважды щелкните на панели Solution Explorer (Обозреватель решений) или щелкните правой кнопкой мыши на файле `Form1.cs` и выберите кнопку Open (Открыть). Обратите внимание: на момент написания книги в сентябре 2020 года конструктор Windows Forms по умолчанию не поддерживает .NET 5 (хотя версия 16.8, выпущенная в ноябре 2020 года, должна работать), как показано на рис. Б.1.

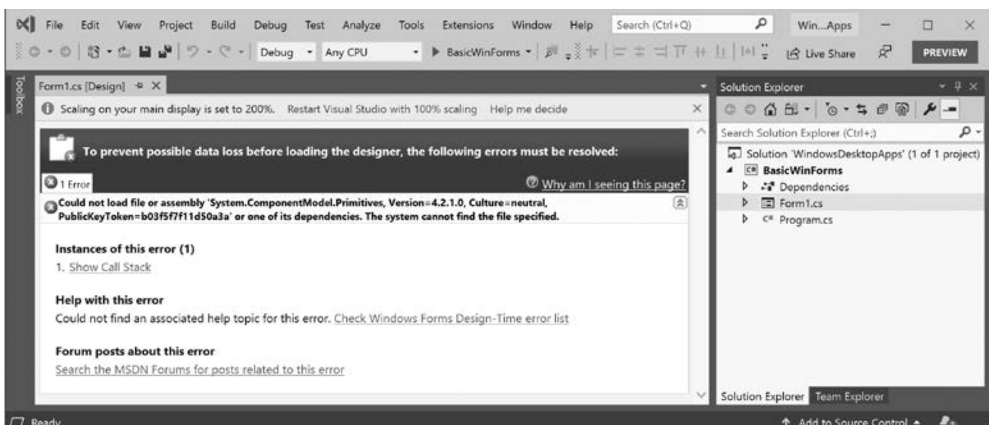


Рис. Б.1

Отслеживать прогресс конструктора Windows Forms можно на сайте https://github.com/dotnet/winforms/tree/master/docs/designer-releases.

9. Выберите команду меню Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки) или нажмите сочетание клавиш Ctrl+F5. Вы увидите окно с изменяемым размером с заголовком Form1, а затем для выхода из приложения нажмите кнопку закрытия в правом верхнем углу.

Обзор нового приложения Windows Forms

Рассмотрим код в пустом приложении Windows Forms.

1. На панели Solution Explorer (Обозреватель решений) найдите и откройте файл `Program.cs`. Обратите внимание: он похож на консольное приложение с методом точки входа `Main`, создает экземпляр класса `Form1` и запускает его, как показано ниже:

```
static class Program
{
  [STAThread]
  static void Main()
  {
    Application.SetHighDpiMode(HighDpiMode.SystemAware);
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
  }
}
```

2. На панели Solution Explorer (Обозреватель решений) разверните файл `Form1.cs`, откройте класс `Form1` и обратите внимание, что он является частичным и что в его конструкторе вызывается метод `InitializeComponent`:

```
public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();
  }
}
```

3. На панели Solution Explorer (Обозреватель решений) разверните файл `Form1.cs`, найдите и откройте файл `Form1.Designer.cs`, разверните раздел Windows Form Designer generated code (Код, сгенерированный конструктором Windows Form Designer) и обратите внимание, что код, описывающий пустую форму, слишком путанный, чтобы понять и вручную менять его:

```
private void InitializeComponent()
{
  this.components = new System.ComponentModel.Container();
```

```
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(800, 450);
    this.Text = "Form1";
  }
```

4. Закройте все окна редактирования.

Миграция устаревшего приложения Windows Forms

В данном примере вы добавите приложение Windows Forms для .NET Framework, чтобы можно было использовать визуальный конструктор Windows Forms, а затем перенесете его в .NET 5.

1. В программе Visual Studio 2019 выберите File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Создать проект).
2. В строке поиска введите `windows forms` и выберите пункт Windows Forms App (.NET Framework). Нажмите кнопку Next (Далее).
3. В качестве Project name (Имя проекта) введите `LegacyWinForms` и нажмите кнопку Create (Создать).
4. Выберите View ▸ Toolbox (Вид ▸ Панель инструментов) или нажмите сочетание клавиш Ctrl+W, X, а затем закрепите панель инструментов. Обратите внимание: у вас могут быть разные привязки клавиш. Если это так, то вы можете выбрать Tools ▸ Import and Export Settings (Инструменты ▸ Импорт и экспорт настроек), затем сбросьте настройки, чтобы установить язык Visual C# и использовать те же привязки клавиш, которые применяются в данном приложении.
5. На вкладке Toolbox (Панель инструментов) разверните пункт меню Common Controls (Общие элементы управления).
6. Перетащите некоторые элементы управления, такие как Button, MonthCalendar, Label и TextBox на Form1 (рис. Б.2).

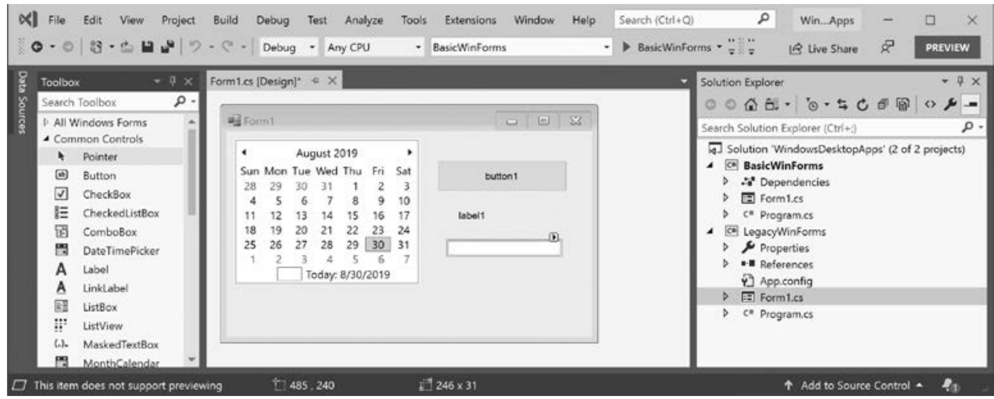


Рис. Б.2

7. Выберите команду меню View ▸ Properties Window (Вид ▸ Окно свойств) либо нажмите сочетание клавиш Ctrl+W, P или клавишу F4.
8. Выберите кнопку (элемент управления button), измените ее свойство (Name) на `btnGoToChristmas` и свойство Text на `Go to Christmas`.
9. Дважды нажмите кнопку. Появится обработчик событий. Далее введите операторы, чтобы установить календарь и выбрать в нем день Рождества 2020 года, как показано ниже:

```
private void btnGoToChristmas_Click(object sender, EventArgs e)
{
  DateTime christmas = new DateTime(2020, 12, 25);
  monthCalendar1.SelectionStart = christmas;
  monthCalendar1.SelectionEnd = christmas;
}
```

10. На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на решении и выберите Set StartUp Projects (Настроить проекты для запуска).
11. В диалоговом окне Solution 'WindowsDesktopApps' Properties Pages (Страницы свойств решения WindowsDesktopApps) выберите Current selection (Текущий выбор) для Startup Project (Проект для запуска) и нажмите кнопку OK.
12. На панели Solution Explorer (Обозреватель решений) выберите проект LegacyWinForms и обратите внимание: его имя выделено полужирным шрифтом — таким образом показывается, что это текущий выбор.
13. Выберите команду меню Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки) или нажмите сочетание клавиш Ctrl+F5. Нажмите кнопку и обратите внимание: календарь анимируется, чтобы выбрать дату Рождества. Затем для выхода из приложения нажмите кнопку закрытия в правом верхнем углу.

Миграция приложения Windows Forms

Теперь мы можем выполнить перенос в проект .NET 5. Однако помните, что это временно. Как только конструктор Windows Forms (ныне тестируется) будет перенесен на .NET 5 и станет доступен в будущем выпуске Visual Studio 2019, миграция не потребуется.

1. Перетащите `Form1` из папки LegacyWinForms в папку BasicWinForms, которая предложит вам перезаписать следующие файлы:
   - `Form1.cs`;
   - `Form1.Designer.cs`;
   - `Form1.resx`.

2. В проекте `BasicWinForms` отредактируйте файл `Program.cs`, чтобы указать устаревшее пространство имен при создании экземпляра и запуске `Form1`, как показано ниже:

```
Application.Run(new LegacyWinForms.Form1());
```

3. На панели Solution Explorer (Обозреватель решений) выберите проект BasicWinForms.
4. Выберите Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки) или нажмите сочетание клавиш Ctrl+F5. Нажмите кнопку и обратите внимание: календарь анимируется, чтобы выбрать день Рождества. Затем закройте приложение.
5. Закройте проект и решение.

Миграция приложений WPF в .NET 5

Если вам необходимо перенести существующие приложения WPF из .NET Framework в .NET 5, то, как и при миграции проектов Windows Forms, это возможно, но трудновыполнимо. Обновления будут доступны в течение следующего года, в частности в грядущих версиях Visual Studio 2019.

Более подробную информацию о переносе приложений WPF можно получить на сайте https://devblogs.microsoft.com/dotnet/migrating-asample-wpf-app-to-net-core-3-part-1/.

Миграция устаревших приложений с помощью Windows Compatibility Pack (пакет обеспечения совместимости Windows)

Пакет обеспечения совместимости Windows предоставляет доступ к API, которые ранее были доступны только для .NET Framework.

Более подробную информацию о пакете обеспечения совместимости Windows можно найти на сайте https://devblogs.microsoft.com/dotnet/announcing-the-windows-compatibility-pack-for-net-core/.

Общие сведения о современной платформе Windows

Корпорация Microsoft продолжает совершенствовать платформу Windows, включающую в себя такие технологии для создания современных приложений, как Universal Windows Platform (UWP) и Fluent Design System.

Общие сведения об универсальной платформе Windows

Универсальная платформа Windows (UWP) — новейшее решение Microsoft для разработки приложений, предназначенных для выполнения в операционных системах семейства Windows. Универсальная платформа Windows обеспечивает гарантированный уровень API для различных типов устройств. Вы создаете один пакет приложения, который загружается в единое хранилище и распространяется на все типы устройств, поддерживаемые приложением. К подобным устройствам относятся настольные компьютеры, ноутбуки и планшеты под управлением операционных систем Windows 10, Xbox One и более поздних версий, а также Mixed Reality Headsets (гарнитуры смешанной реальности), например Microsoft HoloLens.

Универсальная платформа Windows содержит механизмы, позволяющие определить возможности используемого устройства, а затем задействовать дополнительные функции вашего приложения, чтобы применять его максимально эффективно.

Универсальная платформа Windows предоставляет посредством XAML панели макетов, которые адаптируют способ отображения дочерних элементов управления, чтобы максимально использовать устройство, на котором они в данный момент работают. Это эквивалент адаптивного дизайна веб-страниц для приложений Windows. Кроме того, они содержат триггеры визуального состояния для изменения макета на основе динамических изменений, таких как горизонтальная или вертикальная ориентация планшета.

Fluent Design System

Новая система дизайна от корпорации Microsoft под названием Fluent Design System будет поставляться в несколько этапов, чтобы помочь разработчикам постепенно перейти от традиционных стилей пользовательского интерфейса к более современным.

Этап 1, доступный в обновлении Windows 10 Fall Creators, выпущенном в октябре 2017 года, и улучшенный при выпусках последующих этапов, входивших в выпускающиеся два раза в год обновления Windows 10, включал следующие функции:

- акриловый материал;
- связанные анимации;
- параллакс-эффекты (ParallaxView);
- подсвечивание.

Заполнение элементов пользовательского интерфейса акриловыми кистями

Акриловый материал (акрил) — полупрозрачная кисть с эффектом размытия, с помощью которой можно заполнять элементы пользовательского интерфейса, чтобы добавить глубину и перспективу в ваши приложения. Акрил может показать, что находится на заднем плане позади приложения, или элементы в приложении, находящиеся за панелью. Для акрила можно настроить различные цветов и уровень прозрачности.

Более подробно об использовании акрила можно прочитать на сайте https://docs.microsoft.com/ru-ru/windows/uwp/design/style/acrylic.

Связывание элементов пользовательского интерфейса с помощью анимации

При навигации по UI анимация элементов, помогающая проследить связь между экранами, помогает пользователям понять, где они находятся и как взаимодействовать с вашим приложением.

Более подробную информацию о применении связанной анимации можно найти на сайте https://docs.microsoft.com/ru-ru/windows/uwp/design/motion/connected-animation[1].

Параллакс-эффекты и подсвечивание

Параллакс-эффекты (эффекты Parallax) — это фоны, часто представленные изображениями, которые во время прокрутки перемещаются медленнее, чем передний план, чтобы создать ощущение глубины и придать вашим приложениям современный вид.

Более подробную информацию об эффектах Parallax можно получить на сайте https://docs.microsoft.com/ru-ru/windows/uwp/design/motion/parallax.

Подсвечивание помогает пользователю понять, какой из визуальных элементов в UI интерактивный. Элемент *подсвечивается*, когда пользователь наводит на него указатель мыши.

---

[1] На момент перевода книги в статье по ссылке термин Connected animation переведен как «подключенная анимация», однако этот перевод, возможно, сделан машинным переводчиком, что прямо указано вверху страницы. «Связанная аминация» лучше описывает суть явления.

Более подробную информацию о подсвечивании можно найти на сайте https://docs.microsoft.com/ru-ru/windows/uwp/design/style/reveal.

Разработка современного Windows-приложения

В первую очередь начнем с разработки простого приложения UWP с некоторыми общими элементами управления и современными функциями Fluent Design, такими как акриловый материал.

Включение режима разработчика

Для разработки UWP-приложения необходимо в операционной системе Windows 10 включить режим разработчика.

1. Выберите Start ▸ Settings ▸ Update & Security ▸ For developers (Пуск ▸ Параметры ▸ Обновление и безопасность ▸ Для разработчиков). Затем выберите пункт Developer mode (Режим разработчика).
2. Примите предупреждение о том, что даная операция could expose your device and personal data to security risk or harm your device (может подвергнуть ваше устройство и личные данные угрозе безопасности или повредить ваше устройство). Затем закройте приложение Settings (Параметры).

Разработка UWP-приложений

Теперь добавьте новый проект UWP-приложения в свое решение.

1. В программе Visual Studio 2019 откройте решение `WindowsDesktopApps`.
2. Выберите File ▸ Add ▸ New Project (Файл ▸ Добавить ▸ Создать проект).
3. В диалоговом окне Add a new project (Добавить новый проект) в строке поиска введите uwp, выберите шаблон Blank App (Universal Windows) (Пустое приложение (Universal Windows)) для языка C# и нажмите кнопку Next (Далее). Убедитесь, что выбрали шаблон для языка C#, а не Visual Basic!
4. В качестве имени проекта введите `FluentUwpApp` и нажмите кнопку Create (Создать).
5. В диалоговом окне New Universal Windows Project (Новый универсальный проект для Windows), в раскрывающемся списке выберите последнюю версию Windows 10 как Target Version (Целевую версию) и Minimum Version (Минимальную версию) и нажмите кнопку OK.

Поскольку Fluent Design System была впервые выпущена вместе с обновлением Window 10 Fall Creators (10.0; сборка 16299), вы сможете выбрать ее для поддержки функций, которые мы рассмотрим в этом приложении. Но на данный момент во избежание неожиданных несовместимостей лучше всего использовать последнюю версию. Разработчики, создающие UWP-приложения для широкой аудитории, должны выбирать последнюю версию Windows 10 в раскрывающемся списке Minimum Version (Минимальная версия). Разработчики, создающие корпоративные приложения, должны выбрать более старую минимальную версию. Сборка 10240 была выпущена в июле 2015 года и отлично подходит для максимальной совместимости. Однако у вас не будет доступа к современным функциям, таким как Fluent Design System.

6. На панели Solution Explorer (Обозреватель решений) дважды щелкните кнопкой мыши на файле `MainPage.xaml`, чтобы открыть его для редактирования. Вы увидите панель дизайна XAML, отображающую графический вид приложения, и панель XAML (рис. Б.3). Вы можете управлять этими панелями следующим образом:
   - панели XAML расположены по горизонтали, но вы можете переключиться на режим отображения по вертикали или свернуть одну из панелей, нажав соответствующую кнопку в правой части интерфейса между панелями;
   - вы можете поменять панели местами, нажав кнопку ↑↓ между панелями;
   - вы можете прокручивать и изменять масштаб обеих панелей.

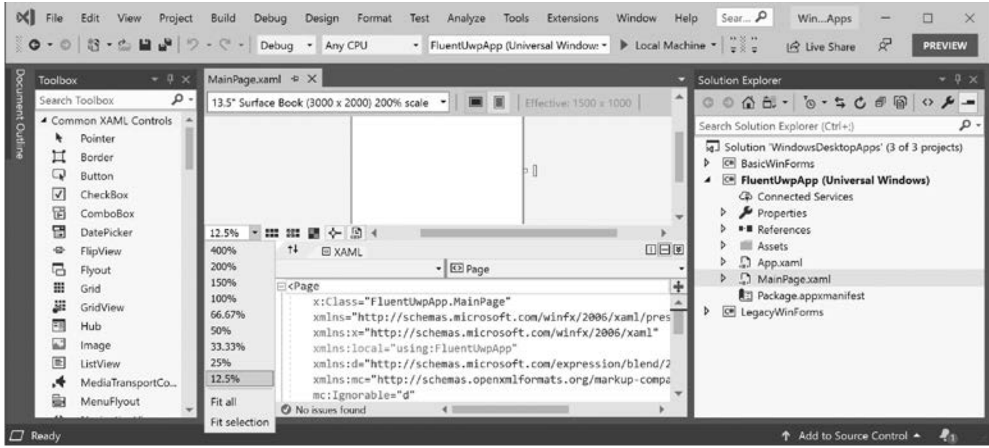


Рис. Б.3

7. На панели Design (Конструктор) измените устройство на 13,3" Desktop (1280 x 720) 100% scale и увеличьте до 100 %.

8. Выберите команду меню View ▸ Toolbox (Вид ▸ Панель инструментов) или нажмите сочетание клавиш Ctrl+W, X. Обратите внимание: панель содержит категории Common XAML Controls (Типовые элементы управления XAML), All XAML Controls (Все элементы управления XAML) и General (Общие).
9. В верхней части панели находится поле поиска. Введите в него буквы bu и обратите внимание, как фильтруется список элементов управления.
10. Перетащите элемент управления Button из панели Toolbox (Панель инструментов) в окно панели Design (Конструктор).
11. Измените размер элемента управления Button. Для этого необходимо нажать и удерживать кнопку мыши на любом из восьми прямоугольных маркеров, затем перетащить с помощью мыши.

    Обратите внимание: для кнопки зафиксированы ширина и высота, а также поля сверху (40 единиц) и слева (30 единиц) по отношению к положению и размерам по сетке (рис. Б.4).

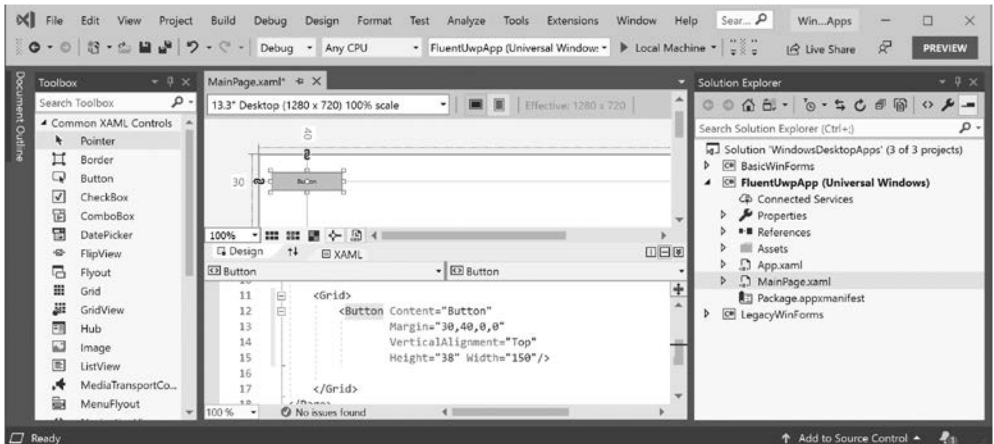


Рис. Б.4

Хотя вы можете перетаскивать элементы управления вручную, для компоновки макета лучше использовать панель XAML, где вы сможете расположить элементы относительно друг друга и реализовать более адаптивный дизайн.

12. На панели XAML найдите элемент `Button` и удалите его.
13. На панели XAML в элементе `Grid` добавьте следующую разметку:

```
<Button Margin="6" Padding="6" Name="ClickMeButton">
  Click Me
</Button>
```

14. Установите масштаб равным 50 % и обратите внимание, что кнопка Button автоматически меняет размер согласно своему содержимому, то есть тексту Click Me,

выравнивается по вертикали по центру и по горизонтали влево, даже если вы переключаетесь между вертикальной и горизонтальной раскладками телефона.

15. На панели XAML удалите элемент `Grid` и измените код, чтобы обернуть элемент `Button` элементом `StackPanel` с ориентацией по горизонтали, который находится в элементе `StackPanel` с ориентацией по вертикали (установленной по умолчанию). Обратите внимание на изменение компоновки макета, то есть на расположение в левом верхнем углу доступного пространства:

```
<StackPanel>
  <StackPanel Orientation="Horizontal" Padding="4"
              Background="LightGray" Name="toolbar">
    <Button Margin="6" Padding="6" Name="ClickMeButton">
      Click Me
    </Button>
  </StackPanel>
</StackPanel>
```

16. Измените код обработчика элемента `Button`, чтобы задать новый обработчик для события `Click` (рис. Б.5).

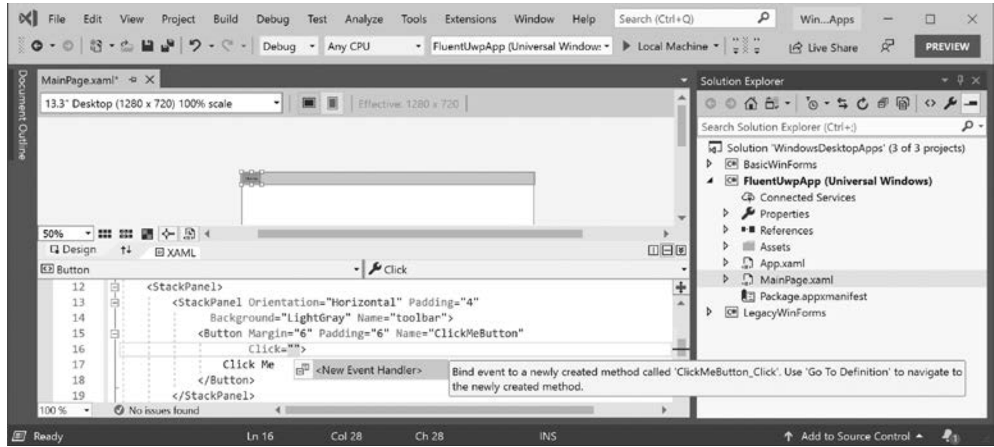


Рис. Б.5

17. Щелкните правой кнопкой мыши на имени обработчика событий и выберите пункт Go to Definition (Перейти к определению) в контекстном меню или нажмите клавишу F12.

18. Добавьте к обработчику событий оператор, устанавливающий содержимое элемента `Button` равным текущему времени:

```
private void ClickMeButton_Click(object sender, RoutedEventArgs e)
{
  ClickMeButton.Content = DateTime.Now.ToString("hh:mm:ss");
}
```

19. Выберите Build ▸ Configuration Manager (Сборка ▸ Диспетчер конфигурации) для проекта FluentUwpApp. Отметьте пункты Build (Сборка) и Deploy (Развертывание), выберите Platform of x64, а затем нажмите кнопку Close (Закрыть), как показано на рис. Б.6.

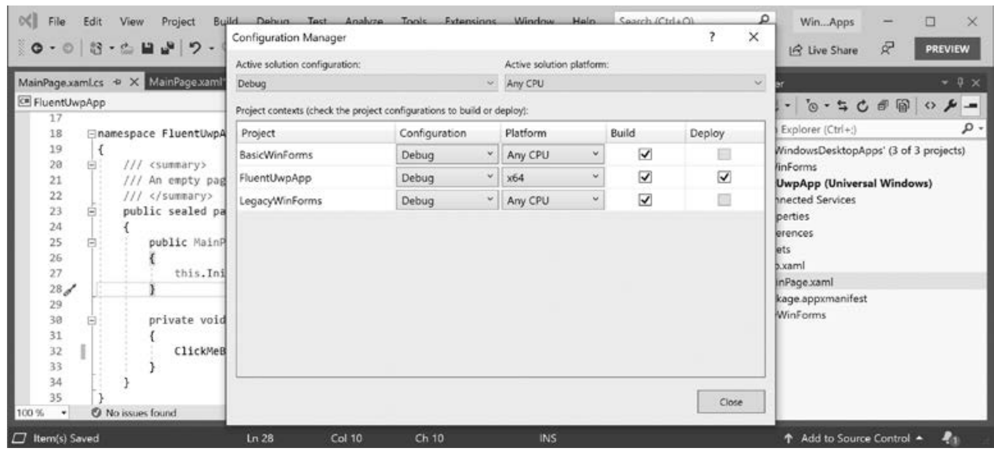


Рис. Б.6

20. Запустите приложение, выбрав Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки) или нажав сочетание клавиш Ctrl+F5.
21. Нажмите кнопку Click Me. Обратите внимание: каждый раз, когда вы ее нажимаете, текст меняется на текущее время.

Типовые элементы управления и акриловая кисть

Теперь изучим некоторые типовые элементы управления и рисования акриловыми кистями.

1. Найдите и откройте файл `MainPage.xaml`. Установите фон панели `StackPanel` для применения системной акриловой кисти для окон и добавьте на панель после кнопки элементы для ввода пользователем имени, как показано ниже (выделено полужирным шрифтом):

```
<StackPanel
  Background="{ThemeResource SystemControlAcrylicWindowBrush}">
  <StackPanel Orientation="Horizontal" Padding="4"
              Background="LightGray" Name="toolbar">
    <Button Margin="6" Padding="6" Name="ClickMeButton"
            Click="ClickMeButton_Click">
```

```
        Click Me
      </Button>
      <TextBlock Text="First name:"
        VerticalAlignment="Center" Margin="4" />
      <TextBox PlaceholderText="Enter your name"
        VerticalAlignment="Center" Width="200" />
    </StackPanel>
  </StackPanel>
```

2. Запустите приложение, выбрав Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки). Обратите внимание на тонированный акриловый материал, показывающий зеленую палатку и оранжевый закат над горами, изображенные на стандартных обоях Windows 10, сквозь фон окна приложения, как показано на рис. Б.7.

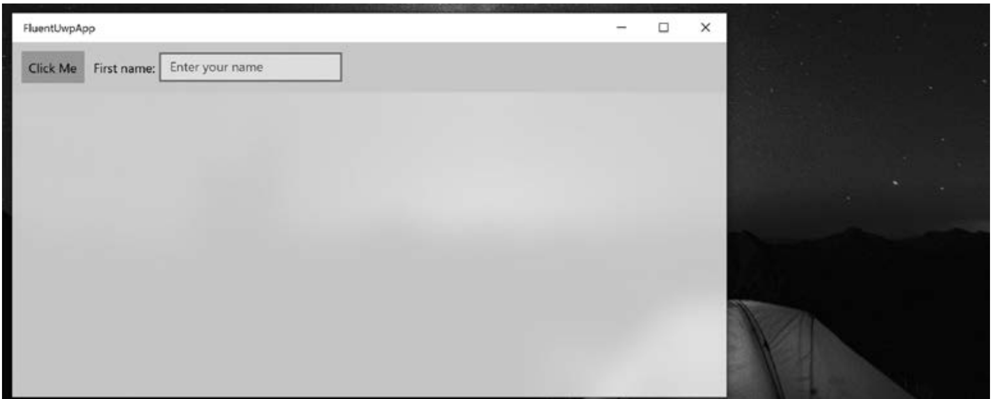


Рис. Б.7

Акрил использует большое количество системных ресурсов, поэтому отключается автоматически, если приложение теряет фокус или ваше устройство разряжено.

Подсвечивание

Подсвечивание включено по умолчанию для одних элементов управления, таких как `ListView` и `NavigationView`. Для других элементов вы можете включить его, применив стиль темы. В первую очередь мы добавим программный код XAML, чтобы определить пользовательский интерфейс калькулятора, состоящий из набора кнопок. Затем добавим обработчик события для события `Loaded` страницы, чтобы мы могли применить стиль темы Reveal и другие свойства, перечислив кнопки вместо необходимости вручную устанавливать атрибуты для каждого из них в программном коде XAML.

1. Откройте файл `MainPage.xaml`. Добавьте новую горизонтальную панель в стеке под панелью, которая используется в качестве панели инструментов, и для определения калькулятора добавьте сетку с кнопками:

```
<StackPanel
  Background="{ThemeResource SystemControlAcrylicWindowBrush}">
  <StackPanel Orientation="Horizontal" Padding="4"
              Background="LightGray" Name="toolbar">
    <Button Margin="6" Padding="6" Name="ClickMeButton"
            Click="ClickMeButton_Click">
      Click Me
    </Button>
    <TextBlock Text="First name:"
      VerticalAlignment="Center" Margin="4" />
    <TextBox PlaceholderText="Enter your name"
      VerticalAlignment="Center" Width="200" />
  </StackPanel>
  <StackPanel Orientation="Horizontal">
    <Grid Background="DarkGray" Margin="10"
          Padding="5" Name="gridCalculator">
      <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
      </Grid.RowDefinitions>
      <Button Grid.Row="0" Grid.Column="0" Content="X" />
      <Button Grid.Row="0" Grid.Column="1" Content="/" />
      <Button Grid.Row="0" Grid.Column="2" Content="+" />
      <Button Grid.Row="0" Grid.Column="3" Content="-" />
      <Button Grid.Row="1" Grid.Column="0" Content="7" />
      <Button Grid.Row="1" Grid.Column="1" Content="8" />
      <Button Grid.Row="1" Grid.Column="2" Content="9" />
      <Button Grid.Row="1" Grid.Column="3" Content="0" />
      <Button Grid.Row="2" Grid.Column="0" Content="4" />
      <Button Grid.Row="2" Grid.Column="1" Content="5" />
      <Button Grid.Row="2" Grid.Column="2" Content="6" />
      <Button Grid.Row="2" Grid.Column="3" Content="." />
      <Button Grid.Row="3" Grid.Column="0" Content="1" />
      <Button Grid.Row="3" Grid.Column="1" Content="2" />
      <Button Grid.Row="3" Grid.Column="2" Content="3" />
      <Button Grid.Row="3" Grid.Column="3" Content="=" />
    </Grid>
  </StackPanel>
</StackPanel>
```

2. В элемент Page добавьте новый обработчик событий для Loaded:

```
<Page
  ...
  Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
  Loaded="Page_Loaded">
```

3. Щелкните правой кнопкой мыши на Page_Loaded и в контекстном меню выберите пункт Go to Definition (Перейти к определению) или нажмите клавишу F12.
4. В метод Page_Loaded добавьте операторы для циклического перебора всех кнопок калькулятора, задав для них одинаковый размер, и примените стиль Reveal, как показано ниже:

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
  Style reveal = Resources.ThemeDictionaries[
    "ButtonRevealStyle"] as Style;

  foreach (Button b in gridCalculator.Children.OfType<Button>())
  {
    b.FontSize = 24;
    b.Width = 54;
    b.Height = 54;
    b.Style = reveal;
  }
}
```

5. Запустите приложение, выбрав Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки). Обратите внимание, что кнопки калькулятора плоские, серого цвета.
6. Когда пользователь наводит указатель мыши на правый нижний угол кнопки 8, мы видим, что кнопка подсвечивается, как и фрагменты окружающих кнопок (рис. Б.8).
7. Закройте приложение.

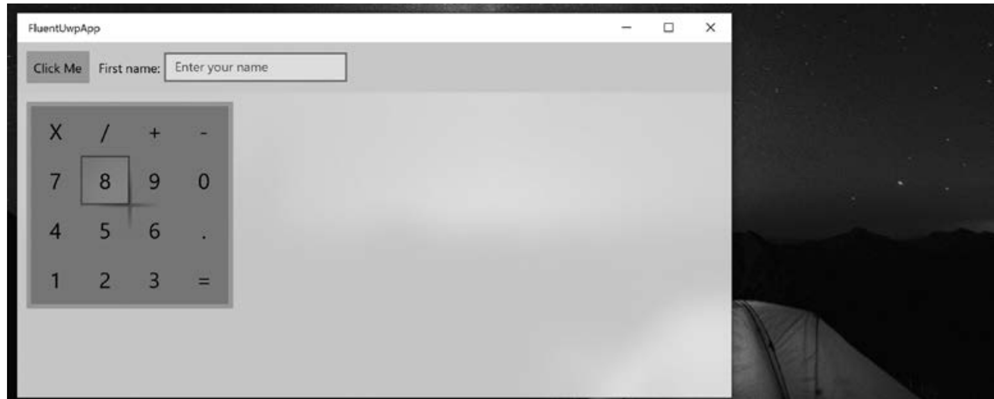


Рис. Б.8

Реализация эффекта подсветки несколько изменялась по мере выхода последних полугодовых обновлений для операционной системы Windows 10, поэтому эффект в вашем приложении может немного отличаться.

Установка дополнительных элементов управления

В дополнение к десяткам встроенных элементов управления вы можете установить дополнительные элементы с помощью пакета NuGet. Один лучших пакетов — это *UWP Community Toolkit*.

Больше информации о UWP Community Toolkit можно получить на сайте https://docs.microsoft.com/en-us/windows/communitytoolkit/.

Один из элементов управления, содержащихся в этом пакете, — редактор для Markdown.

Больше информации о проекте Markdown можно найти на сайте https://daringfireball.net/projects/markdown/.

Установим UWP Community Toolkit и рассмотрим некоторые из его элементов управления.

1. В проекте `FluentUwpApp` щелкните правой кнопкой мыши на References (Ссылки) и выберите пункт Manage NuGet Packages (Управление пакетами NuGet).
2. Нажмите кнопку Browse (Обзор), найдите `Microsoft.Toolkit.Uwp.UI.Controls` и нажмите кнопку Install (Установить).
3. Проанализируйте изменения и примите лицензионное соглашение.
4. Чтобы восстановить пакеты (если это необходимо), выберите Build ▶ Build FluentUwpApp (Сборка ▶ Сборка FluentUwpApp).
5. Откройте файл `MainPage.xaml` и в элементе `Page` импортируйте пространство имен набора инструментов с помощью префикса `kit`:

```
<Page
  ...
  xmlns:kit="using:Microsoft.Toolkit.Uwp.UI.Controls"
  mc:Ignorable="d"
  Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
  Loaded="Page_Loaded">
```

6. Добавьте внутри второй горизонтальной панели после сетки калькулятора текстовое поле и текстовый блок с поддержкой Markdown так, чтобы их дан-

ные были связаны и текст в текстовом поле стал источником для отрисовки Markdown, как показано ниже (выделено полужирным шрифтом):

```
<StackPanel Orientation="Horizontal">
  <Grid Background="DarkGray" Margin="10"
        Padding="5" Name="gridCalculator">
  ...
  </Grid>
  <TextBox Name="markdownSource" Text="# Welcome"
           Header="Enter some Markdown text:"
           VerticalAlignment="Stretch" Margin="5"
           AcceptsReturn="True" />
  <kit:MarkdownTextBlock Margin="5"
    Text="{Binding ElementName=markdownSource, Path=Text}"
    VerticalAlignment="Stretch" HorizontalAlignment="Stretch" />
</StackPanel>
```

7. Запустите приложение, выбрав Debug ▸ Start Without Debugging (Отладка ▸ Запуск без отладки). Вы увидите, что пользователь может ввести в текстовое поле синтаксис Markdown и он отобразится в текстовом блоке Markdown, как показано на рис. Б.9.

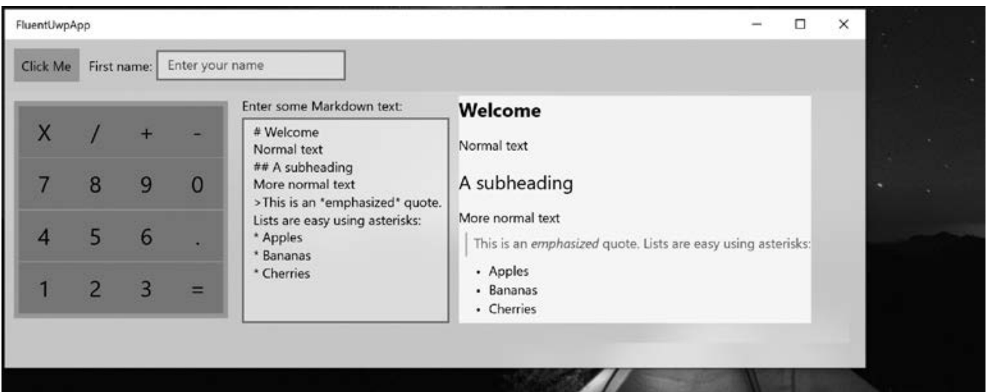


Рис. Б.9

Использование ресурсов и шаблонов

При разработке графических пользовательских интерфейсов вам часто понадобится задействовать ресурсы, например кисти, для окрашивания фона элементов управления. Эти ресурсы можно определить в одном расположении и сделать доступными всему приложению.

Общий доступ к ресурсам

Располагать определение общих ресурсов лучше всего на уровне приложения.

1. На панели Solution Explorer (Обозреватель решений) найдите и откройте файл `App.xaml`.
2. Добавьте следующую разметку к существующей в элементе `<Application>`, чтобы определить кисть с линейным градиентом с ключом `rainbow`, как показано ниже (выделено полужирным шрифтом):

```
<Application
  x:Class="FluentUwpApp.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:FluentUwpApp">

  <Application.Resources>
    <LinearGradientBrush x:Key="rainbow">
      <GradientStop Color="Red" Offset="0" />
      <GradientStop Color="Orange" Offset="0.1" />
      <GradientStop Color="Yellow" Offset="0.3" />
      <GradientStop Color="Green" Offset="0.5" />
      <GradientStop Color="Blue" Offset="0.7" />
      <GradientStop Color="Indigo" Offset="0.9" />
      <GradientStop Color="Violet" Offset="1" />
    </LinearGradientBrush>
  </Application.Resources>

</Application>
```

3. Выберите команду меню Build ▸ Build FluentUwpApp (Сборка ▸ Сборка FluentUwpApp).
4. В файле `MainPage.xaml` измените элемент `StackPanel` с именем `toolbar`, чтобы изменить его фон с `LightGray` на использование статического ресурса — кисти `rainbow`, как показано ниже (выделено полужирным шрифтом):

```
<StackPanel Orientation="Horizontal" Padding="4"
            Background="{StaticResource rainbow}" Name="toolbar">
```

5. Когда вы вводите ссылку на статический ресурс, подсказка IntelliSense отобразит ваш ресурс `rainbow` и встроенные ресурсы (рис. Б.10).

Ресурс может быть экземпляром любого объекта. Чтобы предоставить к нему общий доступ в приложении, определите его в файле App.xaml и присвойте ему уникальный ключ. Чтобы установить свойству элемента значение ресурса, используйте ключевое слово {StaticResource key}.

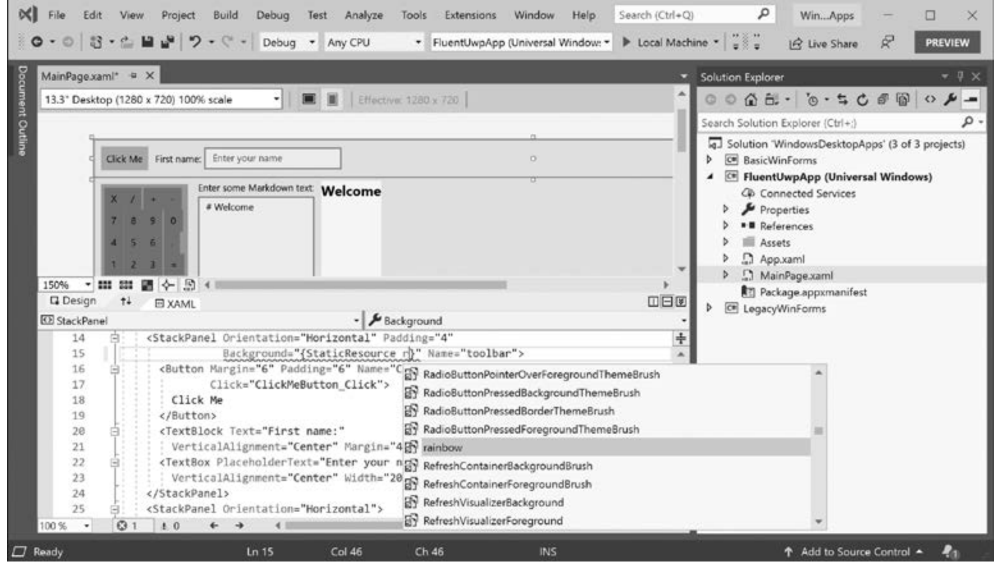


Рис. Б.10

Ресурсы могут быть определены и сохранены в любом элементе XAML, а не только на уровне приложения. Например, если ресурс необходим лишь на `MainPage`, то его можно определить на `MainPage`. Вы также можете динамически загружать файлы XAML во время выполнения.

Более подробную информацию о системе управления ресурсами можно найти на сайте https://docs.microsoft.com/en-us/windows/uwp/app-resources/.

Замена шаблона элемента управления

Вы можете изменить внешний вид элемента управления, заменив его шаблон по умолчанию. Шаблон управления по умолчанию для кнопки плоский и прозрачный.

Один из наиболее часто используемых общих ресурсов — `Style`, который позволяет определять сразу несколько свойств. Если стиль имеет уникальный ключ, то он должен указываться явно, как и в предыдущем примере с линейным градиентом. В случае отсутствия ключ будет применен автоматически на основе свойства `TargetType`.

1. В файле `App.xaml` добавьте следующую разметку в элемент `<Application.Resources>`. Обратите внимание: элемент `Style` автоматически установит для

всех элементов управления, имеющих тип `TargetType`, то есть кнопок, свойство `Template` равным определяемому нами шаблону, как показано ниже (выделено полужирным шрифтом):

```
<Application.Resources>

  <LinearGradientBrush x:Key="rainbow">
  ...
  </LinearGradientBrush>

  <ControlTemplate x:Key="DarkGlassButton"
                   TargetType="Button">
    <Border BorderBrush="#FFFFFFFF"
            BorderThickness="1,1,1,1" CornerRadius="4,4,4,4">
      <Border x:Name="border" Background="#7F000000"
              BorderBrush="#FF000000"
              BorderThickness="1,1,1,1"
              CornerRadius="4,4,4,4">
        <Grid>
          <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
          </Grid.RowDefinitions>
          <Border Opacity="0"
                  HorizontalAlignment="Stretch" x:Name="glow"
                  Width="Auto" Grid.RowSpan="2"
                  CornerRadius="4,4,4,4">
          </Border>
          <ContentPresenter HorizontalAlignment="Center"
                            VerticalAlignment="Center"
                            Width="Auto"
                            Grid.RowSpan="2" Padding="4"/>
          <Border HorizontalAlignment="Stretch" Margin="0,0,0,0"
                  x:Name="shine" Width="Auto"
                  CornerRadius="4,4,0,0">
            <Border.Background>
              <LinearGradientBrush EndPoint="0.5,0.9"
                                   StartPoint="0.5,0.03">
                <GradientStop Color="#99FFFFFF" Offset="0"/>
                <GradientStop Color="#33FFFFFF" Offset="1"/>
              </LinearGradientBrush>
            </Border.Background>
          </Border>
          </Grid>
        </Border>
      </Border>
  </ControlTemplate>

  <Style TargetType="Button">
    <Setter Property="Template"
```

```
            Value="{StaticResource DarkGlassButton}" />
        <Setter Property="Foreground" Value="White" />
      </Style>

    </Application.Resources>
```

2. Перезапустите приложение и проанализируйте результат. Обратите внимание на эффект «затененного стекла» кнопки.

Эффект «затененного стекла» не влияет на кнопки калькулятора во время выполнения, поскольку мы заменяем их стили кодом после загрузки страницы.

Привязка данных

При разработке графических пользовательских интерфейсов часто требуется привязывать свойство одного элемента управления к другому или к определенным данным.

Привязка к элементам

Самый простой тип привязки — привязка между элементами.

1. В файле `MainPage.xaml`, после второй горизонтальной панели и внутри внешней вертикальной панели, добавьте текстовый блок с инструкциями, ползунковый регулятор с диапазоном значений от 0 до 360 градусов, сетку, содержащую панель и текстовые блоки, радиальный датчик из UWP Community Toolkit и красный квадрат, который будет вращаться:

```
<TextBlock Margin="10">
  Use the slider to rotate the square:
</TextBlock>
<Slider Value="180" Minimum="0" Maximum="360"
        Name="sliderRotation" Margin="10,0" />
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <StackPanel Orientation="Horizontal"
              VerticalAlignment="Center"
              HorizontalAlignment="Center">
    <TextBlock FontSize="30"
      Text="{Binding ElementName=sliderRotation, Path=Value}" />
    <TextBlock Text="degrees" FontSize="30" Margin="10,0" />
  </StackPanel>
  <kit:RadialGauge Grid.Column="1" Minimum="0" Maximum="360"
```

```
      Value="{Binding ElementName=sliderRotation, Path=Value}"
      Height="200" Width="200" />
    <Rectangle Grid.Column="2" Height="100" Width="100" Fill="Red">
      <Rectangle.RenderTransform>
        <RotateTransform
          Angle="{Binding ElementName=sliderRotation, Path=Value}" />
      </Rectangle.RenderTransform>
    </Rectangle>
  </Grid>
```

2. Обратите внимание: текст блока, значение радиального датчика и угол поворота вращения привязаны к значению ползункового регулятора с помощью расширения разметки `{Binding}`, в котором указаны имя элемента и имя свойства привязки, также известное как путь.
3. Перезапустите приложение. Нажав и удерживая кнопку мыши, перетащите ползунковый регулятор, чтобы повернуть красный квадрат, как показано на рис. Б.11.

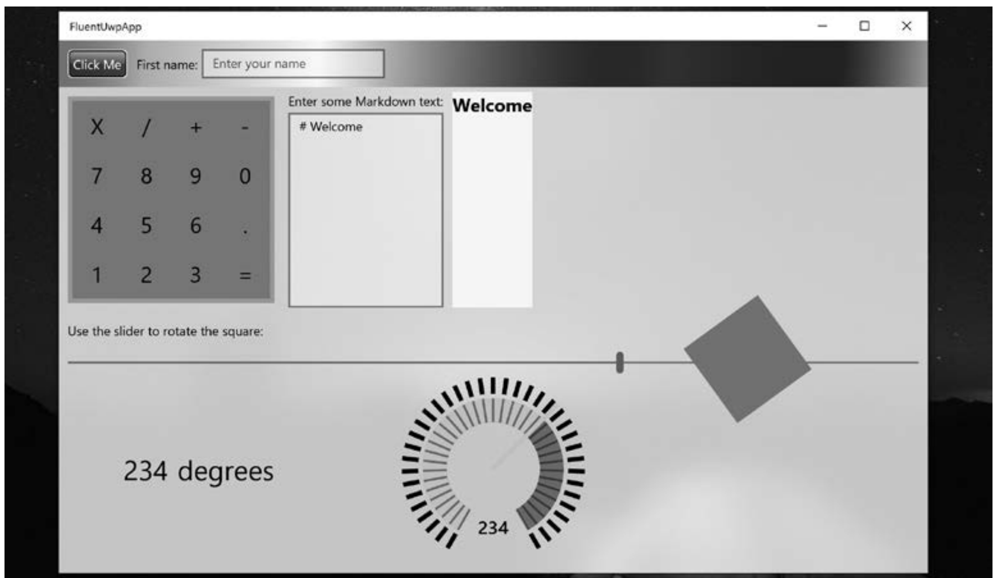


Рис. Б.11

Практические задания

Проверим полученные знания. Для этого ответьте на несколько вопросов, выполните приведенные упражнения и посетите указанные ресурсы, чтобы получить дополнительную информацию.

Упражнение Б.1. Проверочные вопросы

Ответьте на следующие вопросы.

1. Платформа .NET 5 — кросс-платформенная. Приложения Windows Forms и WPF могут работать на .NET 5. Могут ли эти приложения работать на операционных системах macOS и Linux?
2. Как приложение Windows Forms определяет свой пользовательский интерфейс и почему это сложная задача?
3. Как WPF- или UWP-приложение может определять свой пользовательский интерфейс и почему это хорошо для разработчиков?
4. Назовите пять контейнеров макета и варианты расположения их дочерних элементов.
5. В чем разница между `Margin` и `Padding` для такого элемента, как `Button`?
6. Как обработчики событий присоединяются к объекту с помощью XAML?
7. Как работают стили XAML?
8. Где вы можете определить ресурсы?
9. Каким образом вы можете связать свойство одного элемента со свойством другого?
10. Почему вам может потребоваться реализовать интерфейс `IValueConverter`?

Упражнение Б.2. Дополнительные ресурсы

Посетите следующие сайты, чтобы получить дополнительную информацию по темам, приведенным в этой главе:

- подготовка устройства к разработке: https://docs.microsoft.com/en-us/windows/uwp/get-started/enable-your-device-for-development;
- введение в работу с приложениями для операционной системы Windows 10: https://docs.microsoft.com/ru-ru/windows/uwp/get-started/;
- введение в работу с Windows Community Toolkit: https://docs.microsoft.com/ru-ru/windows/communitytoolkit/getting-started;
- проектирование и разработка Windows-приложений: https://docs.microsoft.com/ru-ru/windows/uwp/design/;
- разработка UWP-приложений: https://docs.microsoft.com/ru-ru/windows/uwp/develop/.

Резюме

В этом приложении вы узнали, что при создании настольных Windows-приложений платформа .NET 5 поддерживает более старые технологии, включая Windows Forms и WPF.

Вы узнали, как с помощью языка XAML и новой системы Fluent Design создавать графические пользовательские интерфейсы, включая такие функции, как Acrylic (акриловые материалы) и Reveal (подсвечивание).

Вы также узнали, как совместно использовать ресурсы в приложении, заменить шаблон элемента управления и как связать данные и элементы управления.