EXPERT INSIGHT

SPRINT book

.NET 8 приложения и сервисы

Практика создания проектов с использованием Blazor, .NET MAUI, gRPC, GraphQL и других технологий

Второе издание

Марк Дж. Прайс

<packt>

Apps and Services with .NET 8

Second Edition

Build practical projects with Blazor, .NET MAUI, gRPC, GraphQL, and other enterprise technologies

Mark J. Price

‹packt›

.NET 8: приложения и сервисы

Второе издание

Практика создания проектов с использованием
Blazor, .NET MAUI, gRPC, GraphQL и других технологий

Марк Дж. Прайс

SPRINT book 2025

Марк Дж. Прайс

.NET 8: приложения и сервисы. Практика создания проектов с использованием Blazor, .NET MAUI, gRPC, GraphQL

2-е издание

Перевел с английского А. Ларин

ББК 32.973.2-018
УДК 004.42

Прайс Марк Дж.

П68 .NET 8: приложения и сервисы. Практика создания проектов с использованием Blazor, .NET MAUI, gRPC, GraphQL. 2-е изд. — Астана: «Спринт Бук», 2025. — 768 с.: ил.

ISBN 978-601-12-3150-3

С новым изданием «.NET 8: приложения и сервисы» вы сможете поднять свои навыки владения C# и .NET на новый уровень. В книге описаны практические приемы использования различных технологий, включая Web API, gRPC, GraphQL и SignalR, что выделяет ее среди других источников. Она познакомит вас с новой унифицированной моделью Blazor Full Stack и расскажет, как использовать .NET MAUI для разработки мобильных и десктопных приложений. В новом издании представлены последние достижения, включая бесшовную реализацию веб-сервисов со встроенной поддержкой Ahead-of-Time (AOT) ADO.NET SqlClient. Помимо других популярных библиотек, в книге рассматриваются Humanizer и Noda Time. А совершенно новая глава расскажет об архитектуре, кэшировании, организации очередей и приемах построения надежных фоновых сервисов. Обретите глубокие знания и создайте многофункциональные и эффективные приложения.

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

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. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. 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.

ISBN 978-601-12-3150-3

© Packt Publishing 2023. First published in the English language under the title ‘Apps and Services with .NET 8 - Second Edition – (9781837637133)’
© Перевод на русский язык ТОО «Спринт Бук», 2025
© Издание на русском языке, оформление ТОО «Спринт Бук», 2025

Изготовлено в России. Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес: 010000, Казахстан, город Астана, район Алматы, проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 08.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 15.07.25. Формат 70×100/16. Бумага офсетная. Усл. п. л. 61,920. Тираж 500. Заказ 0000.

Краткое содержание

Об авторе 20
О научных редакторах 21
Предисловие 22
От издательства 27

Глава 1. Знакомство с приложениями и сервисами на .NET 28
Глава 2. Управление реляционными данными в системе SQL Server 66
Глава 3. Построение моделей сущностей для SQL Server с помощью EF Core 113
Глава 4. Управление данными NoSQL с помощью Azure Cosmos DB 157
Глава 5. Многозадачность и конкурентность 200
Глава 6. Популярные сторонние библиотеки 231
Глава 7. Дата, время и интернационализация 276
Глава 8. Создание и защита веб-сервисов на основе минимальных API 328
Глава 9. Кэширование, очереди и отказоустойчивые фоновые сервисы 387
Глава 10. Создание бессерверных наносервисов с помощью Azure Functions 449
Глава 11. Трансляция сообщений в реальном времени с помощью SignalR 488
Глава 12. Объединение источников данных с помощью GraphQL 514
Глава 13. Создание эффективных микросервисов с помощью gRPC 572
Глава 14. Создание пользовательских веб-интерфейсов с помощью ASP.NET Core 622
Глава 15. Создание веб-компонентов с помощью Blazor 664
Глава 16. Создание мобильных и классических приложений с помощью .NET MAUI 701

Послесловие 740
Приложение. Ответы к разделам «Проверьте свои знания» 744

Оглавление

Об авторе 20

О научных редакторах 21

Предисловие 22
- Примеры кода 22
- Структура книги 23
  - Введение 23
  - Данные 23
  - Библиотеки 23
  - Сервисы 24
  - Приложения 25
  - Резюме 25
- Что вам понадобится 25
- Полноцветные иллюстрации 25
- Условные обозначения 26

От издательства 27

Глава 1. Знакомство с приложениями и сервисами на .NET 28
- Знакомство с этой книгой и ее содержимым 28
  - Сопутствующие книги для продолжения обучения 29
  - Что вы узнаете из этой книги 29
  - Уникальная философия обучения 31
  - Исправление моих ошибок 31
  - Готовые решения на сайте GitHub 31
  - Соглашения об именовании проектов и нумерации портов 31
  - Настройка информирования об ошибках 33
- Технологии разработки приложений и сервисов 35
  - Технология .NET 35
  - Создание сайтов и приложений с помощью ASP.NET Core 35
  - Создание веб- и других сервисов 36
  - Windows Communication Foundation 37
  - Общие принципы сервисов 38

Советы по выбору сервиса ... 38
Создание Windows-приложений ... 40
Создание кросс-платформенных мобильных и классических приложений ... 42
Альтернативы .NET MAUI ... 42
Настройка среды разработки ... 43
Выбор подходящего редактора кода ... 44
Кросс-платформенное развертывание ... 46
Загрузка и установка Visual Studio 2022 ... 47
Загрузка и установка Visual Studio Code ... 48
Использование ресурсов Azure ... 52
Использование других шаблонов проектов ... 53
Установка дополнительных пакетов шаблонов ... 54
Обзор высокоуровневых программ, функций и пространств имен ... 55
Обработка локальной функции в процессе компиляции ... 56
Размещение статических функций в отдельном файле класса Program ... 56
Обработка статической функции в процессе компиляции ... 57
GitHub-репозиторий для этой книги ... 58
При возникновении вопросов по книге ... 59
Загрузка кода решения из репозитория GitHub ... 59
Источники справочной информации ... 60
Техническая документация на сайте Microsoft Learn ... 60
Получение справки с помощью команды dotnet ... 60
Инструменты ИИ, такие как ChatGPT и GitHub Copilot ... 62
Подписка на официальный блог .NET и уведомления ... 64
Практика и исследование ... 64
Упражнение 1.1. Проверьте свои знания ... 65
Упражнение 1.2. Дополнительная информация ... 65
Резюме ... 65

Глава 2. Управление реляционными данными в системе SQL Server ... 66
Создание современной БД ... 66
Использование образца реляционной БД ... 66
Подключение к БД SQL Server ... 68
Установка и настройка локальной версии SQL Server ... 69
Настройка Azure SQL Database ... 74
Управление данными с помощью Transact-SQL ... 80
Типы данных T-SQL ... 80
Документирование и комментирование кода ... 81
Объявление переменных ... 81
Указание типов данных ... 81
Управление потоком ... 81
Операторы ... 82
Язык манипулирования данными (DML) ... 82

DML для добавления, обновления и удаления данных ... 85
Язык описания данных (DDL) ... 85
Управление данными с помощью низкоуровневых API ... 86
Типы данных в ADO.NET ... 87
Создание консольного приложения для работы с ADO.NET ... 88
Выполнение запросов и извлечение данных с помощью ADO.NET ... 95
Вывод статистики ... 97
Асинхронная работа с ADO.NET ... 98
Выполнение хранимых процедур с помощью ADO.NET ... 99
Вывод потоков с помощью объекта DataReader ... 103
Создание объектов с помощью объекта DataReader ... 105
Управление данными с помощью Dapper ... 107
Расширение подключений с помощью Dapper ... 107
Выполнение запросов с помощью Dapper ... 108
Удаление использовавшихся ресурсов ... 110
Удаление ресурсов Azure ... 110
Практика и исследование ... 111
Упражнение 2.1. Проверьте свои знания ... 111
Упражнение 2.2. Дополнительная информация ... 111
Упражнение 2.3. Альтернативные варианты хранения конфиденциальной информации ... 111
Резюме ... 112

Глава 3. Построение моделей сущностей для SQL Server с помощью EF Core ... 113
Управление данными с помощью EF Core ... 113
Обзор Entity Framework Core ... 114
Жесткая привязка моделей на основе существующей БД ... 114
Настройка инструмента dotnet-ef ... 115
Конфигурирование моделей данных EF Core ... 116
Соглашения EF Core для конфигурации модели ... 116
Атрибуты аннотаций EF Core для конфигурации модели ... 117
Конфигурация модели EF Core с помощью Fluent API ... 118
Инициализация базы данных с помощью Fluent API ... 119
Конфигурирование модели БД Northwind ... 119
Запрос модели Northwind ... 125
Управление отслеживанием сущностей ... 128
Работа с иерархиями наследования в EF Core ... 132
Стратегия сопоставления «одна таблица — одна иерархия» (TPH). ... 132
Стратегия сопоставления «одна таблица — один тип» (TPT) ... 133
Стратегия сопоставления «одна таблица — конкретный тип» (TPC) ... 134
Настройка стратегий сопоставления для иерархий наследования ... 135
Пример стратегий сопоставления иерархий ... 136
Построение многократно используемой модели данных сущностей ... 144
Создание библиотеки классов для моделей сущностей в SQL Server ... 145

Создание библиотеки классов для контекста данных с помощью SQL Server ... 147
Свойства, вычисляемые при создании модели сущности ... 149
Тестовый проект для проверки интеграции библиотек классов ... 151
Модульные тесты для моделей сущностей ... 152
Запуск модульных тестов в Visual Studio 2022 ... 154
Запуск модульных тестов в Visual Studio Code ... 154
Практика и исследование ... 155
Упражнение 3.1. Проверьте свои знания ... 155
Упражнение 3.2. Практическое сравнение ADO.NET с EF Core ... 155
Упражнение 3.3. Обзор решений в отношении производительности ... 155
Упражнение 3.4. Дополнительная информация ... 155
Резюме ... 156

Глава 4. Управление данными NoSQL с помощью Azure Cosmos DB ... 157
Основы работы с БД NoSQL ... 157
Cosmos DB и поддерживаемые API ... 158
Моделирование документов ... 158
Уровни согласованности ... 160
Иерархия компонентов ... 162
Обеспечение пропускной способности ... 163
Стратегии разбиения ... 164
Проектирование хранилищ данных ... 165
Перенос данных в Cosmos DB ... 165
Создание ресурсов Cosmos DB ... 165
Эмуляция ресурсов Azure Cosmos DB в Windows ... 166
Создание ресурсов Azure Cosmos DB на портале Azure ... 171
Создание ресурсов Azure Cosmos DB с помощью приложения .NET ... 174
Управление данными через API Core (SQL) ... 179
Выполнение CRUD-операций с помощью API Cosmos SQL ... 179
Структура SQL-запросов ... 189
Полезные SQL-запросы в Cosmos DB ... 192
Серверное программирование в Cosmos DB ... 192
Реализация функций, определяемых пользователем ... 192
Реализация хранимых процедур ... 194
Очистка ресурсов Azure ... 197
Практика и исследование ... 198
Упражнение 4.1. Проверьте свои знания ... 198
Упражнение 4.2. Моделирование и разделение данных ... 198
Упражнение 4.3. Дополнительная информация ... 198
Упражнение 4.4. Шпаргалки по запросам Cosmos DB ... 198
Упражнение 4.5. Gremlin API ... 199
Упражнение 4.6. Дополнительные материалы по NoSQL и EF Core ... 199
Резюме ... 199

Глава 5. Многозадачность и конкурентность ... 200
Процессы, потоки и задачи ... 200
Асинхронное выполнение задач ... 201
Синхронное выполнение нескольких действий ... 202
Асинхронное выполнение нескольких действий с помощью задач ... 204
Начальные задания ... 205
Ожидание завершения задач ... 206
Использование методов wait в задачах ... 206
Выполнение задач продолжения ... 207
Вложенные и дочерние задачи ... 208
Обертывание объектов в задачи ... 210
Синхронизация доступа к общим ресурсам ... 212
Доступ к ресурсу из нескольких потоков ... 212
Применение взаимно исключающей блокировки к «раковине» ... 214
Оператор блокировки ... 215
Предотвращение взаимных блокировок ... 216
Синхронизация событий ... 217
Атомарность операций ЦП ... 218
Другие типы синхронизации ... 219
Конструкции async и await ... 220
Ускорение отклика консольных приложений ... 220
Управление асинхронными потоками ... 221
Ускорение отклика GUI-приложений ... 222
Оптимизация масштабируемости веб-приложений и веб-сервисов ... 228
Распространенные типы с поддержкой многозадачности ... 229
Оператор await в блоках catch ... 229
Практика и исследование ... 229
Упражнение 5.1. Проверьте свои знания ... 229
Упражнение 5.2. Дополнительная информация ... 230
Упражнение 5.3. Узнайте больше о параллельном программировании ... 230
Резюме ... 230

Глава 6. Популярные сторонние библиотеки ... 231
Наиболее популярные сторонние библиотеки ... 231
О чем рассказывается в моих книгах ... 232
Работа с изображениями ... 233
Генерация полутоновых изображений ... 233
Функционал ImageSharp для отрисовки и обработки изображений в Интернете ... 238
Обработка строк и чисел с помощью Humanizer ... 238
Работа с текстом ... 238
Преобразование регистра ... 238
Преобразование пробелов ... 239
Методы преобразования форм слов ... 240

Пример обработки текста в консольном приложении ... 240
Работа с числами ... 244
Работа с датами и временем ... 246
Журналирование с помощью Serilog ... 248
Структурированные данные о событиях ... 248
Приемники журналов Serilog ... 249
Журналирование в консоль и файл ... 249
Сопоставление объектов ... 251
Определение моделей для конфигурации AutoMapper ... 252
Определение сопоставлений для конфигурации AutoMapper ... 254
Тестирование конфигурации AutoMapper ... 255
Выполнение сопоставления в консольном приложении ... 257
Формулировка утверждений в модульном тестировании ... 259
Формулировка утверждений для строковых значений ... 259
Формулировка утверждений для коллекций и массивов ... 260
Формулировка утверждений для дат и времени ... 261
Проверка достоверности данных ... 262
Встроенные валидаторы ... 262
Настройка валидации ... 262
Уведомления валидации ... 263
Создание модели и валидатора ... 263
Тестирование валидатора ... 265
Проверка достоверности данных в ASP.NET Core ... 268
Создание PDF-файлов ... 269
QuestPDF на устройствах Mac с Apple Silicon ... 269
Создание библиотек классов для генерации PDF-документов ... 269
Создание консольного приложения для формирования PDF-документов ... 272
Практика и исследование ... 274
Упражнение 6.1. Проверьте свои знания ... 274
Упражнение 6.2. Дополнительная информация ... 275
Резюме ... 275

Глава 7. Дата, время и интернационализация ... 276
Управление датами и временем ... 276
Создание значений даты и времени ... 276
Форматирование значений даты и времени ... 280
Вычисления даты и времени ... 283
Микросекунды и наносекунды ... 284
Глобализация дат и времени ... 285
Сложности перехода на летнее время (DST) ... 286
Локализация перечисления DayOfWeek ... 287
Работа с датами и временем по отдельности ... 288
Получение информации о форматировании даты/времени ... 289
Модульное тестирование с учетом времени ... 290

Учет часовых поясов .NET 294
Классы DateTime и TimeZoneInfo 295
Пример работы с классами DateTime и TimeZoneInfo 296
Применение культур 301
Определение и изменение текущей культуры 302
Временное использование инвариантной культуры 308
Локализация пользовательского интерфейса 309
Определение и загрузка ресурсов 310
Тестирование глобализации и локализации 313
Библиотека Noda Time 319
Важные понятия и значения по умолчанию в Noda Time 320
Преобразование типов даты/времени Noda Time 322
Пример работы с Noda Time в консольном приложении 322
Модульное тестирование и сериализация JSON с помощью Noda Time 326
Практика и исследование 326
Упражнение 7.1. Проверьте свои знания 326
Упражнение 7.2. Дополнительная информация 326
Упражнение 7.3. Уроки эксперта Джона Скита 327
Резюме 327

Глава 8. Создание и защита веб-сервисов на основе минимальных API 328
Создание веб-сервисов на основе минимальных API в ASP.NET Core 328
Преимущества минимальных API 328
Маршрутизация в минимальных API 329
Настройка параметров в минимальных API 330
Возвращаемые значения 331
Документирование сервисов на основе минимальных API 331
Настройка проекта веб-API в ASP.NET Core 332
Тестирование веб-сервисов с помощью Swagger 338
Тестирование веб-сервисов в редакторе кода 341
Исключение путей из документации OpenAPI 345
Создание шаблонного кода для минимальных API в Visual Studio 2022 345
Управление междоменными запросами с помощью механизма CORS 346
Настройка HTTP-журналирования 347
Создание веб-страницы с клиентским JavaScript 348
Создание .NET-клиента 352
Механизм CORS 356
Включение CORS для определенных конечных точек 357
Другие политики CORS 358
Защита от DoS-атак путем ограничения количества запросов 358
Ограничение количества запросов с помощью пакета AspNetCoreRateLimit 359
Создание консольного клиента с ограничением скорости 363
Ограничение количества запросов в ASP.NET Core с помощью промежуточного ПО 368

Оптимизация производительности с помощью технологии Native AOT 368
Ограничения Native AOT 369
Рефлексия и Native AOT 370
Поддержка Native AOT в ASP.NET Core 370
Требования к Native AOT 371
Добавление Native AOT в проект 372
Включение сериализации JSON в Native AOT 372
Создание проекта Native AOT 372
Публикация проекта Native AOT 378
Работа с сервисами идентификации 381
Авторизация с помощью токена JWT Bearer 382
Проверка подлинности клиентов сервисов с помощью JWT Bearer 382
Практика и исследование 385
Упражнение 8.1. Проверьте свои знания 385
Упражнение 8.2. Обзор философии дизайна Microsoft HTTP API 386
Упражнение 8.3. Дополнительная информация 386
Упражнение 8.4. Шаблоны проекта Auth0 386
Резюме 386

Глава 9. Кэширование, очереди и отказоустойчивые фоновые сервисы 387
Архитектура сервисов 387
Самые медленные компоненты системы 387
Время доступа к данным 388
Кэширование в ASP.NET Core 389
Общие рекомендации по кэшированию 389
Создание сервиса веб-API на основе контроллера 389
Кэширование объектов в оперативной памяти 395
Распределенное кэширование объектов 398
Новая абстракция для распределенного кэширования 402
HTTP-кэширование веб-ответов 402
Повышение отказоустойчивости с помощью библиотеки Polly 406
Паттерны Retry и Circuit Breaker 407
Определение и применение политик 407
Настройка интервалов между попытками 408
Применение политик к HTTP-клиентам 409
Внесение в веб-сервис случайных ошибок 409
Создание проекта MVC для вызова отказавшего веб-сервиса 410
Реализация шаблона Retry для обработки временных сбоев 415
Создание очередей с помощью RabbitMQ 417
Настройка RabbitMQ с помощью Docker 419
Отправка сообщений в очередь с сайта MVC 421
Консольное приложение для обработки сообщений в очереди 425
Реализация фоновых сервисов с длительным временем работы 430
Создание Worker Service 430
Обработка сообщений в очереди с помощью Worker Service 433

Выполнение кода по расписанию 435
Создание сайта с функционалом Hangfire 438
Планирование задач с помощью Hangfire 443
Практика и исследование 445
Упражнение 9.1. Проверьте свои знания 446
Упражнение 9.2. Дополнительная информация 446
Упражнение 9.3. Замените модель Distributed Memory Cache другой реализацией распределенного кэша 446
Упражнение 9.4. Замените Hangfire на Quartz.NET 447
Упражнение 9.5. Знакомство с шаблоном Reliable Web App 447
Резюме 448

Глава 10. Создание бессерверных наносервисов с помощью Azure Functions 449
Платформа Azure Functions 449
Триггеры и привязки Azure Functions 450
Выражения NCRONTAB 451
Версии и языки Azure Functions 455
Модели рабочих процессов Azure Functions 456
Тарифные планы Azure Functions 457
Требования к хранилищу Azure Storage 458
Локальное тестирование с помощью Azurite 458
Уровни авторизации Azure Functions 459
Поддержка внедрения зависимостей в Azure Functions 459
Установка Azure Functions Core Tools 459
Создание проекта Azure Functions 460
В Visual Studio 2022 460
В Visual Studio Code 461
В терминале с помощью инструмента func 462
Обзор проекта Azure Functions 462
Реализация функции 464
Тестирование функции 466
Реакция на триггеры таймера и ресурсов 467
Реализация функции с триггером в форме таймера 468
Проверка функции 471
Реализация функции для работы с очередями и BLOB-объектами 474
Тестирование функции 480
Публикация проекта Azure Functions в облаке 482
Публикация в Visual Studio 2022 483
Публикация в Visual Studio Code 485
Удаление ресурсов Azure Functions 486
Практика и исследование 486
Упражнение 10.1. Проверьте свои знания 486
Упражнение 10.2. Дополнительная информация 487
Резюме 487

Глава 11. Трансляция сообщений в реальном времени с помощью SignalR 488
Знакомство с SignalR 488
Развитие связи в реальном времени 488
AJAX 489
WebSocket 489
SignalR 489
Сервис Azure SignalR 490
Разработка сигнатур методов 490
Создание сервиса живого общения в SignalR 491
Определение общих моделей 492
Включение центра SignalR на сервере 493
Разработка веб-клиента на JavaScript в SignalR 497
Добавление на сайт страницы чата 498
Тестирование функции чата 501
Создание консольного клиента на .NET 505
Создание .NET-клиента для SignalR 505
Тестирование консольного клиента .NET 506
Потоковая передача данных с помощью SignalR 507
Определение центра для потоковой передачи 508
Создание .NET-клиента для потоковой передачи 509
Проверка потокового сервиса и клиента 511
Практика и исследование 512
Упражнение 11.1. Проверьте свои знания 513
Упражнение 11.2. Дополнительная информация 513
Резюме 513

Глава 12. Объединение источников данных с помощью GraphQL 514
Основы GraphQL 514
Формат документа GraphQL-запроса 515
Другие возможности GraphQL 516
Платформа ChilliCream GraphQL 516
Создание сервиса с поддержкой GraphQL 517
Определение схемы GraphQL для проекта с приветствием 518
Составление и выполнение GraphQL-запросов 521
Именование GraphQL-запросов и операций 522
Оформление полей в GraphQL 522
Определение GraphQL-запросов для моделей EF Core 523
Добавление поддержки EF Core 523
GraphQL-запросы к БД Northwind 525
Реализация пагинации 532
Реализация фильтрации 535
Реализация сортировки 539
Создание .NET-клиентов для GraphQL-сервиса 539
Форматы GraphQL-запросов 540

Формат GraphQL-ответов 540
Тестирование GraphQL с помощью REST-клиента 540
GraphQL-клиент на основе ASP.NET Core MVC 543
Тестирование .NET-клиента 550
Создание консольного клиента с помощью Strawberry Shake 551
Реализация мутаций GraphQL 556
Добавление мутаций в GraphQL-сервис 556
Исследование мутации для добавления товара 558
Реализация мутаций для обновления и удаления 560
Реализация подписок в GraphQL 566
Добавление подписки и темы в GraphQL-сервис 566
Тестирование подписки на тему 568
Практика и исследование 570
Упражнение 12.1. Проверьте свои знания 570
Упражнение 12.2. Дополнительная информация 570
Упражнение 12.3. Создание .NET-клиентов 571
Резюме 571

Глава 13. Создание эффективных микросервисов с помощью gRPC 572
Обзор фреймворка gRPC 573
Принцип работы gRPC 573
Определение контрактов gRPC в .proto-файлах 573
Преимущества gRPC 576
Ограничения gRPC 576
Типы методов gRPC 576
gRPC-пакеты корпорации Microsoft 577
Создание сервиса и клиента gRPC 577
Создание gRPC-сервиса Hello World 577
Конфигурация файлов проекта 582
Создание gRPC-клиента Hello World 582
Тестирование сервиса и клиента gRPC 586
Реализация gRPC в модели EF Core 587
Создание gRPC-сервиса 587
Реализация gRPC-клиента 590
Улучшение gRPC-сервиса 594
Улучшение gRPC-сервиса с помощью публикации Native AOT 594
Работа с метаданными запроса и ответа 598
Ограничение срока выполнения в рамках повышения надежности 599
Обработка дат, времени и десятичных чисел 602
Определение пользовательского типа для десятичных чисел, обработка времени и дат 603
Реализация gRPC-сервисов для работы с товарами и сотрудниками 607
Создание gRPC-клиентов для вызова сервисов 607
Проверка обработки десятичных дробей, дат и байтов 610

Перехватчики gRPC и обработка ошибок 612
Добавление перехватчика на стороне клиента 612
Обработка исключений и временных сбоев 613
Транскодирование JSON в gRPC 616
Включение транскодирования JSON 616
Проверка транскодирования JSON 618
Сравнение с gRPC-Web 619
Практика и исследование 620
Упражнение 13.1. Проверьте свои знания 620
Упражнение 13.2. Сравнение сервисов gRPC с API HTTP 620
Упражнение 13.3. Дополнительная информация 620
Резюме 621

Глава 14. Создание пользовательских веб-интерфейсов с помощью ASP.NET Core 622
Настройка сайта MVC ASP.NET Core 622
Создание сайта MVC ASP.NET Core 623
Обзор стандартного сайта MVC ASP.NET Core 625
Регистрация посетителей 626
Структура проекта сайта MVC 626
Ссылка на библиотеку классов EF Core и регистрация контекста данных 629
Определение пользовательских веб-интерфейсов с помощью представлений Razor 629
Обзор представлений Razor 629
Создание прототипов с помощью Bootstrap 632
Синтаксис и выражения Razor 633
HTML-хелперы 634
Строго типизированное представление Razor 635
Локализация и глобализация в ASP.NET Core 638
Создание файлов ресурсов 638
Изменение представлений Razor с помощью внедряемого сервиса локализатора представлений 641
Заголовок Accept-Language 644
Применение тег-хелперов к HTML-элементам 645
Сравнение HTML-хелперов и тег-хелперов 645
Тег-хелпер привязки 646
Тег-хелпер кэша 651
Тег-хелпер среды 653
Кэширование и тег-хелперы 655
Тег-хелперы формы 656
Кэширование выходных данных 658
Конечные точки кэширования выходных данных 659
Кэширование выходных данных представления MVC 660

Практика и исследование ... 662
Упражнение 14.1. Проверьте свои знания ... 662
Упражнение 14.2. Создание пользовательских интерфейсов с помощью библиотеки Bootstrap ... 662
Упражнение 14.3. Дополнительная информация ... 663
Резюме ... 663

Глава 15. Создание веб-компонентов с помощью Blazor ... 664
Знакомство с Blazor ... 665
Модели размещения Blazor ... 665
Компоненты Blazor ... 666
Маршрутизация к страницам-компонентам ... 667
Передача параметров маршрута ... 669
Установка параметров из строки запроса ... 669
Ограничения маршрута ... 669
Базовые классы компонентов ... 670
Макеты Blazor ... 671
Настройка маршрутов Blazor к компонентам страницы ... 672
Изоляция CSS и JavaScript ... 672
Создание компонентов Blazor ... 673
Создание проекта Blazor из шаблона ... 673
Подключение значков Bootstrap ... 678
Создание ссылок на библиотеку классов EF Core и регистрация контекста данных ... 679
Статический Blazor-компонент с визуализацией данных на стороне сервера ... 680
Blazor-компонент с интерактивностью на стороне сервера ... 681
Blazor-компонент индикатора выполнения ... 683
Blazor-компонент диалогового окна ... 684
Blazor-компонент уведомлений ... 688
Создание компонента данных Blazor ... 690
Создание компонента ... 691
Превращение компонента в маршрутизируемую страницу ... 692
Получение данных для компонента через веб-сервис ... 693
Получение данных в компонент через вызов веб-сервиса ... 696
Практика и исследование ... 699
Упражнение 15.1. Проверьте свои знания ... 699
Упражнение 15.2. Создание компонентов Blazor ... 699
Упражнение 15.3. Создание сервиса для работы с IndexedDB ... 700
Упражнение 15.4. Дополнительная информация ... 700
Резюме ... 700

Глава 16. Создание мобильных и классических приложений с помощью .NET MAUI ... 701
Обзор XAML ... 702
Оптимизация кода с помощью XAML ... 702
Пространства имен .NET MAUI ... 704

Преобразователи типов ... 705
Выбор элементов управления .NET MAUI ... 705
Расширения разметки в XAML ... 706
Обзор .NET MAUI ... 707
Поддержка .NET MAUI и Xamarin ... 708
Средства разработки для мобильных и облачных проектов ... 708
Установка рабочих нагрузок .NET MAUI вручную ... 709
Категории компонентов пользовательского интерфейса .NET MAUI ... 711
Обработчики .NET MAUI ... 712
Разработка платформенно-специфичного кода ... 713
Разработка мобильных и классических приложений с помощью .NET MAUI ... 714
Виртуализация Android-устройства для локального тестирования приложений ... 714
Включение режима разработчика Windows ... 715
Создание проекта .NET MAUI ... 716
Добавление навигационной оболочки и страниц с дополнительным содержимым ... 719
Внедрение дополнительных страниц с содержимым ... 725
Управление общими ресурсами ... 727
Определение общих ресурсов в приложении ... 727
Обращение к общим ресурсам ... 728
Динамическое изменение общих ресурсов ... 729
Привязка данных ... 736
Привязка к элементам ... 736
Практика и исследование ... 738
Упражнение 16.1. Проверьте свои знания ... 738
Упражнение 16.2. Дополнительная информация ... 739
Резюме ... 739

Послесловие ... 740
Облачно-нативная разработка с .NET Aspire ... 740
Ознакомление с проектом Survey Project Challenge ... 741
Следующие шаги на пути изучения C# и .NET ... 741
Сопутствующие книги для продолжения обучения ... 742

Приложение. Ответы к разделам «Проверьте свои знания» ... 744

Об авторе

Марк Дж. Прайс — сертифицированный специалист Microsoft по программам Programming in C# и Architecting Microsoft Azure Solutions, имеет более чем 20-летний опыт работы. С 1993 года Марк сдал свыше 80 экзаменов Microsoft по программированию и специализируется на подготовке к сдаче экзаменов других людей. С 2001 по 2003 год Марк занимался составлением официальных учебных программ для Microsoft в Редмонде, США. С коллегами он выпустил первые учебные курсы по C# еще в пору альфа-тестирования. В Microsoft Марк проводил «тренинг тренеров», готовя сертифицированных преподавателей компании к работе с C# и .NET. Большую часть своей карьеры Марк обучает студентов разного возраста — от 16-летних подростков до 70-летних пенсионеров, в большинстве своем профессиональных разработчиков. Имеет степень бакалавра компьютерных наук.

Спасибо всем моим читателям. Благодаря вашей поддержке я пишу эти книги и радуюсь вашим успехам.

Отдельное спасибо читателям, оставляющим мне информативные отзывы в моем репозитории GitHub, по электронной почте, а также общающимся со мной в тематических сообществах в Discord. Вы помогаете улучшать мои книги с каждым изданием.

Отдельное спасибо Трою, читателю, коллеге и, что наиболее важно, хорошему другу.

О научных редакторах

Трой Мартин — программист-самоучка, пишущий код больше десяти лет, причем последнее время в основном на C#. Глубоко мотивированный и увлеченный, он получил более 20 сертификатов по различным языкам программирования и движкам для разработки игр. Сейчас он занят разработкой своего первого независимого игрового проекта и стремится помочь единомышленникам достичь их собственных целей в программировании.

Я хотел бы поблагодарить свою замечательную девушку, Хейли, поддерживающую меня в самые тяжелые времена; я люблю тебя, моя макарошка!

А еще выражаю глубочайшую благодарность Марку Дж. Прайсу, автору этой книги, замечательному другу и интересному собеседнику.

Киран Фут — программист-самоучка на C# с удивительным стремлением к изучению новых технологий и методов. Он постоянно отслеживает последние достижения в области C#, уделяя особое внимание веб-технологиям. Будучи ведущим разработчиком ПО в небольшой софтверной компании ConnX Business Solutions, расположенной в Великобритании, он помогает другим людям в приобретении новых навыков.

Предисловие

Есть книги по программированию объемом тысячи страниц, авторы которых стремились написать всеобъемлющие справочники по языку C#, библиотекам .NET и моделям сайтов, сервисов, классических (десктопных) и мобильных приложений.

Эта книга другая. У вас в руках пошаговое руководство по изучению различных технологий создания приложений и сервисов на .NET. Книга лаконична и представляет собой скорее быстрое, увлекательное чтиво с практическими разборами каждой темы. Из-за широты изложения в ряде случаев приходится расплачиваться краткостью, но я дополняю материал множеством ссылок для дальнейшего изучения.

По моему опыту, наисложнейшее в изучении новой технологии — это начать. Когда уяснены наиболее важные ключевые понятия и изучены практические примеры кода, появляются уверенность и желание углубиться в изучение дополнительной официальной документации. Разобравшись с основами, вы начнете уверенно экспериментировать самостоятельно.

Эта книга лучше всего подойдет знающим основы C# и .NET.

Я расскажу о наиболее важных особенностях моделей приложений и фреймворках для построения современных пользовательских интерфейсов и внедрения сервисов. Так вы сможете обсуждать с коллегами различия технологий и архитектурных решений и быстро приступать к их реализации.

Примеры кода

Вы можете скачать/клонировать решения пошаговых заданий и упражнений из репозитория GitHub по ссылке https://github.com/markjprice/apps-services-net8.

Структура книги

Введение

Глава 1 посвящена настройке среды разработки и концепциям использования приложений Visual Studio 2022, Visual Studio Code и JetBrains Rider. Вы также узнаете, как получить справочные сведения и связаться со мной (автором этой книги) в целях решения проблем или отправки отзыва для улучшения книги.

Данные

Глава 2 посвящена настройке SQL Server под Windows и в облаке Azure с помощью SQL Database. Затем вы создадите демонстрационную БД вымышленной организации Northwind. С помощью библиотек ADO.NET (`Microsoft.Data.SqlClient`) вы научитесь писать код низкоуровневого доступа для достижения максимальной производительности, а также упрощать разработку посредством *Dapper* — технологии преобразования объектов в данные.

Глава 3 рассказывает об использовании технологии преобразования объектов в данные более высокого уровня — *Entity Framework Core (EF Core)*. Вы научитесь создавать библиотеки классов, определяющие модель EF Core, для работы с БД Northwind, созданной в главе 2. Эти библиотеки классов используются во многих последующих главах.

Глава 4 посвящена нативному облачному NoSQL-хранилищу данных Azure Cosmos DB. Вы узнаете, как считывать и записывать данные с помощью нативного API.

Библиотеки

В главе 5 рассказывается о повышении производительности, масштабируемости и продуктивности работы пользователей посредством одновременного выполнения нескольких действий с помощью потоков и задач.

В главе 6 рассматриваются вопросы решения таких распространенных практических задач, как форматирование текста и чисел с помощью инструмента Humanizer, обработка изображений в ImageSharp, логирование с библиотекой Serilog, преобразование объектов с помощью AutoMapper, генерация модульных тестов в FluentAssertions, проверка данных посредством FluentValidation и создание PDF-документов в библиотеке QuestPDF.

В главе 7 рассказывается о способах решения таких популярных задач, как обработка дат и времени, работа с часовыми поясами, а также глобализация и локализация данных и пользовательского интерфейса приложения в рамках интернационализации. Помимо встроенных типов данных для работы с датами и временем, мы рассмотрим гораздо более совершенную стороннюю библиотеку Noda Time.

Сервисы

В главе 8 представлен наипростейший способ создания веб-сервисов на основе минимальных API в ASP.NET Core. Благодаря им отпадает необходимость в использовании классов-контроллеров. Вы узнаете, как сократить время запуска и количество требуемых ресурсов, используя инструментарий публикации Native AOT. Затем мы рассмотрим, как защитить и обезопасить веб-сервис, ограничивая количество запросов, а также используя технологии CORS, проверки подлинности и авторизации. Вы изучите способы тестирования веб-сервиса в новом HTTP-редакторе, встроенном в программу Visual Studio 2022, и с помощью расширения REST Client для Visual Studio Code.

В главе 9 речь пойдет о проектировании сервисов и добавлении в них средств повышения масштабируемости и надежности, таких как кэширование и очереди. Вы также научитесь справляться с временными сбоями и решать длительные задачи с помощью фоновых сервисов.

Глава 10 знакомит с сервисом Azure Functions, настраиваемым для работы исключительно за счет ресурсов сервера. При инициации некоего действия, скажем отправки сообщения в очередь, загрузки файла в хранилище или с указанной периодичностью запускается соответствующий наносервис.

В главе 11 рассматривается SignalR — технология для создания многопользовательских сервисов, позволяющих рассылать сообщения всем или определенным клиентам в реальном времени, например системам уведомлений и дашбордам с регулярно обновляемой информацией, в частности с котировками акций.

В главе 12 рассказывается о создании сервисов, обеспечивающих удобный единый канал агрегации данных из нескольких источников. Для реализации сервиса вы воспользуетесь платформой ChilliCream GraphQL, содержащей сервер Hot Chocolate. Новое в этом издании книги — реализация функций пагинации, фильтрации, сортировки и подписки.

Глава 13 посвящена сборке эффективных gRPC-микросервисов. Вы узнаете о `.proto`-файлах для конфигурации сервисных контрактов и бинарном формате Protobuf в рамках сериализации сообщений. Путем транскодирования gRPC JSON вы реализуете поддержку gRPC-сервисов в браузерах. В этом издании я описал приемы оптимизации запуска и уменьшения затрат памяти gRPC-сервиса с помощью инструментария публикации Native AOT, процесс работы с пользовательскими типами данных, в том числе десятичными значениями, а также способы реализации перехватчиков и обработчиков сбоев.

Приложения

Глава 14 посвящена созданию пользовательских веб-интерфейсов с помощью фреймворка MVC на платформе ASP.NET Core. Вы узнаете о синтаксисе Razor, тег-хелперах и фреймворке Bootstrap для быстрого прототипирования пользовательских интерфейсов.

В главе 15 вы займетесь разработкой компонентов пользовательского интерфейса с помощью нового унифицированного полностекового хостинга Blazor, представленного в .NET 8. Теперь допускается персональная настройка компонентов Blazor для выполнения приложения на сторонах клиента и сервера одновременно. На случай, когда необходимо использовать такие объекты браузера, как локальное хранилище, вы узнаете о JS-взаимодействии.

Глава 16 рассматривает процессы разработки кросс-платформенных мобильных и классических приложений под системы Android, iOS, macOS и Windows. Вы познакомитесь с языком XAML, используемым для конфигурации графического пользовательского интерфейса приложения.

Резюме

В конце книги описываются способы получения дополнительной информации о разработке приложений и сервисов на C# и .NET, а также инструменты и навыки, необходимые каждому профессиональному разработчику .NET.

В приложении приведены ответы на вопросы из разделов «Проверьте свои знания» в конце каждой главы.

Что вам понадобится

Вы можете разрабатывать и развертывать приложения и сервисы на C# и .NET с помощью программы Visual Studio 2022 или Visual Studio Code и консольных инструментов в большинстве операционных систем, в том числе Windows, macOS и многие версии Linux. Операционная система, поддерживающая Visual Studio Code, и подключение к Интернету — это все, что вам необходимо для изучения данной книги. Разумеется, вы можете использовать и другие инструменты, такие как JetBrains Rider.

Полноцветные иллюстрации

По ссылке https://packt.link/gbp/9781837637133 вы можете скачать PDF-файл с полноцветными снимками экрана и диаграммами, приведенными в книге. На них лучше видны изменения в коде и выводе.

Условные обозначения

Текст в книге размечен разными стилями. Ниже приведены примеры форматирования и объяснение их предназначения.

`Моноширинным` шрифтом в тексте выделены фрагменты кода в тексте, названия таблиц базы данных, каталогов, файлов, а также расширения файлов, пути к ним, данные, введенные пользователем. Например, так: «папки `Controllers`, `Models` и `Views` содержат классы ASP.NET Core и файлы `.cshtml`, которые обрабатываются на стороне сервера».

Листинги оформлены следующим образом:

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

Когда вам стоит обратить внимание на определенный фрагмент листинга, соответствующие строки или элементы выделены полужирным:

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

Ввод/вывод командной строки/терминала оформлен следующим образом:

```
dotnet new console
```

Курсивным начертанием выделяется новый термин или важное слово.

Шрифтом без засечек выделены слова, которые вы видите на экране, в частности названия элементов меню или диалоговых окон, упомянутые в тексте. Например: «при нажатии кнопки Next (Далее) вы переходите к следующему экрану». Кроме того, шрифтом без засечек выделены действующие URL-адреса и адреса электронной почты, а также названия клавиш.

Так оформлены важные примечания и ссылки на сторонние источники информации для дальнейшего чтения.

Дельный совет: так оформлены экспертные рекомендации по программированию.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство SprintBook, компьютерная редакция).

Мы будем рады узнать ваше мнение!

1 Знакомство с приложениями и сервисами на .NET

В первой главе мы настроим среду разработки, установив программы Visual Studio 2022 и Visual Studio Code, а также разберемся с необходимыми инструментами для разработки приложений и сервисов. В конце главы вы узнаете, куда обращаться за помощью.

В репозитории GitHub, который я создал для этой книги, размещены полные решения для всех описываемых задач. Он доступен по ссылке https://github.com/markjprice/apps-services-net8/.

Решать задачи из книги можно как в браузерной версии Visual Studio Code, так и в обычном редакторе кода, а также использовать эти инструменты одновременно. Удобно сравнивать свой код с моим решением и при необходимости копировать/вставлять нужные части.

Упоминая в этой книге слово *«современная»* в контексте платформы .NET, я имею в виду версию 8 и ее предшественниц, таких как .NET 6, произошедших из .NET Core. Слово *«устаревшая»* относится к платформам .NET Framework, Mono, Xamarin и .NET Standard. Современная версия .NET консолидирует перечисленные унаследованные платформы и стандарты.

Знакомство с этой книгой и ее содержимым

Эта книга рассчитана на следующие аудитории:

- читателей моей книги *C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals*, желающих продолжить обучение;
- разработчиков с базовыми навыками и знаниями языка C# и платформы .NET, желающих получить опыт разработки реальных приложений и сервисов.

Сопутствующие книги для продолжения обучения

Эта книга — вторая из трилогии, посвященной платформе .NET 8.

1. В первой книге даются основы языка C# и библиотеки .NET и ASP.NET Core для разработки веб-приложений. Она предназначена для последовательного чтения, поскольку навыки и знания, полученные в предыдущих главах, дополняются и необходимы для понимания материала последующих глав.
2. Во второй книге (вы держите ее в руках) рассматриваются более узкие темы, такие как интернационализация, а также популярные пакеты сторонних разработчиков, в том числе Serilog и NodaTime. Вы научитесь разрабатывать нативные сервисы с AOT-компиляцией на основе минимальных API ASP.NET Core и узнаете, как повысить производительность, масштабируемость и надежность посредством кэширования данных, очередей и фоновых сервисов. С технологиями GraphQL, gRPC, SignalR и Azure Functions вы реализуете дополнительные сервисы. Наконец, вы узнаете, как с помощью Blazor и .NET MAUI создавать графические пользовательские интерфейсы для сайтов, классических и мобильных приложений.
3. В третьей книге[1] описываются важные инструменты и навыки, которыми должен обладать каждый профессиональный разработчик .NET. Сюда входят паттерны проектирования и архитектуры решений, отладка, анализ памяти, все важные виды тестирования — модульное, производительности, системное, веб- и мобильное, а также инструменты хостинга и развертывания, такие как Docker и Azure Pipelines. Наконец, мы рассмотрим, как подготовиться к собеседованию на замещение вакансии разработчика .NET.

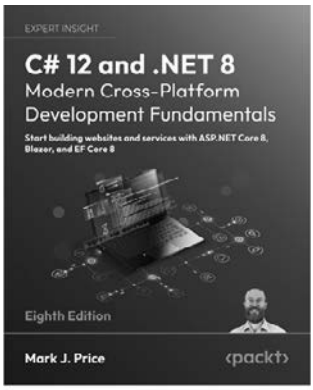
Краткое описание трилогии .NET 8 и ее наиболее важных тем показано на рис. 1.1.

Что вы узнаете из этой книги

За исключением первой главы, книгу можно разделить на четыре характерные части.

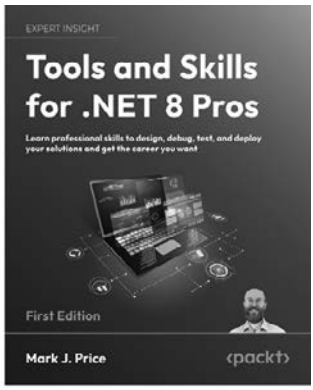
1. **Управление данными:** хранение и управление локальными и облачными данными с помощью SQL Server и Azure Cosmos DB. В последующих главах используются БД SQL Server и модели сущностей, созданием которых вы займетесь в конце главы 3. В главе, посвященной Cosmos DB, используется библиотека SQL API, а глава о Gremlin, интерфейсе, обеспечивающем способ конфигурации, выполнения и взаимодействия с обходами графа, доступна в качестве дополнения в Интернете.

[1] *Прайс М.* .NET 8: лучшие практики и паттерны проектирования. — СПб.: Питер, 2025.



- **Язык C#**, в том числе новые возможности версии C# 12, объектно-ориентированное программирование (ООП), отладка и модульное тестирование.
- **Библиотеки .NET**, в том числе касающиеся работы с числами, текстом, регулярными выражениями, коллекциями, файловым вводом-выводом и данными с помощью EF Core и SQLite.
- **Разработка сайтов и веб-сервисов** с помощью ASP.NET Core и Blazor

- **Дополнительные библиотеки:** интернационализация, многозадачность и пакеты сторонних разработчиков.
- **Дополнительная информация об обработке данных:** SQL Server и Cosmos DB.
- **Дополнительная информация о сервисах:** кэширование, очереди, минимальные API, GraphQL, gRPC, SignalR и Azure Functions.
- **Дополнительная информация о пользовательских интерфейсах**: ASP.NET Core и .NET MAUI



- **Инструменты:** интегрированные среды разработки, отладка, анализ памяти и ИИ-ассистенты.
- **Тестирование:** модульное, интеграционное, производительности, системное и веб-, в том числе тесты внедрения зависимостей и инверсий контроля.
- **Проектирование:** паттерны и архитектуры.
- **Развертывание:** непрерывная интеграция/непрерывное развертывание, хостинг в Azure, конвейеры развертывания и автоматизация.
- **Карьера:** подготовка к собеседованию

Рис. 1.1. Трилогия для изучения C# 12 и .NET 8

2. **Специализированные библиотеки:** обработка дат и времени, интернационализация, повышение производительности с помощью потоков и путем делегирования задач, сторонние библиотеки для работы с изображениями, правила проверки данных и т. д. Эти главы сродни книге кулинарных рецептов. Если вас не интересует какая-то тема, вы можете ее пропустить, читать эти главы можно в любом порядке.
3. **Сервисные технологии:** создание и защита сервисов, построенных на минимальных программных интерфейсах ASP.NET Core, GraphQL, gRPC, SignalR и Azure Functions. Вы узнаете об очередях, кэшировании и планировании событий в рамках повышения масштабируемости и надежности сервисов.

   Технологии пользовательских интерфейсов: создание пользовательских интерфейсов средствами ASP.NET Core, Blazor и .NET MAUI.

Уникальная философия обучения

Большинство людей лучше всего усваивают сложные темы путем подражания и повторения, а не читая подробное объяснение теории. Поэтому я не буду перегружать вас доскональными объяснениями каждого шага. Суть в том, чтобы мотивировать вас писать код и анализировать результаты его выполнения.

С ходу вам не понадобятся все тонкости и нюансы. Это дело времени, когда вы будете разрабатывать собственные приложения и выходить за рамки того, о чем прочли в книгах.

Исправление моих ошибок

Говоря словами Сэмюэла Джонсона, автора толкового словаря английского языка 1755 года, я совершил *«ряд дичайших промахов и забавных нелепостей, от которых не застрахована ни одна работа подобного масштаба»*. Я беру на себя всю ответственность за них и надеюсь, что вы оцените мою попытку идти против ветра, написав эту книгу о стремительно развивающихся технологиях, таких как C# и .NET, и приложениях и сервисах, создаваемых с их помощью.

Если у вас возникли какие-либо затруднения с этой книгой, пожалуйста, свяжитесь со мной, прежде чем писать отрицательный отзыв на сайте книжного магазина. Авторы не могут отвечать на такие отзывы, поэтому я не могу связаться с вами с целью решить проблему. Я хочу, чтобы вы извлекли максимальную пользу от моей книги, поэтому прислушиваюсь к вашим отзывам и учитываю их в следующем издании. Пожалуйста, напишите мне (адрес моей электронной почты опубликован в репозитории GitHub к этой книге), пообщайтесь со мной в специальном канале Discord (https://discord.com/invite/ETZTPmrQFX) или поднимите вопрос на форуме по адресу https://github.com/markjprice/apps-services-net8/issues.

Готовые решения на сайте GitHub

Универсальный код решений в репозитории GitHub для этой книги доступен по адресу https://github.com/markjprice/apps-services-net8/tree/main/code.

Соглашения об именовании проектов и нумерации портов

Выполняя задания по программированию, описанные в книге, вы создадите десятки проектов. Многие из них будут представлять собой сайты и сервисы, требующие номера портов для размещения на домене `localhost`.

В крупных и сложных решениях бывает трудно ориентироваться в кодовой базе. Поэтому для структурирования проектов весьма желательно упрощать поиск

компонентов. Рекомендуется использовать всеобъемлющие названия решений, передающие суть приложения/проекта.

В 1990-х годах компания Microsoft зарегистрировала торговый знак *Northwind* в качестве названия вымышленной компании для использования в БД и примерах кода. Сначала оно применялось в примерах для программы Access, а затем стало использоваться и в SQL Server. Мы создадим несколько проектов для этой вымышленной компании, поэтому будем использовать префикс Northwind в названиях всех проектов.

Дельный совет: есть множество способов структурирования и присвоения названий проектам и решениям, например выстраивание иерархии папок, а также соглашения об именовании. Если вы работаете в команде, согласуйте эти правила.

Чтобы любой разработчик мог с ходу определить предназначение проектов в решении, полезно учитывать соглашение об именовании проектов. Обычно упоминается тип проекта, например библиотека классов, консольное приложение, сайт и т. д. (табл. 1.1).

Таблица 1.1. Примеры названий для распространенных типов проектов

Название	Описание
`Northwind.Common`	Проект библиотеки классов для типов общего назначения, таких как интерфейсы, перечисления, классы, записи и структуры, используемых в ряде проектов
`Northwind.Common.EntityModels`	Проект библиотеки классов для моделей сущностей EF Core общего назначения. Модели сущностей часто используются как на стороне сервера, так и на стороне клиента, поэтому лучше всего отделить зависимости от конкретных провайдеров баз данных
`Northwind.Common.DataContext`	Проект библиотеки классов в контексте базы данных EF Core с зависимостями от конкретных провайдеров баз данных
`Northwind.Mvc`	Проект ASP.NET Core для сложного сайта на основе паттерна MVC. Проще проводить модульное тестирование
`Northwind.WebApi.Service`	Проект ASP.NET Core для сервиса HTTP API. Хороший выбор для интеграции с сайтами, поскольку для взаимодействия с сервисом может использоваться любая библиотека JavaScript или Blazor

Название	Описание
Northwind.WebApi.Client.Console	Клиент для веб-сервиса. Последняя часть названия указывает на то, что это консольное приложение
Northwind.gRPC.Service	Проект ASP.NET Core для gRPC-сервиса
Northwind.gRPC.Client.Mvc	Клиент для gRPC-сервиса. Последняя часть названия указывает на то, что это проект сайта ASP.NET Core MVC
Northwind.BlazorWasm.Client	Проект клиентской части ASP.NET Core Blazor WebAssembly
Northwind.BlazorWasm.Server	Проект серверной части ASP.NET Core Blazor WebAssembly
Northwind.BlazorWasm.Shared	Библиотека классов, совместно используемая клиентскими и серверными проектами Blazor

Чтобы запускать любой из этих проектов одновременно, важно не задавать одинаковые номера портов для разных проектов. Я использовал следующее соглашение:

```
https://localhost:5[номер_главы]1/
http://localhost:5[номер_главы]2/
```

Например, для зашифрованного подключения к сайту, созданному в главе 14, я использовал порт 5141:

```
https://localhost:5141/
```

Настройка информирования об ошибках

По умолчанию предупреждения компилятора возникают в случае потенциальных проблем с кодом при первой сборке проекта, но не препятствуют компиляции и скрыты при повторной сборке. Предупреждения возникают не на пустом месте, так что игнорировать их в любом случае не стоит.

Добросовестные разработчики своевременно реагируют на предупреждения, решая возникающие проблемы, для чего настраивают .NET-проекты следующим образом:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>
```

Я задействовал строгую обработку предупреждений (как ошибок) в (почти) всех решениях в репозитории GitHub.

Исключение касается gRPC-проектов ввиду сочетания ряда факторов. В .NET 7 и более поздних версиях компилятор выбрасывает предупреждение в процессе компиляции исходных файлов с типами, набранными строчными буквами. Такая ситуация возникает, если, к примеру, вы определите класс `person`:

```
public class person
{
}
```

Данное предупреждение компилятора было введено для предотвращения конфликтов добавляемых имен в будущих версиях C#.

Увы, инструменты Google для генерации исходных файлов C# из `.proto`-файлов создают имена классов, содержащие только строчные буквы:

```
#region Designer generated code

using pb = global::Google.Protobuf;
```

При строгой обработке предупреждений (они обрабатываются как ошибки) компилятор откажется компилировать исходный код:

```
Error CS8981 The type name 'pb' only contains lower-cased ascii characters.
Such names may become reserved for the language. Northwind.Grpc.Service C:\
apps-services-net8\Chapter14\Northwind.Grpc.Service\obj\Debug\net8.0\Protos\
Greet.cs
```

Дельный совет: в проектах .NET (за исключением gRPC, пока Google не обновит инструменты генерации кода) всегда задействуйте строгую обработку предупреждений.

Если после активации этой функции будет возникать слишком много ошибок, вы можете отключить определенные предупреждения, используя элемент `<WarningsNotAsErrors>` со списком кодов предупреждений, разделенных запятыми:

```
<WarningsNotAsErrors>0219,CS8981</WarningsNotAsErrors>
```

В копилку знаний: подробнее о настройке информирования об ошибках можно узнать по ссылке https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/compiler-options/errors-warnings#warningsaserrors-and-warningsnotaserrors.

Технологии разработки приложений и сервисов

В компании Microsoft платформы для создания приложений и сервисов называются *моделями приложений* (app models) или *рабочими нагрузками* (workloads).

Технология .NET

.NET, .NET Core, .NET Framework и Xamarin — это родственные и частично дублирующиеся платформы для разработки приложений и сервисов. Если вы не знакомы с историей .NET, рекомендую изучить концепции развития .NET по ссылке https://github.com/markjprice/cs12dotnet8/blob/main/docs/ch01-dotnet-history.md.

Создание сайтов и приложений с помощью ASP.NET Core

Сайты состоят из множества веб-страниц, загружаемых статически из файловой системы или генерируемых динамически с помощью серверной технологии типа ASP.NET Core. Браузер отправляет `GET`-запросы с *URL*-адресами (*Unique Resource Locator* — универсальный локатор ресурса) — уникальными идентификаторами каждой страницы — и управляет данными на сервере с помощью запросов `POST`, `PUT` и `DELETE`.

На многих сайтах браузер рассматривается как уровень представления, а почти вся обработка выполняется на стороне сервера. На стороне клиента могут выполняться JavaScript-сценарии для реализации некоторых функций представления, таких как веб-карусели, и для проверки данных.

Платформа ASP.NET Core предоставляет множество технологий для создания сайтов.

- *ASP.NET Core Razor Pages* позволяет динамически генерировать HTML-код простых сайтов.
- *ASP.NET Core MVC* — это реализация паттерна проектирования *MVC* (*Model — View — Controller*, «Модель — представление — контроллер»), востребованного при разработке сложных сайтов. О том, как с его помощью создавать пользовательские интерфейсы (user interface, UI), вы узнаете из главы 14.
- *Библиотеки классов Razor* позволяют упаковывать многократно используемый функционал проектов ASP.NET Core, в том числе компоненты пользовательского интерфейса.

- *Платформа Blazor* позволяет создавать UI-компоненты с помощью C# и .NET, а затем запускать их в браузере или встроенном веб-компоненте вместо JavaScript-фреймворков пользовательского интерфейса, таких как Angular, React и Vue.

Платформа Blazor предназначена не только для создания сайтов. В сочетании с .NET MAUI она также может использоваться для создания гибридных мобильных и классических приложений.

Предполагается, что вы уже знакомы с основами разработки на ASP.NET Core, поэтому, хотя в книге и рассматриваются основы, мы быстро перейдем к более продвинутым темам.

Создание веб- и других сервисов

Официальных определений не существует, но иногда сервисы описываются в зависимости от их сложности.

- *Сервис* — вся функциональность, необходимая для работы клиентского приложения, реализована в одном монолитном сервисе.
- *Микросервис* — небольшой автономный компонент, который предоставляет определенный набор функциональных возможностей. Руководящий принцип при определении границ функциональности микросервиса заключается в том, что каждый из них должен взаимодействовать с собственными данными. Только он должен читать и записывать эти данные. Если в ваше хранилище данных обращается несколько сервисов, то их нельзя назвать микросервисами.
- *Наносервис* — отдельная функция, предоставляемая в качестве услуги. В отличие от непрерывно работающих сервисов и микросервисов наносервисы часто неактивны, пока к ним не обратятся. Такая схема работы применяется для экономии ресурсов и затрат. По этой причине их также называют бессерверными сервисами.

Хотя в последнее десятилетие благодаря теоретическим преимуществам использовать микро- и наносервисы и стало модно, монолитные сервисы вновь набирают популярность, поскольку разработчики обнаружили, что в реальности микросервисы далеко не всегда так привлекательны, как в теории.

Вы научитесь создавать веб- и минимальные API ASP.NET Core — веб-сервисы с протоколом HTTP для базовых коммуникаций и принципами проектирования REST-архитектур, разработанными Роем Филдингом. Вы также узнаете, как созда-

вать сервисы с использованием веб- и прочих технологий, расширяющих базовые веб-API, в том числе следующих.

- *gRPC* — разработка высокоэффективных и производительных микросервисов с поддержкой практически любых платформ.
- *SignalR* — реализация между компонентами коммуникаций в реальном времени.
- *GraphQL* — управление со стороны клиента извлечением данных из нескольких источников. Хотя GraphQL и поддерживает протокол HTTP, это опциональная функция, и данный язык запросов не следует принципам веб-дизайна, описанным Роем Филдингом в своей диссертации о REST API.
- *Azure Functions* — развертывание бессерверных наносервисов в облаке.
- *OData* — удобное превращение EF Core и других моделей данных в веб-сервис.

Windows Communication Foundation

В 2006 году компания Microsoft выпустила версию .NET Framework 3.0 с несколькими новыми крупными фреймворками, в числе которых *Windows Communication Foundation (WCF)*. В нем бизнес-логика сервиса абстрагировалась от инфраструктуры коммуникационных технологий, чтобы в будущем можно было легко перейти на альтернативный вариант или даже использовать несколько механизмов для взаимодействия с сервисом.

Платформа WCF в значительной степени использует XML-конфигурацию для декларативного определения конечных точек, в том числе их адрес (address), привязку (binding) и контракт (contract). Это так называемая концепция ABC, трех ключевых элементов, конечных точек WCF. Освоив ее, вы оцените мощь и одновременную гибкость технологии WCF.

Компания Microsoft не стала официально портировать WCF на современную платформу .NET, тем не менее существует открытый проект *CoreWCF*, созданный сообществом разработчиков и управляемый организацией .NET Foundation. Если вам необходимо перенести сервис из .NET Framework на современную версию .NET или создать клиент для WCF-сервиса, воспользуйтесь проектом CoreWCF. Имейте в виду, что это решение никогда не будет считаться полноценным портом, поскольку некоторые компоненты технологии WCF специфичны для Windows.

Такие платформы, как WCF, позволяют создавать распределенные приложения. Клиентское приложение способно выполнять *удаленные вызовы процедур* (remote procedure calls, RPC) к серверному приложению. Вместо порта WCF следует

воспользоваться альтернативной технологией RPC, например gRPC, речь о которой и пойдет в этой книге.

В копилку знаний: дополнительная информация о CoreWCF приведена в репозитории GitHub по адресу https://github.com/CoreWCF/CoreWCF. А по ссылке https://devblogs.microsoft.com/dotnet/wcf-client-60-has-been-released/ вы сможете прочитать анонс о внедрении клиентской функциональности для вызова сервисов WCF/CoreWCF с помощью System.ServiceModel 6.0.

Общие принципы сервисов

Один из важнейших принципов архитектуры сервисов — оптимизация вызовов методов. Другими словами, старайтесь скомпоновать все данные, необходимые для выполнения операции, в одном вызове, а не в нескольких. Накладные расходы из-за удаленных вызовов — один из наибольших недостатков сервисов. Именно поэтому дробление на отдельные сервисы может негативно сказаться на архитектуре решения.

Советы по выбору сервиса

У каждой технологии сервисов есть свои преимущества и недостатки, касающиеся функционала (табл. 1.2).

Таблица 1.2. Преимущества и недостатки распространенных технологий сервисов

Функционал	Веб-API	OData	GraphQL	gRPC	SignalR
Клиенты могут запрашивать только необходимые данные	Нет	Да	Да	Нет	Нет
Минимальная версия HTTP	1.1	1.1	1.1	2.0	1.1
Поддержка браузеров	Да	Да	Да	Нет	Да
Формат данных	XML, JSON	XML, JSON	GraphQL (через JSON)	Бинарный	Различные
Сервисная документация	Swagger	Swagger	Нет	Нет	Нет
Генерация кода	Сторонняя	Сторонняя	Сторонняя	Google	Microsoft
Кэширование	Легко	Легко	Сложно	Сложно	Сложно

В табл. 1.3 приведены рекомендации для различных ситуаций.

Таблица 1.3. Сценарии использования сервисов и рекомендуемые технологии реализации

Ситуация	Рекомендация
Общедоступный сервис	В качестве публичных лучше всего подойдут сервисы на основе HTTP/1.1, особенно если к ним обращаются из браузера или с мобильного устройства
Общедоступный сервис данных	OData и GraphQL — прекрасные варианты для обработки сложных иерархических наборов данных, поступающих из различных хранилищ. Стандарт OData разработан и реализуется Microsoft в виде официальных пакетов .NET. Язык GraphQL разработан компанией Facebook и выпускается в виде сторонних пакетов
Межсервисное взаимодействие	Технология gRPC предназначена для обмена данными с низкой задержкой и высокой пропускной способностью. gRPC отлично подходит для легких внутренних микросервисов, когда решающее значение имеет эффективность
Прямая связь в реальном времени	Технология gRPC отлично поддерживает двунаправленную потоковую передачу. gRPC-сервисы могут передавать сообщения в реальном времени без опроса. Библиотека SignalR предназначена для различных видов коммуникаций в реальном времени, поэтому она, как правило, проще в реализации, чем gRPC, хотя и менее эффективна
Широковещательная связь в реальном времени	Технология SignalR поддерживает широковещательную трансляцию данных в реальном времени для многих клиентов
Многоязычная среда	Инструментарий gRPC поддерживает все популярные языки разработки, поэтому отлично подойдет для мультиязычных и платформенных сред
Среда с ограниченной пропускной способностью сети	Сообщения gRPC сериализуются в легком формате Protobuf, поэтому gRPC-сообщения компактнее эквивалентов в формате JSON
Бессерверный наносервис	Сервисы Azure Functions не нуждаются в круглосуточном хостинге, поэтому прекрасно подходят для наносервисов, обычно не нуждающихся в постоянной работе. В качестве альтернативы можно назвать сервис Amazon Web Services (AWS) Lambda

Создание Windows-приложений

Технологии для создания Windows-приложений, в основном для персональных компьютеров, включают в себя:

- *Windows Forms*, 2002 год;
- *Windows Presentation Foundation (WPF)*, 2006 год;
- *Windows Store*, 2012 год;
- *Universal Windows Platform (UWP)*, 2015 год;
- *Windows App SDK* (ранее *WinUI 3* и *Project Reunion*), 2021 год.

Устаревшие платформы Windows-приложений

После выхода версии Microsoft Windows 1.0 в 1985 году приложения под эту операционную систему можно было писать только на языке C, вызывая функции из трех базовых DLL-библиотек — `KERNEL`, `USER` и `GDI`. С выходом первой 32-битной версии операционной системы, Windows 95, в имена DLL-библиотек добавили суффикс 32 и обозначили их как *Win32 API*.

В 1991 году компанией Microsoft был выпущен язык Visual Basic, предоставивший разработчикам наглядный способ создания пользовательских интерфейсов Windows-приложений путем перетаскивания элементов с панели инструментов. Он пользовался огромной популярностью, а среда выполнения Visual Basic и сегодня поставляется в составе Windows 11.

С первой версией C# и .NET Framework, выпущенной в 2002 году, Microsoft представила технологию разработки классических Windows-приложений *Windows Forms*. В то время аналог для веб-проектов назывался *Web Forms*, отсюда и пошли взаимодополняющие названия. Поддерживался код на языках Visual Basic или C#. На платформе Windows Forms был реализован аналогичный визуальный конструктор с перетаскиванием элементов, а для функционирования пользовательского интерфейса генерировался код на языке C# или Visual Basic, который сложно понять и редактировать.

В 2006 году Microsoft выпустила более мощную платформу для разработки классических Windows-приложений — *Windows Presentation Foundation (WPF)*. Это был ключевой компонент технологии .NET Framework 3.0 наряду *с WCF* и *Windows Workfiow (WF)*.

Хотя WPF-приложения программируются исключительно на языке C#, для настройки пользовательского интерфейса поддерживается язык *eXtensible Application Markup Language (XAML)*, легко понятный как человеку, так и машине. Программа Visual Studio 2022 частично поддерживает WPF.

В 2012 году компания Microsoft представила релиз Windows 8 с компонентом Windows Store, запускаемым в защищенной песочнице.

В 2015 году компания Microsoft выпустила версию Windows 10 с обновленной концепцией приложений для Windows Store, получившей название *Universal Windows Platform (UWP)*. UWP-приложения создаются с помощью C++ и DirectX UI, JavaScript и HTML или C# на отдельном ответвлении современной платформы .NET, не кросс-платформенном, но предоставляющем полный доступ к базовым API WinRT.

UWP-приложения поддерживаются только операционными системами Windows 10 и 11, а также консолями Xbox и гарнитурами Windows Mixed Reality с контроллерами движения.

Многие разработчики Windows отказались от приложений Windows Store и UWP из-за ограниченного доступа к базовой системе.

Недавно компания Microsoft представила совместимые компоненты *Project Reunion* и *WinUI 3*. Благодаря им создателям WPF-программ стали доступны некоторые современные возможности разработки под Windows, а также преимущества и системные интеграции UWP-проектов. Эта инициатива известна как набор инструментов *Windows App SDK*.

В копилку знаний: в этой книге набор инструментов Windows App SDK не рассматривается, поскольку он не кросс-платформенный. Прочитать о нем вы можете на сайте https://learn.microsoft.com/ru-ru/windows/apps/windows-app-sdk/.

Поддержка устаревших платформ Windows-приложений в современной версии .NET

Набор инструментов .NET SDK для Linux и macOS занимает около 330 Мбайт. Размер .NET SDK для Windows составляет около 440 Мбайт, так как содержит набор компонентов *.NET Desktop Runtime*. С его помощью в современной версии .NET реализована поддержка устаревших платформ Windows Forms и WPF.

Выпущено множество корпоративных приложений Windows Forms и WPF, которые разработчики продолжают поддерживать и обновлять. До недавнего времени они работали на устаревшей платформе .NET Framework. С современной платформой .NET и набором компонентов .NET Desktop Runtime этим приложениям доступны все возможности современной версии .NET. Разработчики классических Windows-приложений также могут установить пакет совместимости Windows. Прочитать о нем можно по ссылке https://learn.microsoft.com/ru-ru/dotnet/core/porting/windows-compat-pack.

Создание кросс-платформенных мобильных и классических приложений

Сейчас в мире распространены две основные мобильные платформы, Apple iOS и Google Android, поддерживающие собственные языки программирования и API. Существуют также две основные операционные системы для компьютеров — Apple macOS и Microsoft Windows, тоже поддерживающие разные языки программирования и API:

- *iOS*: Objective-C и Swift, UIkit;
- *Android*: Java и Kotlin, Android API;
- *macOS*: Objective-C и Swift, AppKit и Catalyst;
- *Windows*: C, C++ и многие другие языки, Win32 API и Windows App SDK.

На платформе *.NET Multi-platform App UI (MAUI)* можно создавать кросс-платформенные мобильные и классические приложения, поддерживающие работу на многих мобильных и классических платформах.

Технология .NET MAUI упрощает разработку таких приложений за счет общих компонентов пользовательского интерфейса и логики. Применяются те же API .NET, что и при создании консольных приложений, сайтов и сервисов.

Приложения могут функционировать независимо, но чаще они задействуют специальные службы, обеспечивающие поддержку всех используемых устройств, от серверов и ноутбуков до смартфонов и консолей.

Платформа .NET MAUI поддерживает паттерны MVVM и XAML. Кроме того, в будущем разработчики платформы планируют добавить поддержку паттерна *Model-View-Update (MVU)* на C# в духе Swift UI компании Apple. MVU с фреймворком Comet пока находится на этапе проверки концепции. Он не так развит и не так хорошо поддерживается, как Swift UI. Я не буду рассказывать о нем в этой книге. Подробнее о нем вы можете прочитать по адресу https://github.com/dotnet/Comet.

Альтернативы .NET MAUI

Еще до выпуска фреймворка .NET MAUI компанией Microsoft свет увидели сторонние открытые XAML-решения, *Uno* и *Avalonia*. С их помощью .NET-разработчики смогли создавать кросс-платформенные приложения.

Платформа Uno

Uno — «первая открытая и бесплатная платформа с поддержкой C# и XAML для создания по-настоящему единых мультиплатформенных приложений». Так указано на сайте проекта по адресу https://platform.uno/.

Разработчики могут многократно использовать 99 % кода бизнес-логики и UI-слоя в нативных мобильных, веб- и классических программах.

Платформа Uno использует нативный стек Xamarin (но не Xamarin.Forms). Для создания WebAssembly-проектов в Uno предназначен инструмент mono-wasm в духе Blazor WebAssembly. Для разработки Linux-программ на платформе Uno используется библиотека Skia для рисования пользовательского интерфейса на холсте.

Могу порекомендовать замечательную книгу о платформе Uno — https://www.packtpub.com/en-us/product/creating-cross-platform-c-applications-with-uno-platform-9781801078498.

Платформа Avalonia

Avalonia — «кросс-платформенный UI-фреймворк для .NET, позволяющий создавать приложения в технике Pixel Perfect», который «поддерживается на всех основных платформах». Avalonia — «надежный UI-фреймворк для разработки сложных приложений», как утверждается на главной странице официального сайта по адресу https://avaloniaui.net/.

Проект Avalonia можно рассматривать как духовного наследника WPF. Разработчики WPF-, Silverlight- и UWP-приложений смогут и дальше извлекать пользу из своих многолетних знаний и опыта.

В лице проекта Avalonia компания JetBrains модернизировала устаревшие WPF-инструменты и превратила их в кросс-платформенные, поэтому редактор C#-кода поддерживается в Windows, macOS и Linux.

Благодаря расширению Avalonia для программы Visual Studio 2022 и глубокой интеграции с JetBrains Rider разработка приложений упрощается и ускоряется.

А теперь теорию о технологиях приложений и сервисов на .NET 8 отставим в сторону и перейдем к практике. Давайте разбираться, как настроить среду разработки.

Настройка среды разработки

Чтобы приступить к программированию, вам понадобится редактор C#-кода. Компания Microsoft выпускает семейство редакторов кода и *интегрированных сред разработки (IDE)*, к которым относятся:

- Visual Studio 2022 для Windows;
- Visual Studio Code для Windows, macOS и Linux;
- веб-версия Visual Studio Code и GitHub Codespaces.

Сторонние разработчики выпускают собственные редакторы кода C#, например платное приложение JetBrains Rider для Windows, macOS и Linux. Оно популярно среди более опытных разработчиков .NET.

Внимание! Хотя JetBrains — фантастическая компания, выпускающая отличное ПО, и Rider, и расширению ReSharper для Visual Studio не чужды ошибки и причудливое поведение. К примеру, при работе с ним во фреймворках Razor Pages, Razor Views и Blazor Components могут возникать ошибки в духе «Невозможно определить символ». Тем не менее вы можете вполне успешно запускать такие файлы, так как на самом деле проблемы нет. А плагин Unity Support для Rider может вызывать сообщения об ошибках при преобразовании структур в объекты, приводя в бешенство разработчиков игр на Unity. В книге мы не будем программировать игры под Unity, так что эта проблема нас интересовать не должна.

В главах 1–15 для создания приложений и сервисов будет достаточно среды Visual Studio 2022 или кросс-платформенных приложений Visual Studio Code и JetBrains Rider. Упражнения из главы 16 вы можете выполнить и в Visual Studio Code, хотя это будет не так-то и просто. Рекомендую использовать Visual Studio 2022, в которой поддержка .NET MAUI реализована гораздо лучше, чем в Visual Studio Code (по крайней мере, на данный момент).

Выбор подходящего редактора кода

Какое ПО лучше всего подходит для создания приложений и сервисов на C# и .NET?

Для выполнения упражнений из этой книги вы можете выбрать любой редактор или IDE с поддержкой программирования на языке C#, в том числе Visual Studio Code, Visual Studio 2022 или даже JetBrains Rider.

В книге приведены общие инструкции, выполнимые в любом редакторе, поэтому выбирайте тот, который вам больше нравится.

Универсальная разработка в Visual Studio 2022

Программа Visual Studio 2022 позволяет создавать большинство типов проектов, в том числе консольные, классические и мобильные приложения, сайты и веб-сервисы.

Хотя в Visual Studio 2022 можно создать кросс-платформенное мобильное приложение .NET MAUI, для его компиляции вам понадобится операционная система macOS и среда Xcode.

Программа Visual Studio 2022 поддерживает только 64-битные версии Windows.

Поддерживается система Windows 10 версии 1909 или более поздней либо Windows Server 2016 или более поздняя. Полная поддержка архитектуры Arm64 была внедрена в версии Visual Studio 2022 17.4.

Внимание! Программа Visual Studio 2022 для Mac официально не поддерживает .NET 8, ее поддержка прекратилась в августе 2024 года. Если вы пользовались версией Visual Studio 2022 для Mac, вам следует перейти на Visual Studio Code для Mac, JetBrains Rider для Mac или запускать Visual Studio 2022 для Windows на виртуальной машине на локальном компьютере или в облаке с помощью технологии Microsoft Dev Box. Объявление о прекращении поддержки можно прочитать по адресу https://devblogs.microsoft.com/visualstudio/visual-studio-for-mac-retirement-announcement/.

Кросс-платформенная разработка в Visual Studio Code

Самый современный и легкий редактор кода, а также единственный кросс-платформенный редактор компании Microsoft — это Visual Studio Code. Он поддерживается во всех распространенных операционных системах, в том числе Windows, macOS и многих разновидностях Linux, включая дистрибутивы *Red Hat Enterprise Linux (RHEL)* и Ubuntu.

Visual Studio Code — оптимальный выбор для современной кросс-платформенной разработки, так как в нем доступен обширный и пополняемый набор расширений для поддержки многих языков, помимо C#.

Этот кросс-платформенный и легкий редактор может быть установлен на всех целевых платформах ваших приложений для быстрого исправления ошибок и т. д. В отличие от мощной поддержки веб-проектов функционал для разработки классических и мобильных приложений в Visual Studio Code пока реализован слабо.

Программа Visual Studio Code поддерживается на процессорах ARM, поэтому разработка приложений может осуществляться на компьютерах Apple Silicon и Raspberry Pi.

Visual Studio Code — самая популярная интегрированная среда разработки: по результатам опроса на сайте Stack Overflow, проведенного в 2023 году (https://survey.stackoverflow.co/2023/), свыше 73 % профессиональных разработчиков выбрали именно ее.

Облачная разработка в GitHub Codespaces

GitHub Codespaces — это полностью настраиваемая среда разработки на базе Visual Studio Code, запускаемая на облачном сервере и доступная в любом браузере. В ней поддерживаются Git-репозитории, расширения и внедрен интерфейс командной строки. Созданные проекты можно редактировать, запускать и тестировать на любом устройстве.

В копилку знаний: подробнее о платформе GitHub Codespaces можно узнать по адресу https://github.com/features/codespaces.

Мое рабочее окружение

При написании и тестировании кода для этой книги я использовал следующее аппаратное и программное обеспечение:

- Visual Studio 2022 для Windows:
  - в операционной системе Windows 11 на ноутбуке HP Spectre (Intel);
- Visual Studio Code:
  - в операционной системе macOS на стационарном компьютере Apple Silicon Mac mini (M1);
  - в операционной системе Windows 11 на ноутбуке HP Spectre (Intel);
- JetBrains Rider:
  - в операционной системе Windows 11 на ноутбуке HP Spectre (Intel);
  - в операционной системе macOS на стационарном компьютере Apple Silicon Mac mini (M1).

Благодаря использованию различных аппаратных и программных средств можно понаблюдать за различиями в работе проектов на разных платформах и разобраться с проблемами, возникающими в процессе разработки. С другой стороны, любой из вышеперечисленных комбинаций достаточно для создания описываемых в книге приложений и веб-сервисов.

Проблема при использовании среды JetBrains Rider

Плагин Unity Support в JetBrains Rider часто сбоит при преобразовании структур в объекты. Обычно проблема возникает при передаче таких типов значений, как `int` и `DateTime`, в виде строковых позиционных аргументов. Ошибка актуальна только для проектов Unity, поскольку в этом случае используется другой сборщик мусора, отличный от обычной среды выполнения .NET. Для всех остальных проектов, в том числе и описанных в этой книге, данная ошибка не имеет значения. Подробнее о ней можно прочитать по адресу https://docs.unity3d.com/Manual/performance-optimizing-code-managed-memory.html#boxing.

Кросс-платформенное развертывание

Выбор редактора кода и операционной системы для разработки не ограничивает целевые платформы для развертывания ваших приложений.

.NET 8 поддерживает следующие платформы для развертывания:

- *Windows:* Windows 10 версии 1607 или более поздней; Windows 11 версии 21H2 или более поздней; Windows Server 2012 R2 SP1 или более поздняя версия; Nano Server версии 1809 или более поздней;
- *Mac:* macOS Catalina версии 10.15 или более поздней, а также в эмуляторе Rosetta 2 x64; Mac Catalyst 11.0 или более поздняя версия;

- *Linux:* Alpine Linux 3.17 или более поздняя версия; Debian 11 или более поздняя версия; Fedora 37 или более поздняя версия; openSUSE 15 или более поздняя версия; Oracle Linux 8 или более поздняя версия; *Red Hat Enterprise Linux* (*RHEL*) 8 или более поздняя версия; SUSE Enterprise Linux 12 SP2 или более поздняя версия; Ubuntu 20.04 или более поздняя версия;
- *Android:* API 21 или более поздняя версия;
- *iOS* и *tvOS:* 11.0 или более поздняя версия.

Внимание! Поддержка .NET для Windows 7 и 8.1 прекратилась в январе 2023 года: https://github.com/dotnet/core/issues/7556.

Поддержка Windows ARM64 в .NET 5 и более поздних версиях означает, что вы можете разворачивать и разрабатывать приложения для устройств Windows Arm, таких как Microsoft Windows Dev Kit 2023 (ранее известный как Project Volterra) и Surface Pro X.

Ознакомиться с актуальными версиями поддерживаемых операционных систем можно по адресу https://github.com/dotnet/core/blob/main/release-notes/8.0/supported-os.md.

Загрузка и установка Visual Studio 2022

Многие профессиональные разработчики Microsoft работают над своими проектами в Visual Studio 2022. Хотя для решения задач из этой книги вы могли выбрать программу Visual Studio Code, я рекомендую ознакомиться и с Visual Studio 2022.

Если у вас нет компьютера под управлением операционной системы Windows, то этот раздел можно пропустить и перейти к инструкциям по загрузке и установке Visual Studio Code в системе macOS или Linux.

С октября 2014 года компания Microsoft предоставляет студентам, разработчикам программ с открытым исходным кодом и частным лицам бесплатную профессиональную версию Visual Studio в редакции Community Edition. Для этой книги подойдет любая редакция, в том числе и Community. Если вы еще не установили Visual Studio, выполните следующие шаги.

1. Загрузите дистрибутив Visual Studio 2022 версии 17.8 или более поздней по адресу https://visualstudio.microsoft.com/ru/downloads/.
2. Запустите программу установки.
3. На вкладке Workloads (Рабочие нагрузки) выберите следующие варианты:
   - ASP.NET and web development (ASP.NET и разработка веб-приложений);
   - Azure development (Разработка для Azure);

- .NET Multi-platform App UI development (Разработка с помощью .NET Multi-platform App UI);
- .NET desktop development (Разработка классических приложений .NET) (данная нагрузка добавляет и разработку консольных приложений);
- Desktop development with C++ (Разработка классических приложений на C++) со всеми компонентами по умолчанию (чтобы публиковать оптимизированные по скорости запуска и затратам памяти консольные приложения и веб-сервисы).

4. На вкладке Individual components (Отдельные компоненты) в разделе Code tools (Средства для работы с кодом) установите флажок Git for Windows (Git для Windows).
5. Нажмите кнопку Install (Установить) и дождитесь, пока программа установки загрузит выбранное ПО и установит его.
6. Когда установка будет завершена, нажмите кнопку Launch (Запустить).
7. При первом запуске Visual Studio вам будет предложено авторизоваться в системе. Если у вас есть учетная запись Microsoft, укажите свой логин и пароль. Если нет, то зарегистрируйте новую учетную запись по адресу https://signup.live.com/.
8. При первом запуске Visual Studio вам будет предложено настроить среду. В разделе Development Settings (Параметры разработки) выберите пункт Visual C#. В качестве цветовой темы я выбрал синий цвет, но вы можете выбрать любой другой.
9. Если вы хотите настроить сочетания клавиш, перейдите в меню Tools ▸ Options (Средства ▸ Параметры), а затем выберите пункт Environment ▸ Keyboard (Окружение ▸ Клавиатура).

Сочетания клавиш Visual Studio 2022

В книге я обычно не указываю сочетания клавиш, поскольку они часто настраиваются индивидуально. Приведу только там, где они одинаковы в разных редакторах кода и часто используются.

Если вы хотите добавить и настроить сочетания клавиш, прочитайте инструкцию по адресу https://learn.microsoft.com/ru-ru/visualstudio/ide/identifying-and-customizing-keyboard-shortcuts-in-visual-studio.

Загрузка и установка Visual Studio Code

Программа Visual Studio Code заметно улучшилась за последние пару лет и приятно удивила сотрудников Microsoft своей популярностью. Если вы отчаянный человек и любите брать от жизни все, обратите внимание на выпуск Insiders — ежедневную сборку готовящейся к релизу версии.

Даже если для разработки вы планируете использовать только среду Visual Studio 2022, я рекомендую вам скачать и установить программу Visual Studio Code и попробовать выполнить в ней код из этой главы с его помощью. И затем уже определиться, в остальной части книги продолжать работу в Visual Studio Code или использовать Visual Studio 2022.

Итак, давайте загрузим и установим Visual Studio Code, .NET SDK и расширение C# Dev Kit.

1. Загрузите и установите сборку Stable или Insiders программы Visual Studio Code по адресу https://code.visualstudio.com/.

В копилку знаний: если вам требуется дополнительная помощь в установке Visual Studio Code в вашей операционной системе, обратите внимание на официальное руководство по адресу https://code.visualstudio.com/docs/setup/setup-overview.

2. По адресу https://dotnet.microsoft.com/ru-ru/download скачайте и установите пакет .NET SDK для версии .NET 8.0.
3. Чтобы установить расширение C# Dev Kit в графическом интерфейсе, необходимо сначала запустить приложение Visual Studio Code.
4. В Visual Studio Code нажмите кнопку Extensions (Расширения) или выберите команду меню View ▸ Extensions (Вид ▸ Расширения).
5. C# Dev Kit — одно из самых популярных расширений, поэтому оно появится в верхней части списка. В противном случае введите запрос C# Dev Kit в поле поиска.

C# Dev Kit зависит от расширения C# версии 2.0 или более поздней, поэтому вам не потребуется устанавливать его отдельно. Обратите внимание, что в расширении C#, начиная с версии 2.0, вместо OmniSharp используется новый протокол Language Service Protocol (LSP). Кроме того, C# Dev Kit зависит от расширений .NET Install Tool for Extension Authors и IntelliCode for C# Dev Kit, поэтому они также будут автоматически установлены.

6. Нажмите кнопку Install (Установить) и дождитесь загрузки/установки пакетов расширений.

Дельный совет: обязательно прочитайте лицензионное соглашение к расширению C# Dev Kit. В нем перечислены более строгие правила, чем в соглашении к расширению C#: https://aka.ms/vs/csdevkit/.

Установка расширений

В книге вы будете использовать несколько разных расширений Visual Studio Code. Чтобы вы могли установить их прямо сейчас, я перечислил их в табл. 1.4.

Таблица 1.4. Расширения Visual Studio Code, используемые в этой книге

Название и идентификатор расширения	**Описание**
C# Dev Kit `ms-dotnettools.csdevkit`	Официальное расширение компании Microsoft, добавляющее поддержку инструментов C#-разработчиков. Управляйте проектами и решениями на панели обозревателя решений и проверяйте их с помощью интегрированной среды модульного тестирования. Зависит от расширений C# и IntelliCode for C# Dev Kit
C# `ms-dotnettools.csharp`	Добавляет поддержку C#-кода с дополнительным функционалом, в том числе с подсветкой синтаксиса, IntelliSense, переходом к определениям, поиском ссылок, отладкой для .NET и поддержкой `csproj`-файлов в операционной системе Windows, macOS и Linux
IntelliCode for C# Dev Kit `ms-dotnettools.vscodeintellicode-csharp`	Реализует ИИ-функции разработки для разработчиков на Python, TypeScript/JavaScript, C# и Java
MSBuild project tools `tintoy.msbuild-project-tools`	Добавляет поддержку технологии IntelliSense в файлах проектов MSBuild, в том числе автодополнение элементов `<PackageReference>`
SQL Server (mssql) for Visual Studio Code `ms-mssql.mssql`	Расширение с богатым набором функциональных возможностей для разработки проектов SQL Server, Azure SQL Database и SQL-хранилищ данных
REST Client `humao.rest-client`	Позволяет отправлять и просматривать результаты HTTP-запросов прямо в Visual Studio Code
ilspy-vscode `icsharpcode.ilspy-vscode`	Расширение для декомпиляции сборок MSIL, поддерживающее современные версии .NET, .NET Framework, .NET Core и .NET Standard
vscode-proto3 `zxh404.vscode-proto3`	Добавляет подсветку и проверку синтаксиса, сворачивание фрагментов сценария, функции автодополнения и форматирования кода, проверку парности скобок и инструмент блочного и строчного комментирования
Azure Functions for Visual Studio Code `ms-azuretools.vscode-azurefunctions`	Предоставляет возможности по созданию, отладке, управлению и развертыванию бессерверных приложений прямо из программы Visual Studio Code. Зависит от расширений Azure Account (`ms-vscode.azure-account`) и Azure Resources (`ms-azuretools.vscode-azureresourcegroups`)

Управление расширениями в терминале

Вы можете установить расширение Visual Studio Code с помощью терминала (табл. 1.5).

Таблица 1.5. Управление расширениями в терминале

Команда	Описание
`code --list-extensions`	Перечисление установленных расширений
`code --install-extension <идентификатор>`	Установка расширения с указанным *идентификатором*
`code --uninstall-extension <идентификатор>`	Удаление расширения с указанным *идентификатором*

Так, чтобы установить расширение C# Dev Kit, введите в терминале следующий код:

```
code --install-extension ms-dotnettools.csdevkit
```

Я подготовил PowerShell-скрипты для установки и удаления перечисленных в таблице расширений Visual Studio Code. Вы найдете их по адресу https://github.com/markjprice/apps-services-net8/tree/main/scripts/extension-scripts.

Версии программы Microsoft Visual Studio Code

Компания Microsoft выпускает новую функциональную версию Visual Studio Code (практически) каждый месяц, а версии с исправлениями ошибок — еще чаще. Например:

- версия 1.78.0, релиз в апреле 2023 года;
- версия 1.78.1, перевыпуск апрельской версии с исправленными ошибками.

Версия, используемая в этой книге, — 1.83.0, выпущена в сентябре 2023 года. Обратите внимание, что версия программы Microsoft Visual Studio Code не так важна, как актуальность установленного расширения C#.

Хотя без расширения C# можно обойтись, оно внедряет удобные IntelliSense-инструменты автодополнения кода по мере ввода, функции навигации по коду и устранения ошибок. В силу этого я рекомендую его установить и обновлять для поддержки актуальных возможностей языка C#.

Сочетания клавиш Visual Studio Code

В книге я не указываю сочетания клавиш для выполнения таких операций, как создание файла, поскольку в разных операционных системах они часто отличаются. Я буду приводить многократно нажимаемые сочетания клавиш, например,

во время отладки. С высокой долей вероятности они одинаковы в разных операционных системах.

Если вы хотите настроить сочетания клавиш для Visual Studio Code, см. инструктаж по адресу https://code.visualstudio.com/docs/getstarted/keybindings.

Рекомендую загрузить PDF-файл с перечнем сочетаний клавиш для вашей операционной системы по следующим ссылкам:

- 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.

Использование ресурсов Azure

В некоторых главах для выполнения упражнений требуются учетная запись и ресурсы Azure. В рамках локальной разработки описываемых сервисов обычно хватает бесплатного уровня доступа, но некоторые ресурсы создаются и функционируют только после платной подписки.

К примеру, для проверки книг издательства Packt привлекаются научные редакторы, выполняющие все описываемые упражнения, словно читатели. Один из редакторов второго издания этой книги сообщил мне, что «получил на платформе Azure счет в 3,01 доллара за запуск проектов второго издания на “платной” основе».

В настоящее время на сайте Microsoft указано: *«С бесплатной учетной записью Azure новые соответствующие пользователи получат деньги на счет в Azure на сумму 200 долларов США, которые можно потратить в течение первых 30 дней, и 12 месяцев использования бесплатных сервисов в ограниченном количестве»*. Более подробную информацию вы найдете по адресу https://learn.microsoft.com/ru-ru/azure/cost-management-billing/manage/avoid-charges-free-account.

Дельный совет: удаляйте ненужные ресурсы Azure, чтобы сократить расходы.

Сводная информация о том, в каких главах вам потребуются ресурсы Azure и имеется ли бесплатная альтернатива в рамках локальной разработки, приведена в табл. 1.6.

Проверить использование бесплатных сервисов Azure можно по инструкции, приведенной по адресу https://learn.microsoft.com/ru-ru/azure/cost-management-billing/manage/check-free-service-usage.

Таблица 1.6. Главы, в примерах которых используются ресурсы Azure, и альтернативные варианты при локальной разработке

Главы	Ресурс Azure	Бесплатный уровень	Альтернатива при локальной разработке
2–16	БД SQL	Бесплатно на протяжении первого года	ПО SQL Server Developer Edition на Windows или SQL Edge в контейнере Docker на Windows, Linux и macOS
4	Cosmos DB	Пропускная способность 1000 ЕЗ/с и 25 Гбайт памяти	Эмулятор Azure Cosmos DB для Windows или ознакомительная версия для Linux
11	Azure Functions	1 млн запросов и 400 000 Гбайт трафика в месяц	Эмулятор Azurite с открытым исходным кодом для тестирования BLOB-объектов Azure, очередей Azure и приложений, использующих табличные хранилища, такие как Azure Functions
12	Azure SignalR	20 одновременных подключений и 20 000 сообщений в день с уровнем доступности в 99,9 %	Добавление в проект ASP.NET Core для локальной разработки функционала SignalR

Использование других шаблонов проектов

В комплекте с .NET SDK доступно множество шаблонов проектов. Давайте рассмотрим их.

1. В командной строке/терминале (далее в книге используется термин «терминал») введите следующую команду:

```
dotnet new list
```

ПО SDK для .NET 7 и более поздних версий поддерживают две команды, dotnet new --list и dotnet new list. Версия SDK для .NET 6 и более ранняя — только команду dotnet new --list.

2. Вы увидите список установленных шаблонов, в том числе предназначенных для разработки классических приложений Windows (при выполнении команды в Windows). Наиболее распространенные шаблоны приведены в табл. 1.7.

Таблица 1.7. Полные и краткие имена шаблонов проектов

Название шаблона	Краткое название	Язык
Приложение .NET MAUI	`maui`	C#
Приложение .NET MAUI Blazor	`maui-blazor`	C#
Пустой шаблон ASP.NET Core	`web`	C#, F#
Сервис gRPC ASP.NET Core	`grpc`	C#
Веб-API ASP.NET Core	`webapi`	C#, F#
Веб-API ASP.NET Core (Native AOT)	`webapiaot`	C#
Веб-приложение ASP.NET Core (MVC)	`mvc`	C#, F#
Веб-приложение Blazor	`blazor`	C#
Библиотека класса	`classlib`	C#, F#, VB
Консольное приложение	`console`	C#, F#, VB
Файл `EditorConfig`	`editorconfig`	
Файл `global.json`	`globaljson`	
Файл решения	`sln`	
Тестовый проект xUnit	`xunit`	

Проекты .NET MAUI не поддерживают Linux. По словам сотрудников компании, ее поддержка делегирована сообществу сторонних разработчиков в рамках проекта с открытым исходным кодом. Если вам необходимо создать действительно кросс-платформенное GUI-приложение, взгляните на проект Avalonia по адресу https://avaloniaui.net/.

Установка дополнительных пакетов шаблонов

Разработчикам доступно множество дополнительных шаблонов в виде пакетов.

1. Запустите браузер и перейдите на сайт https://www.nuget.org/packages.
2. Введите запрос vue в текстовое поле Search for packages (Поиск пакетов). В результате будет найдено 260 пакетов (на момент написания книги).
3. В разделе Package Type (Тип пакета) установите переключатель в положение Template (Шаблон), нажмите кнопку Apply (Применить), и вы увидите список из 33 доступных шаблонов, один из которых выпущен компанией Microsoft.

4. Выберите пакет Vue.Simple.Template, затем щелкните на ссылке Project website (Спроектировать сайт). Для установки и использования этого шаблона предназначены следующие команды:

```
dotnet new --install "Vue.Simple.Template"
dotnet new simplevue
```

5. Закройте браузер.

Обзор высокоуровневых программ, функций и пространств имен

Начиная с версии .NET 6, в дефолтном шаблоне консольных приложений используются операторы верхнего уровня, добавленные в .NET 5. Важно понять, как они взаимодействуют с автоматически сгенерированным классом `Program` с методом `<Main>$`.

Рассмотрим, как функционируют данные операторы верхнего уровня, когда вы определяете функции.

1. В редакторе кода создайте консольное приложение со следующими настройками:
   - выберите шаблон проекта Console App (Консольное приложение);
   - в поле Project name (Имя проекта) укажите значение TopLevelFunctions;
   - в поле Solution name (Имя решения) укажите значение Chapter01;
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
   - снимите флажок Enable native AOT publishing (Включить публикацию Native AOT).
2. В файле `Program.cs` удалите сгенерированные операторы, определите локальную функцию в конце файла и вызовите ее:

```
using static System.Console;

WriteLine("Top-level functions example");

WhatsMyNamespace();     // Вызов функции

void WhatsMyNamespace() // Определение локальной функции
{
  WriteLine("Namespace of Program class: {0}",
    arg0: typeof(Program).Namespace ?? "null");
}
```

3. Запустите консольное приложение и обратите внимание на пространство имен `null` для класса `Program`:

```
Top-level functions example
Namespace of Program class: null
```

Обработка локальной функции в процессе компиляции

Компилятор автоматически генерирует класс `Program` с функцией `<Main>$`, помещает операторы и функцию в метод `<Main>$` и переименовывает локальную функцию:

```
using static System.Console;

partial class Program
{
  static void <Main>$(String[] args)
  {
    WriteLine("Top-level functions example");

    <<Main>$>g__WhatsMyNamespace|0_0();     // Вызов функции

    void <<Main>$>g__WhatsMyNamespace|0_0() // Определение локальной функции
    {
      WriteLine("Namespace of Program class: {0}",
        arg0: typeof(Program).Namespace ?? "null");
    }
  }
}
```

Для успешной компиляции проекта необходимо указывать операторы согласно следующим правилам.

- Операторы импорта (`using`) должны находиться в начале файла `Program.cs`.
- Содержимое функции `<Main>$` — в середине файла `Program.cs`. Все функции в методе `<Main>$` *локальны*.

Последний пункт крайне важен, потому что у локальных функций есть определенные ограничения, к примеру, их нельзя документировать XML-комментариями.

Размещение статических функций в отдельном файле класса Program

Лучше всего размещать все функции в отдельном файле и определять их как статические (`static`) члены класса `Program`.

1. Создайте файл класса и присвойте ему имя `Program.Functions.cs`. Имя можно задать любое, но лучше, если оно будет информативным и указывать на связь с файлом класса `Program.cs`.

2. В файле `Program.Functions.cs` определите разделяемый (`partial`) класс `Program`. Затем вырежьте из файла `Program.cs` функцию `WhatsMyNamespace()` и вставьте ее в файл `Program.Functions.cs`. Добавьте ключевое слово `static`:

```
using static System.Console;

// Не определяйте пространство имен, чтобы этот
// класс попал в пустое пространство имен
// по умолчанию так же, как и автоматически
// генерируемый разделяемый класс Program.

partial class Program
{
  static void WhatsMyNamespace() // Определение статической функции
  {
    WriteLine("Namespace of Program class: {0}",
      arg0: typeof(Program).Namespace ?? "null");
  }
}
```

3. В файле `Program.cs` должны остаться три оператора:

```
using static System.Console;

WriteLine("Top-level functions example");

WhatsMyNamespace();   // Вызов функции
```

4. Запустите консольное приложение. На этапе выполнения не должно быть никаких изменений по сравнению с предыдущим запуском.

Обработка статической функции в процессе компиляции

Итак, у нас есть отдельный файл с разделяемым (`partial`) классом `Program` и статическими функциями. Обрабатывая его, компилятор определяет класс `Program` с функцией `<Main>$` и перемещает внутрь нее ваши операторы верхнего уровня, а затем выполняет внедрение функции в качестве члена класса `Program`:

```
using static System.Console;

partial class Program
{
  static void <Main>$(String[] args)
  {
    WriteLine("Top-level functions example");

    WhatsMyNamespace(); // Вызов функции
  }
```

```
  static void WhatsMyNamespace() // Определение статической функции
  {
    WriteLine("Namespace of Program class: {0}",
      arg0: typeof(Program).Namespace ?? "null");
  }
}
```

Так гораздо лучше, и вы можете документировать функции XML-комментариями. Их текст отображается во всплывающих подсказках в редакторе кода при вызове функции.

Дельный совет: соберите все функции из файла Program.cs в отдельный документ и вручную определите их в разделяемом классе Program. Так они будут добавлены в автоматически генерируемый класс Program на том же уровне, что и метод <Main>$, а не в качестве локальных функций внутри него.

Обратите внимание на отсутствие объявления пространств имен. Оба класса `Program`, и автоматически сгенерированный, и явно определенный, располагаются в стандартном пространстве имен — `null`. Определяя разделяемый класс `Program` в ином пространстве имен, вы предотвращаете его слияние с автоматически сгенерированным разделяемым классом `Program`.

Дельный совет: не определяйте пространство имен для всех создаваемых разделяемых классов Program. Так они будут автоматически определены в стандартном пространстве null.

Вы можете все статические методы класса `Program` явно объявить приватными (`private`), хотя они таковы по умолчанию. Поскольку все функции вызываются внутри класса `Program`, модификатор доступа не нужен.

GitHub-репозиторий для этой книги

Git — это популярная система управления исходным кодом, а *GitHub* — это компания, сайт и классическое приложение, упрощающее управление данными в Git. Корпорация Microsoft приобрела компанию GitHub в 2018 году, чем объясняется тесная интеграция с инструментами Microsoft.

Для этой книги я создал GitHub-репозиторий. Он используется для следующих целей:

- хранения файлов с примерами кода из книги, которые я поддерживаю после выхода книги в печать;
- размещения дополнительных материалов, дополняющих книгу, например исправлений ошибок, небольших улучшений, списков полезных ссылок и объемных сопроводительных публикаций, не вместившихся в печатную книгу;
- общения читателей со мной, если у них возникнут проблемы.

При возникновении вопросов по книге

Если вы оказались в тупике, выполняя шаги в этой книге, или заметили ошибку в тексте или коде, пожалуйста, задайте вопрос в репозитории GitHub.

1. В браузере перейдите по адресу https://github.com/markjprice/apps-services-net8/issues.
2. Нажмите кнопку New Issue (Новый вопрос).
3. Укажите как можно больше подробностей, которые помогут мне понять проблему. В том числе:
   - название раздела книги, номер страницы и номер шага;
   - используемый редактор кода, например Visual Studio 2022, Visual Studio Code или другой, в том числе номер версии;
   - всю необходимую информацию о коде, с которым возникли затруднения, и конфигурации;
   - описание ожидаемого и полученного результатов;
   - снимки экрана (вы можете перетаскивать графические файлы в поле описания проблемы).

Следующая информация менее актуальна, но тоже может оказаться полезной:

- используемая операционная система, скажем 64-битная версия Windows 11 или macOS Big Sur версии 11.2.3;
- используемое аппаратное обеспечение, например процессор Intel, Apple Silicon или ARM.

Я хочу, чтобы с помощью моей книги все читатели добились успеха в своих начинаниях, поэтому я с радостью постараюсь помочь вам (и всем остальным).

Загрузка кода решения из репозитория GitHub

В репозитории GitHub размещен код всех практических пошаговых примеров и упражнений из этой книги. Репозиторий доступен по адресу https://github.com/markjprice/apps-services-net8.

Если вы хотите скачать все файлы примеров на свой компьютер, нажмите зеленую кнопку Code (Код) и выберите пункт Download ZIP (Загрузить архив ZIP).

Рекомендую добавить показанную выше ссылку в избранное браузера.

Дельный совет: рекомендую клонировать/скачивать файлы примеров в папку с коротким именем, например C:\cs12dotnet8\ или C:\book\, чтобы предотвратить превышение максимальной длины пути у генерируемых компилятором файлов. Также следует избегать специальных символов, таких как #. Например, не используйте имя папки C:\C# projects\. Такое имя хотя и подойдет для записи проекта консольного приложения, но при автоматической генерации кода компилятором у вас возникнут проблемы. Имена папок должны быть короткими и простыми.

Источники справочной информации

Этот раздел посвящен поиску качественной информации о программировании в Интернете.

Техническая документация на сайте Microsoft Learn

Исчерпывающий ресурс для получения помощи по инструментам и платформам Microsoft для разработчиков находится в технической документации на Microsoft Learn, которую вы найдете по адресу https://learn.microsoft.com/ru-ru/docs/.

Получение справки с помощью команды dotnet

В терминале вы можете запросить документацию по командам с помощью инструмента `dotnet`. Синтаксис следующий:

```
dotnet help <команда>
```

В браузере откроется страница документации по указанной *команде*. К распространенным `dotnet`-командам относятся `new`, `build` и `run`.

Внимание! Команда dotnet help new работала в версиях .NET Core 3.1 — .NET 6, а в .NET 7 и более поздних релизах возвращает ошибку: «Указанная команда new не является допустимой командой пакета SDK. Укажите допустимую». Для получения дополнительных сведений выполните команду dotnet help. Надеюсь, разработчики .NET скоро исправят эту ошибку!

Документацию можно вывести и в самом терминале. Синтаксис следующий:

```
dotnet <команда> -?|-h|--help
```

Так, команда `dotnet new -?`, или `dotnet new -h`, или `dotnet new --help` выводит в терминале документацию о команде `new`.

Итак, команда dotnet help help открывает в браузере страницу документации к команде help, а dotnet help -h выводит документацию для команды help в терминале!

Приведу пару примеров.

1. Чтобы открыть официальную документацию в окне браузера для `dotnet`-команды `build`, введите следующий код в терминале Visual Studio Code:

```
dotnet help build
```

Обратите внимание на страницу, открывшуюся в браузере (рис. 1.2).

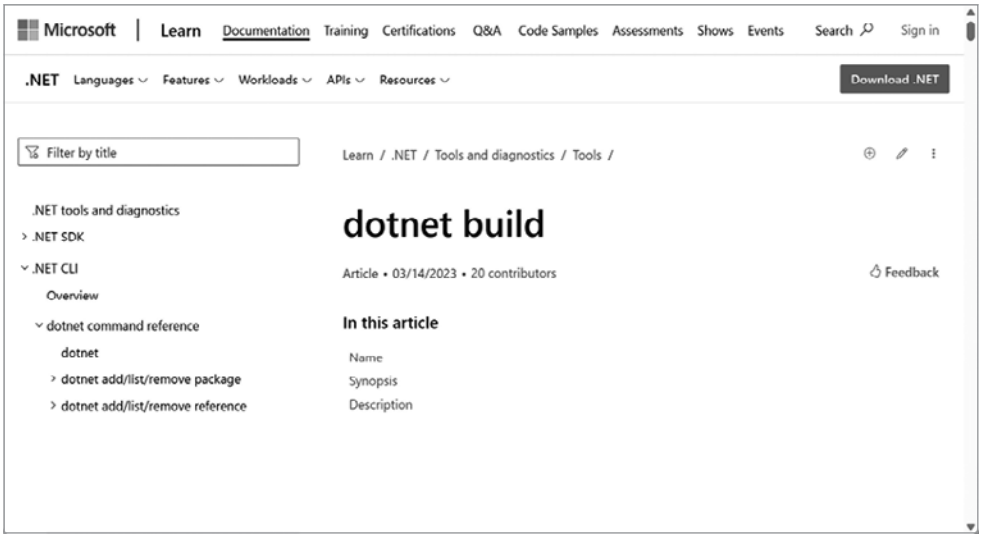


Рис. 1.2. Веб-страница документации для команды dotnet build

2. Чтобы вывести документацию в терминале, используйте флаг `-?`, `-h` или `--help`:

```
dotnet build -?
```

3. Вы увидите следующий вывод (фрагмент):

```
Description:
  .NET Builder

Usage:
  dotnet build [<PROJECT | SOLUTION>...] [options]

Arguments:
  <PROJECT | SOLUTION> The project or solution file to operate on. If a file is
not specified, the command will search the current directory for one.

Options:
  --ucr, --use-current-runtime        Use current runtime as the target runtime.
  -f, --framework <FRAMEWORK>         The target framework to build for.
The target framework must also be specified in the project file.

...
  -?, -h, --help                      Show command-line help.
```

4. Просмотрите документацию для команд `add`, `help`, `list`, `new` и `run`.

Инструменты ИИ, такие как ChatGPT и GitHub Copilot

Одним из самых значительных изменений в области программирования и разработки за последний год стало появление универсальных инструментов *искусственного интеллекта (ИИ)*, помогающих в решении таких задач, как автоматическое дополнение операторов, реализация целых функций, генерация модульных тестов, а также отладка и улучшение загруженного кода.

ChatGPT

На момент написания книги используются две модели ChatGPT: 3.5 (бесплатно) и 4.0 (20 долларов в месяц).

Допустим, вам необходимо больше узнать о том, как разработать микросервис. Вы можете ввести следующий запрос:

```
You are an expert .NET solution architect with experience advising global
enterprises. What are the top three most common mistakes when designing a
microservice architecture that are serious enough to cause a project to be
abandoned?
```

В ответ на это нейросеть выведет краткое описание трех главных ошибок, приведенное на рис. 1.3 и далее в тексте.

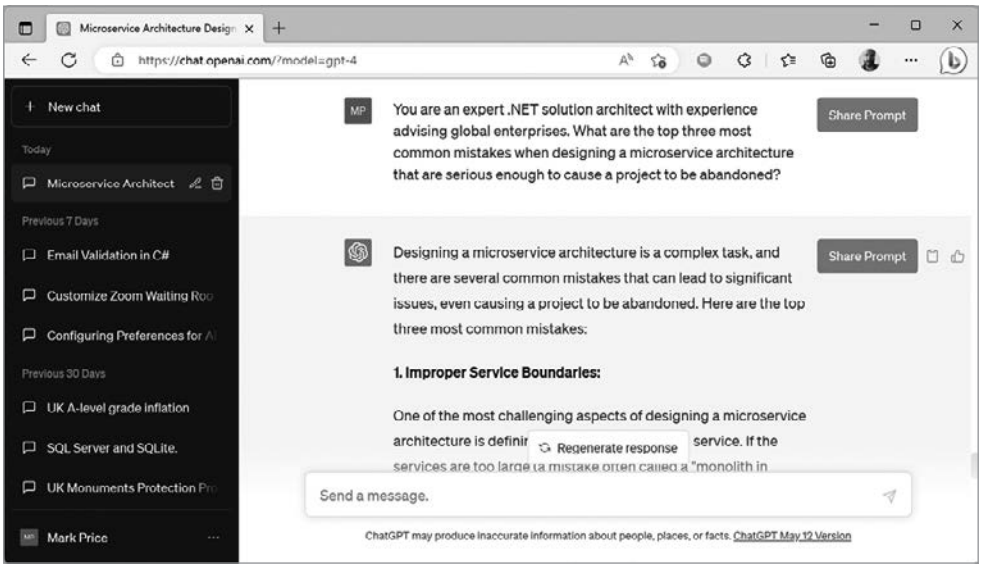


Рис. 1.3. ChatGPT вещает о проектировании архитектуры микросервисов

«Проектирование микросервисной архитектуры — это сложный и многоэтапный процесс, требующий внимательности и глубокого понимания. Ниже представлены

три наиболее распространенные ошибки, которые могут быть достаточно серьезными, чтобы привести к отказу от проекта.

1. **Неправильное определение границ микросервисов.** Одна из самых частых ошибок — это нечеткое или неправильное определение границ между микросервисами. Если границы не продуманы должным образом, это может привести к излишнему взаимозависимому коду, что, в свою очередь, вызовет проблемы с масштабируемостью и сложностью тестирования. Например, если бизнес-логика и данные оказываются переплетены в нескольких сервисах, то вместо независимости микросервисов можно получить монолит, который трудно поддерживать и развивать.

   Решение: перед проектированием микросервисов следует глубоко проанализировать бизнес-процессы и понять, какие функции могут быть разделены на независимые сервисы. Также стоит воспользоваться принципом Bounded Context, чтобы выделить области, которые могут быть автономными.

2. **Отсутствие единой стратегии управления данными.** В микросервисной архитектуре каждый сервис должен управлять собственными данными, однако это создает вызовы для синхронизации данных между сервисами и поддержания согласованности. Многие команды упрощают этот момент, выбрав подход с централизованной базой данных, что может привести к проблемам с масштабируемостью, производительностью и отказоустойчивостью. Кроме того, отсутствие четкой стратегии для работы с транзакциями (например, в распределенных системах) может вызвать проблемы с согласованностью данных.

   Решение: необходимо использовать подходы к управлению данными, такие как Event Sourcing или CQRS (Command Query Responsibility Segregation), для того чтобы обеспечить согласованность и независимость данных между сервисами. Важно также четко продумать стратегию управления транзакциями и синхронизации данных через механизмы, такие как «саги» или асинхронные сообщения.

3. **Невозможность эффективного мониторинга и управления.** Микросервисная архитектура увеличивает сложность отслеживания и управления приложениями, так как сервисов становится много и их взаимодействия могут быть сложными. Ошибки при проектировании системы мониторинга и логирования могут привести к тому, что проблемы, возникающие в сервисах, будет трудно обнаружить и диагностировать. Это особенно актуально при появлении новых багов в продакшен-среде, когда отсутствие надлежащего мониторинга замедляет реакцию на инциденты.

   Решение: важно заранее спроектировать и внедрить систему мониторинга, логирования и трассировки (например, с использованием таких инструментов, как Prometheus, Grafana, ELK Stack или OpenTelemetry), чтобы иметь возможность быстро выявлять проблемы в системе. Кроме того, стоит внедрить автоматические тесты, CI/CD и стратегии для контроля производительности.

Эти ошибки могут привести к значительным затратам в ресурсах, времени и деньгах, а также создать потенциально неустойчивую систему, что, в свою очередь, может повлиять на успех всего проекта.

GitHub Copilot

Компания Microsoft поддерживает сервис для программистов GitHub Copilot, автоматически дополняющий код прямо в редакторе. В настоящее время сервис совершенствуется с помощью нейросети GPT-4. Вы можете скачать плагины GitHub Copilot для редакторов кода, в том числе Visual Studio 2022, Visual Studio Code и среды разработки на базе JetBrains IntelliJ.

Проект GitHub Copilot бесплатен для студентов, преподавателей и ряда разработчиков проектов с открытым исходным кодом. Для всех остальных физических лиц доступна 30-дневная бесплатная пробная версия, а затем подключается подписка стоимостью 10 долларов в месяц или 100 долларов в год. С платной учетной записью вам будут доступны продвинутые экспериментальные функции GitHub Copilot X.

На сайте https://github.com/features/copilot указано, какие функции Copilot доступны для различных редакторов кода. Мир быстро меняется, и многое из написанного в этой книге устареет к моменту чтения.

Компания JetBrains выпустила аналогичный проект — AI Assistant. Прочитать о нем вы можете по адресу https://blog.jetbrains.com/idea/2023/06/ai-assistant-in-jetbrains-ides/.

Оформить учетную запись GitHub Copilot можно по адресу https://github.com/github-copilot/signup/.

Подписка на официальный блог .NET и уведомления

Чтобы быть в курсе событий .NET, рекомендую подписаться на официальный блог .NET, который ведут команды разработчиков .NET. Он доступен по адресу https://devblogs.microsoft.com/dotnet/.

Рекомендую также подписаться на официальный репозиторий анонсов .NET по адресу https://github.com/dotnet/announcements.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 1.1. Проверьте свои знания

Ответьте на следующие вопросы (при необходимости пользуйтесь источниками информации в Интернете).

1. Для чего в файлы проекта добавляется показанный ниже код? А когда его не следует приводить?

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

2. Для какой технологии сервиса требуется протокол HTTP/2 или версии выше?
3. В 2010 году в вашей компании был создан сервис на базе .NET Framework и WCF. На какую технологию лучше всего его перенести и почему?
4. Какой редактор кода или среду разработки следует установить для программирования .NET-проектов?
5. Чего следует остерегаться при создании ресурсов Azure?

Упражнение 1.2. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-1---introducing-apps-and-services-with-net.

Резюме

В этой главе вы:

- познакомились с технологиями создания приложений и сервисов, описываемыми в этой книге;
- настроили среду разработки;
- узнали, где искать справочную информацию.

В следующей главе вы научитесь использовать SQL Server для хранения и управления реляционными данными.

2 Управление реляционными данными в системе SQL Server

Эта глава посвящена управлению реляционными данными в системе SQL Server, Azure SQL Database или Azure SQL Edge. Сначала вы узнаете, как управлять данными с помощью нативных операторов расширения Transact-SQL. Затем — как управлять данными на низком уровне с помощью библиотек ADO.NET (`Microsoft.Data.SqlClient`). Наконец, вы воспользуетесь инструментом Dapper, упрощающим управление моделями сущностей.

Создание современной БД

Два наиболее распространенных варианта хранения данных:

- *реляционная система управления базами данных* (Relational Database Management System, RDBMS или *РСУБД*), такая как *SQL Server*, *PostgreSQL*, *MySQL* и *SQLite*;
- *база данных NoSQL*, такая как *Azure Cosmos DB*, *MongoDB*, *Redis* и *Apache Cassandra*.

В этой главе мы рассмотрим самую популярную РСУБД для Windows (а также Linux) — SQL Server. Для кросс-платформенной разработки можно использовать либо Azure SQL Database с хранением данных в облаке, либо Azure SQL Edge, запускаемую в контейнере Docker под Windows, macOS или Linux, а также на процессорах архитектуры Intel и ARM.

Использование образца реляционной БД

Чтобы научиться управлять РСУБД с помощью .NET, понадобится образец БД средней сложности и с приличным количеством записей. На сайте компании Microsoft доступно несколько образцов БД, в большинстве своем слишком сложных для наших нужд, поэтому мы воспользуемся БД *Northwind*, разработанной в начале 1990-х годов.

Взгляните на диаграмму БД Northwind с восемью наиболее важными таблицами. К диаграмме на рис. 2.1 можно обращаться при написании кода и запросов в этой книге.

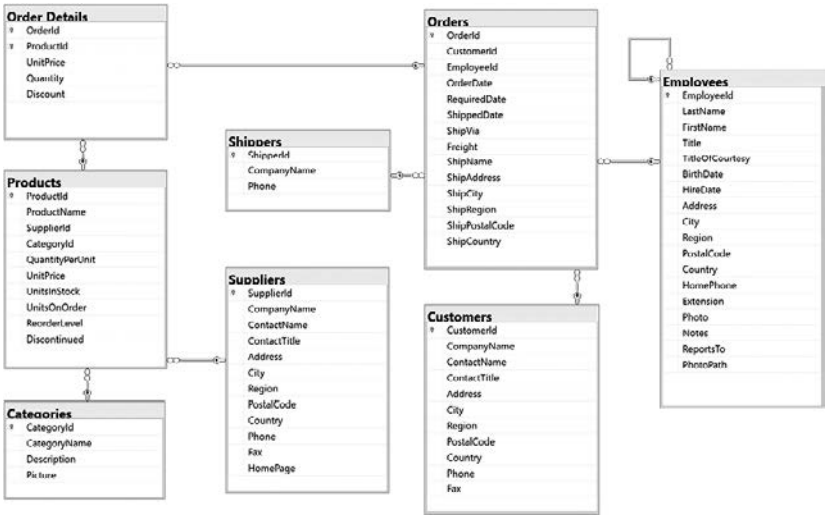


Рис. 2.1. Таблицы и взаимосвязи БД Northwind

Обратите внимание, что:

- у каждой категории есть уникальный идентификатор, название, описание и изображение. Изображение хранится в виде массива байтов в формате JPEG;
- у каждого товара есть уникальный идентификатор, название, цена единицы, количество единиц на складе и другие столбцы;
- каждый товар ассоциируется с категорией по ее уникальному идентификатору;
- отношения между таблицами `Categories` и `Products` — «один ко многим», то есть каждая категория может содержать один, несколько или вовсе не иметь товаров;
- каждый товар поставляется компанией с уникальным идентификатором;
- в заказе указываются количество и цена единицы товара;
- каждый заказ оформляется клиентом, принимается сотрудником и отправляется транспортной компанией;
- у каждого сотрудника есть имя, адрес, контактные данные, даты рождения и приема на работу, отсылка на своего руководителя (за исключением его самого, в поле `ReportsTo` которого указано значение `null`) и фотография в виде массива байтов в формате JPEG. В таблице используется отношение «один ко многим», поскольку один сотрудник может управлять другими сотрудниками.

Подключение к БД SQL Server

Чтобы подключиться к БД SQL Server, нам понадобится такая информация:

- *имя сервера* (и *имя экземпляра*, если их несколько). При сетевом подключении может включать протокол, IP-адрес и номер порта;
- *имя БД*;
- *конфиденциальные данные*, например имя пользователя и пароль, либо автоматическая передача данных подключающегося пользователя системой проверки подлинности Windows.

Эту информацию следует указывать в *строке подключения* к SQL Server совместно с опциональным рядом параметров:

- `Data Source`, `server` или `addr` — эти параметры передают имя сервера (и при необходимости экземпляр). Точкой (`.`) обозначается локальный сервер;
- `Initial Catalog` или `database` — позволяют указать имя изначально активной БД. Можно изменить с помощью SQL-оператора `USE <имя_БД>`;
- `Integrated Security` или `trusted_connection` — параметрам присваивается значение `true` или `SSPI` в случае передачи учетных данных текущего пользователя системой проверки подлинности Windows;
- `User Id` и `Password` — предназначены для проверки подлинности во всех редакциях SQL Server. Обязательно используются в системах Azure SQL Database или Azure SQL Edge, не поддерживающих проверку подлинности Windows. Полная Windows-версия SQL Server допускает передачу как имени пользователя с паролем, так и данных системы проверки подлинности Windows;
- `Authentication` — параметр для проверки подлинности с помощью идентификаторов Azure AD, допускающих вход без пароля. Допустимые значениями: `Active Directory Integrated`, `Active Directory Password` и `Sql Password`;
- `Persist Security Info` — если данному параметру присвоено значение `false`, ненадежный источник не сможет получить доступ к конфиденциальным данным строки подключения. После проверки подлинности из строки подключения удаляется поле `Password`;
- `Encrypt` — если данному параметру присвоено значение `true`, подключение осуществляется по протоколу SSL в целях шифрования данных, передаваемых между клиентом и сервером;
- `TrustServerCertificate` — присвойте этому параметру значение `true`, если в локальном хостинге у вас возникает ошибка «Подключение к серверу успешно установлено, но затем произошла ошибка при входе (поставщик: поставщик SSL, ошибка: 0 — цепочка сертификатов выпущена центром сертификации, не имеющим доверия)»;
- `Connection Timeout` — по умолчанию этому параметру присвоено значение, соответствующее 30 секундам;

- `MultipleActiveResultSets` — присвойте этому параметру значение `true`, чтобы в целях повышения производительности использовать одно подключение при одновременной работе с несколькими таблицами. Используется для отложенной загрузки строк из связанных таблиц.

Как я уже говорил, при составлении строки подключения к БД SQL Server вам необходимо знать имя сервера. Оно зависит от редакции и версии SQL Server (табл. 2.1).

Таблица 2.1. Примеры имен серверов для различных редакций SQL Server

Редакция SQL Server	Имя сервера\имя экземпляра
LocalDB 2012	`(localdb)\v11.0`
LocalDB 2016 или более поздней версии	`(localdb)\mssqllocaldb`
Express	`.\sqlexpress`
Full/Developer (экземпляр по умолчанию)	`.`
Full/Developer (именованный экземпляр)	`.\apps-services-book`
Azure SQL Edge (локальный Docker)	`tcp:127.0.0.1,1433`
Azure SQL Database	`tcp:[`*имя_сервера*`].database.windows.net,1433`

Дельный совет: для краткости вместо полного имени локального компьютера (localhost) указывайте точку (.). Не забывайте, что имена серверов SQL Server могут состоять из двух частей: имени компьютера и имени экземпляра SQL Server. Имена экземпляров настраиваются во время установки сервера.

Установка и настройка локальной версии SQL Server

Компания Microsoft предоставляет различные редакции популярной и функциональной системы SQL Server для Windows, Linux и контейнеров Docker. На компьютере под управлением Windows вы можете использовать бесплатную автономную версию SQL Server Developer Edition. Вы также можете задействовать редакцию Express или бесплатную версию SQL Server LocalDB, устанавливаемую совместно с ПО Visual Studio 2022 для Windows.

Если у вас нет компьютера под управлением Windows или вы хотите развернуть кросс-платформенную систему БД, перейдите к подразделу «Настройка Azure SQL Database» далее в этом разделе.

Если вы планируете развернуть локальную версию SQL Server в Linux, изучите инструкции по адресу https://learn.microsoft.com/ru-ru/sql/linux/sql-server-linux-setup.

Установка SQL Server Developer Edition для Windows

Если вы хотите использовать полную версию SQL Server, а не упрощенные редакции LocalDB и Express, перейдите к каталогу версий SQL Server по адресу https://www.microsoft.com/ru-ru/sql-server/sql-server-downloads.

Выполните следующие действия.

1. Загрузите версию для разработчиков — Developer.
2. Запустите программу установки.
3. Выберите тип установки Custom (Пользовательский).
4. Выберите каталог установки, а затем нажмите кнопку Install (Установить).
5. Дождитесь загрузки файлов дистрибутива (примерно 1,5 Гбайт).
6. В окне SQL Server Installation Center (Центр установки SQL Server) перейдите на вкладку Installation (Установка), а затем щелкните на ссылке New SQL Server standalone installation or add features to an existing installation (Новая установка изолированного экземпляра SQL Server или добавление компонентов к существующей установке) (рис. 2.2).

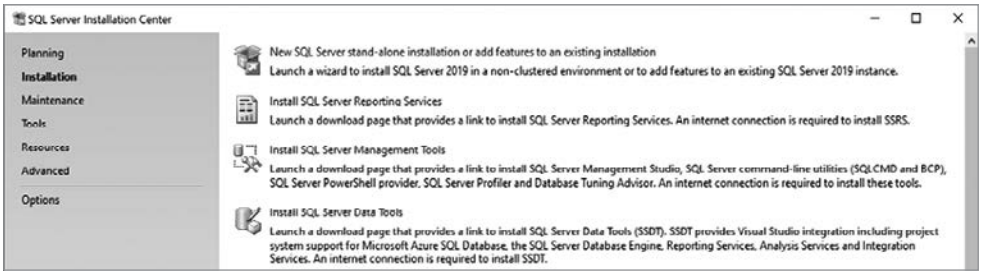


Рис. 2.2. Установка нового экземпляра SQL Server

7. Выберите бесплатный выпуск Developer в раскрывающемся списке, а затем нажмите кнопку Next (Далее).
8. Установив флажок, примите условия лицензии, а затем нажмите кнопку Next (Далее).
9. Просмотрите параметры службы Microsoft Update, а затем нажмите кнопку Next (Далее).
10. Просмотрите правила установки, исправьте обнаруженные ошибки. Вы можете проигнорировать предупреждения брандмауэра, так как порты не следует открывать, а затем нажмите кнопку Next (Далее).
11. Флажок Azure Extension for SQL Server (Расширение Azure для SQL Server) следует снять. Нажмите кнопку Next (Далее).

12. На вкладке Feature Selection (Выбор компонентов) выберите Database Engine Services (Службы ядра СУБД), а затем нажмите кнопку Next (Далее).
13. На вкладке Instance Configuration (Настройка экземпляра) установите переключатель в положение Default instance (Экземпляр по умолчанию), а затем нажмите кнопку Next (Далее). Если у вас уже настроен экземпляр по умолчанию, можно создать именованный экземпляр, к примеру, с именем `apps-services-book`.
14. На вкладке Server Configuration (Конфигурация сервера) обратите внимание на то, что ядро СУБД SQL Server настроено на автоматический запуск. Также настройте на автоматический запуск обозреватель SQL Server, который по умолчанию отключен, а затем нажмите кнопку Next (Далее).
15. В разделе Database Engine Configuration (Настройка ядра СУБД) на вкладке Server Configuration (Конфигурация сервера) установите переключатель Authentication Mode (Режим проверки подлинности) в положение Mixed Mode (Смешанный режим). Далее укажите надежный пароль учетной записи, нажмите кнопку Add Current User (Добавить текущего пользователя), а затем кнопку Next (Далее).
16. На вкладке Ready to Install (Все готово для установки) просмотрите сводку выполняемых настроек, а затем нажмите кнопку Install (Установить).
17. На вкладке Complete (Готово) обратите внимание на успешно выполненные действия, а затем нажмите кнопку Close (Закрыть).
18. В окне SQL Server Installation Center (Центр установки SQL Server) в разделе Installation (Установка) щелкните кнопкой мыши на ссылке Install SQL Server Management Tools (Установить средства управления SQL Server).
19. В окне браузера нажмите кнопку загрузки последней версии SSMS (рис. 2.3).

Рис. 2.3. Страница загрузки ПО SQL Server Management Studio (SSMS)

Прямая ссылка на дистрибутив SSMS выглядит следующим образом: https://learn.microsoft.com/ru-ru/sql/ssms/download-sql-server-management-studio-ssms.

20. Запустите программу установки SSMS и нажмите кнопку Install (Установить).
21. Когда программа установки завершит работу, нажмите кнопку Restart (Перезапустить) (при необходимости) или кнопку Close (Закрыть).

Вместе с SSMS автоматически устанавливается инструмент Azure Data Studio (ADS). Он кросс-платформенный и доступен свободно, поэтому вы можете использовать его для работы с БД SQL Server в любой операционной системе.

Расширение Visual Studio Code для работы с SQL Server

Существует множество инструментов, упрощающих работу с SQL Server. Если вы пользуетесь средой разработки Visual Studio Code, то можете установить расширение *SQL Server (mssql)*, `ms-mssql.mssql`. После этого на боковой панели появится новое представление *SQL Server* (рис. 2.4).

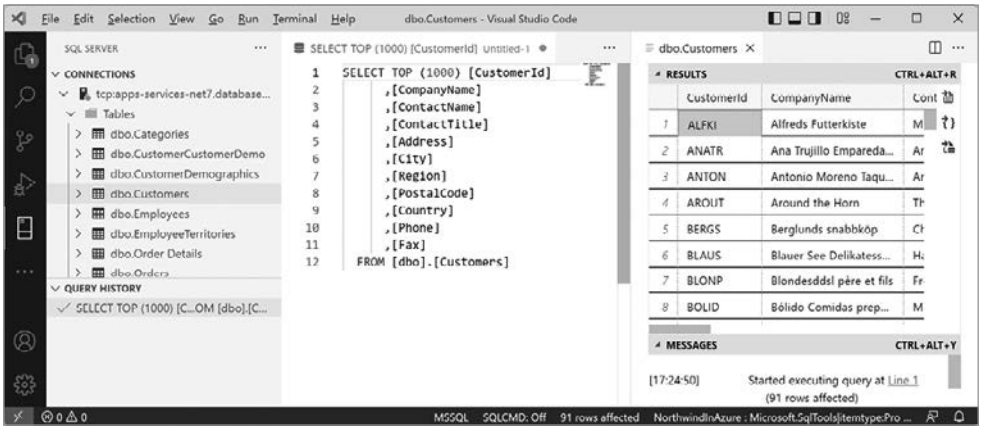


Рис. 2.4. Расширение SQL Server (mssql) для Visual Studio Code

Создание локального образца БД Northwind

Теперь мы можем запустить сценарий для создания локального образца БД Northwind в Windows с помощью среды SQL Server Management Studio (SSMS).

1. Если вы еще не скачали/клонировали GitHub-репозиторий для этой книги, то сделайте это сейчас по ссылке https://github.com/markjprice/apps-services-net8/.
2. В папке `apps-services-net8` создайте каталог `Chapter02`.
3. Скопируйте сценарий создания БД Northwind для SQL Server из локального Git-репозитория по следующему пути: `/scripts/sql-scripts/Northwind4SQLServer.sql` в папку `Chapter02`.
4. Запустите программу SQL Server Management Studio.

5. В диалоговом окне Connect to Server (Подключение к серверу) в поле Server name (Имя сервера) введите . (точку), то есть имя локального компьютера, а затем нажмите кнопку Connect (Подключить).

Внимание! Если вам необходимо создать именованный экземпляр БД, например apps-services-book, введите значение .\apps-services-book. В случае возникновения ошибки, связанной с сертификатом сервера, нажмите кнопку Options >> (Параметры >>) и установите флажок Trust server certificate (Доверять сертификату сервера).

6. Выберите команду меню File ▸ Open ▸ File (Файл ▸ Открыть ▸ Файл).
7. Выберите файл `Northwind4SQLServer.sql` и нажмите кнопку Open (Открыть).
8. На панели инструментов нажмите кнопку Execute (Выполнить) и обратите внимание на сообщение Command(s) completed successfully (Команда успешно завершена).
9. На панели Object Explorer (Обозреватель объектов) раскройте БД Northwind, а затем строку Tables (Таблицы).
10. Щелкните правой кнопкой мыши на строке dbo.Products, выберите в контекстном меню пункт Select Top 1000 Rows (Выбрать 1000 верхних строк) и обратите внимание на результаты (рис. 2.5).

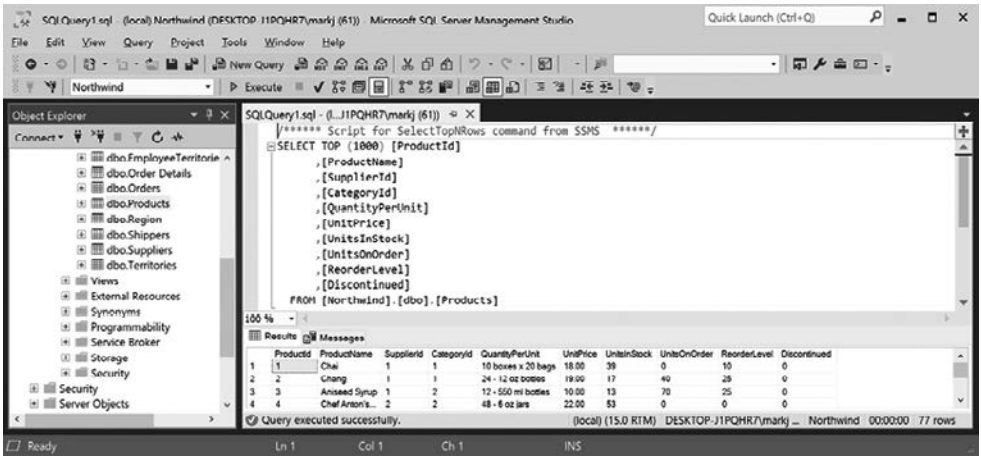


Рис. 2.5. Таблица Products в окне программы SQL Server Management Studio

11. На панели Object Explorer (Обозреватель объектов) нажмите кнопку Disconnect (Отключить).
12. Завершите работу программы SQL Server Management Studio.

Для выполнения описанного выше сценария не обязательно использовать программу SQL Server Management Studio. Подойдут также Visual Studio 2022, в частности панели SQL Server Object Explorer (Обозреватель объектов SQL Server) и Server Explorer (Обозреватель серверов), или кросс-платформенные инструменты, такие как расширение для Visual Studio Code с поддержкой SQL Server или независимое решение Azure Data Studio, доступное для загрузки по адресу https://aka.ms/getazuredatastudio.

Настройка Azure SQL Database

Если у вас нет компьютера под управлением Windows, вы можете создать экземпляр SQL Server в облаке. Для этого вам понадобится учетная запись Azure. Зарегистрироваться можно по адресу https://signup.azure.com. Далее необходимо выполнить следующие действия.

1. Войдите в свою учетную запись Azure: https://portal.azure.com/.
2. Перейдите по адресу https://portal.azure.com/#create/hub.
3. Найдите пункт Resource group (Группы ресурсов) и нажмите кнопку Create (Создать).
4. Введите имя группы ресурсов `apps-services-book` и выберите подходящее местонахождение в раскрывающемся списке Region (Регион). Теперь нажмите кнопку Review + create (Обзор + создать) (рис. 2.6).

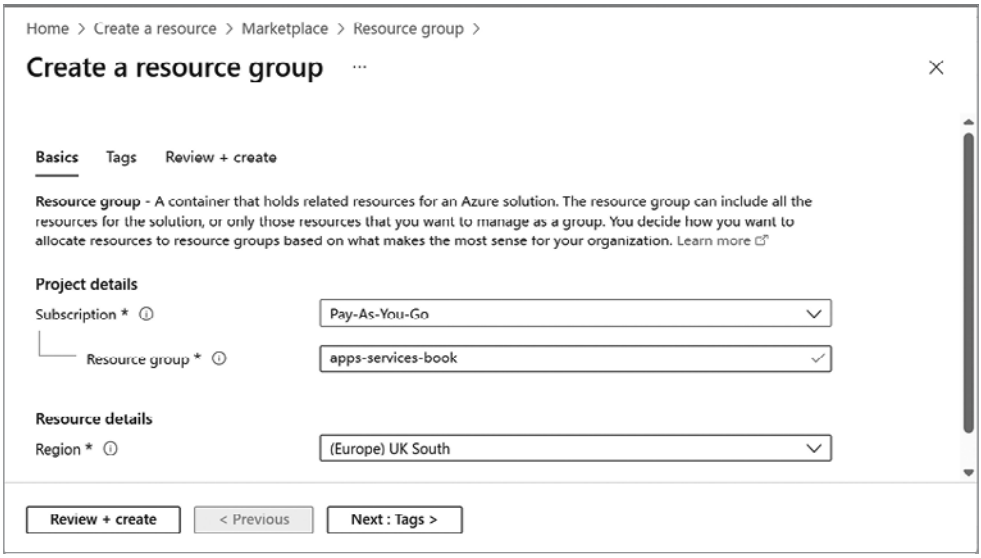


Рис. 2.6. Создание группы ресурсов на портале Azure

5. Проверьте настройки, а затем нажмите кнопку Create (Создать).

   Создадим еще один ресурс.
6. Найдите ресурс SQL Database (Базы данных SQL) и нажмите кнопку Create (Создать).
7. На вкладке Basics (Основные параметры) страницы Create SQL Database (Создание БД SQL) в поле Database name (Имя БД) введите слово Northwind и выберите группу ресурсов, созданную ранее.
8. В разделе Server (Сервер) нажмите кнопку Create New (Создать новый).
9. Введите следующие данные для сервера с БД SQL (рис. 2.7).

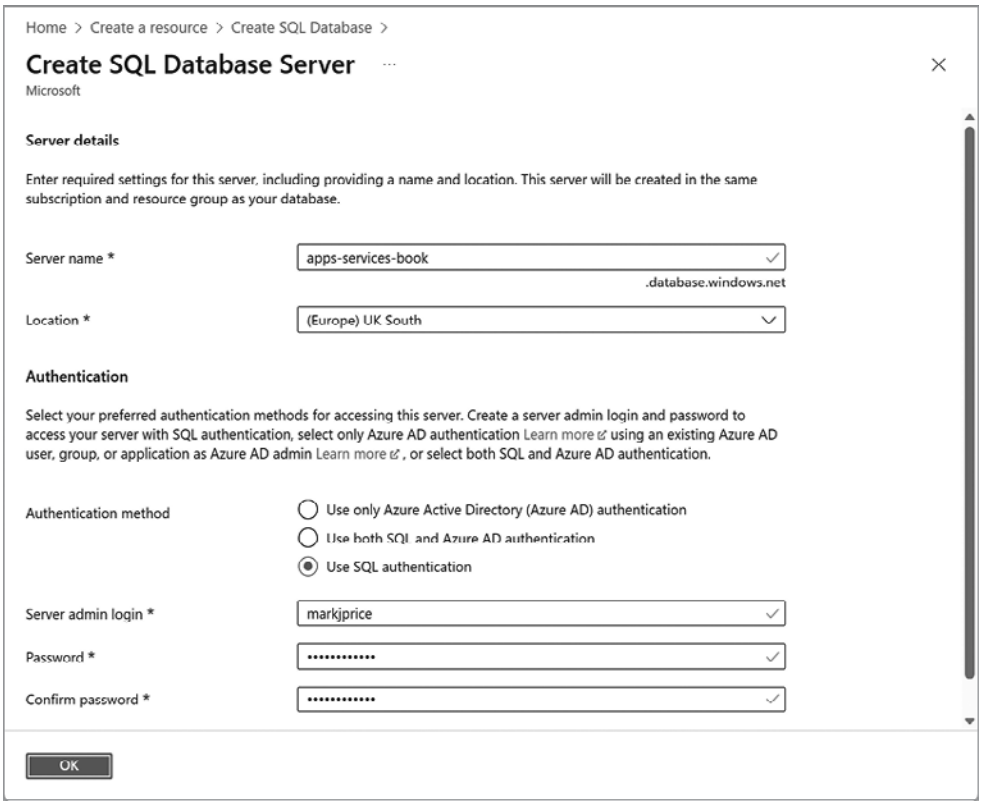


Рис. 2.7. Ввод данных сервера для экземпляра БД SQL

- Server name (Имя сервера): `apps-services-book-[`*`ваше_имя`*`]` или нечто в этом духе. Имя сервера должно быть уникальным среди всех БД, поскольку оно становится частью общедоступного URL-адреса.

- Location (Расположение): выберите подходящее местонахождение. Я выбрал вариант (Europe) UK South ((Европа) Великобритания Юг). Не все регионы доступны для тех или иных ресурсов. Если выбранный регион не поддерживается ресурсами сервера с БД SQL, возникнет ошибка.
- Authentication method (Метод проверки подлинности): выберите вариант с проверкой подлинности SQL.
- Server admin login (Логин администратора сервера): ваш адрес электронной почты или иное пользовательское имя. Я ввел значение `markjprice`.
- Password/Confirm password (Пароль/подтверждение пароля): введите надежный пароль.

10. Нажмите кнопку OK.
11. Присвойте параметру Want to use SQL elastic pool (Использовать эластичный пул SQL) значение No (Нет).
12. Присвойте параметру Workload environment (Рабочая среда) значение Development (вместо Production).
13. На странице Create SQL Database (Создание БД SQL) в разделе Compute + storage (Вычисления + хранилище) нажмите кнопку Configure database (Настроить БД).
14. Присвойте параметру Service tier (Уровень обслуживания) значение Basic (Базовый) — для менее требовательных рабочих нагрузок. Обратите внимание, что максимальный размер БД составляет 2 Гбайт, а предполагаемая стоимость — около 5 долларов в месяц (или менее 1 цента в час). Вы можете удалить ресурсы сразу после завершения чтения этой главы, чтобы сэкономить.
15. Нажмите кнопку Apply (Применить).
16. На странице Create SQL Database (Создание БД SQL) присвойте параметру Backup storage redundancy (Резервное хранилище копий) значение Locally-redundant backup storage (Локальное резервное хранилище).
17. Нажмите кнопку Next: Networking (Далее: сеть).
18. В разделе Network connectivity (Сетевое подключение) выберите пункт Public endpoint (Публичная конечная точка).
19. В разделе Firewall rules (Правила брандмауэра) присвойте параметру Add current client IP address (Добавить текущий IP-адрес клиента) значение Yes (Да).
20. Нажмите кнопку Next: Security (Далее: безопасность).
21. Изучите настройки, но не меняйте их.
22. Нажмите кнопку Next: Additional settings (Далее: дополнительные настройки).
23. Изучите настройки, но не меняйте их.
24. Нажмите кнопку Review + create (Обзор + создать).
25. Нажмите кнопку Create (Создать).
26. Дождитесь завершения развертывания (рис. 2.8).

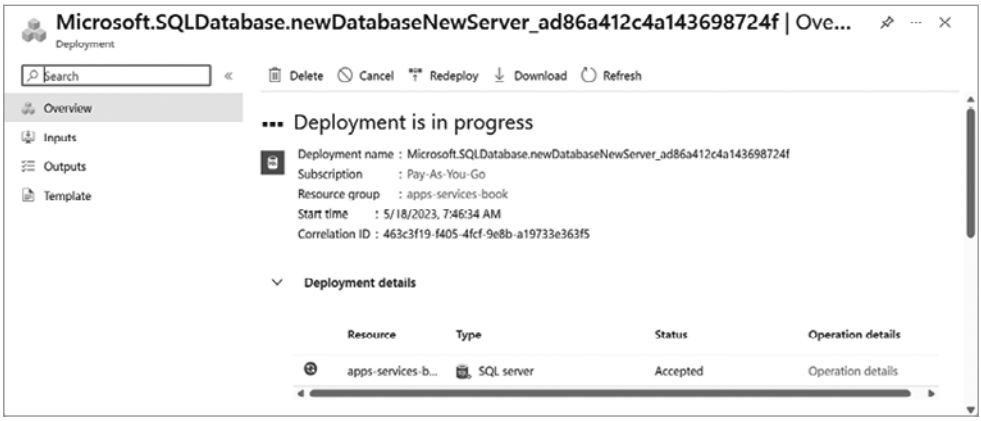


Рис. 2.8. Ход развертывания БД SQL

27. После завершения развертывания нажмите кнопку Go to resource (Перейти к ресурсу).
28. Перейдите на вкладку Overview (Обзор) и изучите подробное описание БД (рис. 2.9).

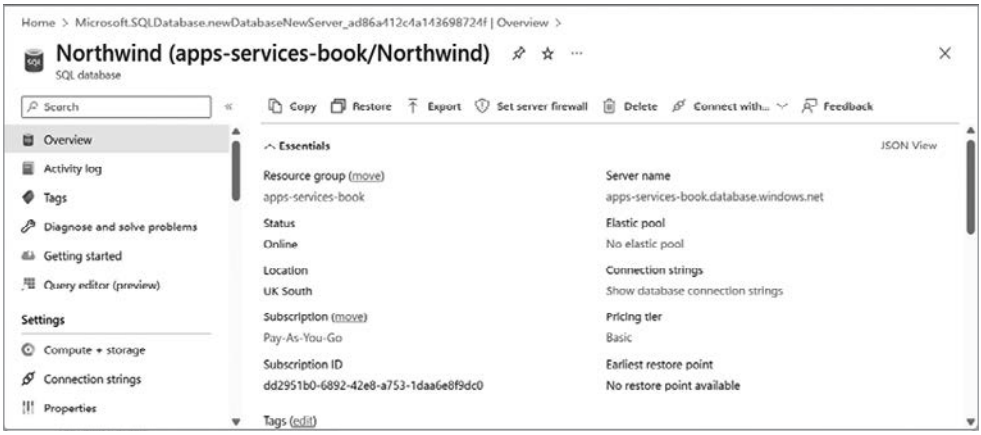


Рис. 2.9. Подробное описание БД SQL

29. Щелкните на ссылке See connection strings (Посмотреть строки подключения) (или перейдите на вкладку Connection strings (Строки подключения) на навигационной панели слева).
30. Скопируйте строку подключения из поля ADO.NET (SQL authentication) в буфер обмена.

31. Запустите Блокнот или любой другой текстовый редактор, вставьте строку подключения и для удобства разбейте ее на отдельные строки после каждой точки с запятой:

```
Server=tcp:apps-services-book.database.windows.net,1433;
Initial Catalog=Northwind;
Persist Security Info=False;
User ID=markjprice;
Password={your_password};
MultipleActiveResultSets=False;
Encrypt=True;
TrustServerCertificate=False;
Connection Timeout=30;
```

Ваше значение параметра Server будет иным, поскольку часть имени пользовательского сервера, в данном случае apps-services-book, общедоступна и уникальна.

32. При необходимости сохраните файл в приложении Блокнот для дальнейшего использования.

Окно инструмента JetBrains Rider для работы с SQL Server

В программе JetBrains Rider на любой операционной системе для подключения к БД SQL Server можно сделать следующее.

1. В JetBrains Rider выберите команду меню View ▸ Tool Windows ▸ Database (Вид ▸ Панели инструментов ▸ База данных).
2. На панели Database (База данных) нажмите кнопку Connect to database (Подключиться к БД).
3. Выберите пункт Use connection string (Использовать строку подключения).
4. Присвойте параметру Database type (Тип БД) значение Microsoft SQL Server.
5. В поле String (Строка) введите строку подключения к БД.
6. В поле {your_password} введите выбранный вами пароль.
7. Опционально нажмите кнопку Test Connection (Проверка подключения) и при возникновении ошибок исправьте их. Если вы получите ошибку языковой версии (Inconsistent language), можете проигнорировать ее, поскольку мы используем SQL Server в качестве диалекта.
8. Нажмите кнопку Connect to Database (Подключиться к БД).

Создание образца БД Northwind в облаке

Теперь мы можем запустить сценарий БД для создания образца БД Northwind в Azure SQL Database.

1. Используйте любой предпочитаемый инструмент для работы с БД, чтобы подключиться к SQL-серверу в Azure.

- В программе Visual Studio 2022 обратитесь к панели Server Explorer (Обозреватель серверов).
- В Windows запустите приложение SQL Server Management Studio.
- В Visual Studio Code обратитесь к представлению SQL Server.
- В JetBrains Rider выберите команду меню View ▸ Tool Windows ▸ Database (Вид ▸ Окна инструмента ▸ БД), а затем нажмите кнопку Connect to database (Подключиться к БД).

2. Укажите имя сервера и внесите в диалоговое окно все необходимые для подключения данные (рис. 2.10).

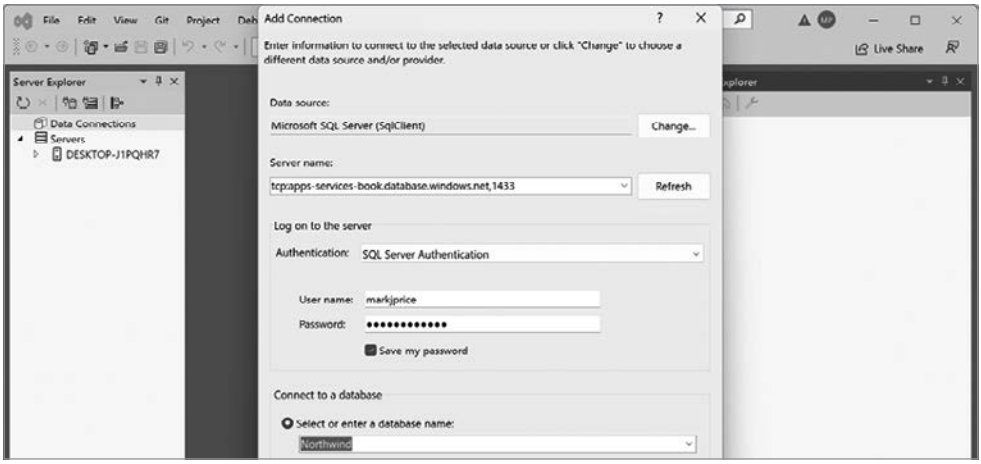


Рис. 2.10. Подключение к БД Azure SQL из Visual Studio

Может также появиться диалоговое окно Choose Data Source (Выбрать источник данных). Выберите вариант Microsoft SQL Server и при необходимости установите соответствующий флажок, чтобы данная настройка использовалась по умолчанию.

3. Щелкните правой кнопкой мыши на строке подключения данных и выберите в контекстном меню пункт New Query (Новый запрос).

В программе JetBrains Rider щелкните правой кнопкой мыши на пункте SQL Server, в контекстном меню выполните команду SQL Scripts ▸ Run SQL Script (Сценарии SQL ▸ Запустить сценарий SQL), а затем выберите файл Northwind4AzureSQLdatabase.sql.

4. Скопируйте и вставьте содержимое файла `Northwind4AzureSQLdatabase.sql` в окно запроса и отправьте его.

Сценарий Northwind4SQLServer.sql удаляет и создает БД Northwind заново. В сценарии Northwind4AzureSQLdatabase.sql этого не происходит, так как в этом случае БД функционирует как ресурс Azure. В этом заключается основное различие между этими двумя сценариями. Их можно скачать по адресу https://github.com/markjprice/apps-services-net8/tree/main/scripts/sql-scripts.

5. Дождитесь появления сообщения Command completed successfully (Команда выполнена успешно). Оно может появиться через несколько минут.
6. На панели Server Explorer (Обозреватель серверов) щелкните правой кнопкой мыши на строке Tables (Таблицы) и выберите в контекстном меню пункт Refresh (Обновить). Обратите внимание на создание 13 таблиц, таких как `Categories`, `Customers` и `Products`, а также на десятки представлений и хранимых процедур.

Теперь у вас в облаке запущена БД Azure SQL, к которой вы можете подключиться из проекта .NET.

Управление данными с помощью Transact-SQL

Transact-SQL (T-SQL) — это диалект языка структурированных запросов (Structured Query Language, SQL) в SQL Server.

В отличие от C# язык T-SQL нечувствителен к регистру. Например, вы можете указать тип `int` или `INT` для 32-битного целочисленного типа данных либо команду `SELECT` или `select` в начале выражения запроса. Текстовые данные, хранящиеся в таблицах SQL Server, могут как учитывать регистр, так и нет, это зависит от конфигурации.

Полный справочник по языку T-SQL доступен по адресу https://learn.microsoft.com/ru-ru/sql/t-sql/language-reference. Изучите такие темы, как Data types (Типы данных), Queries (Запросы) и Statements (Выражения).

Типы данных T-SQL

В T-SQL есть типы данных для столбцов, переменных, параметров и т. д. (табл. 2.2).

Таблица 2.2. Типы данных SQL Server, распределенные по категориям

Категория	Примеры
Числа	`bigint`, `bit`, `decimal`, `float`, `int`, `money`, `numeric`, `real`, `smallint`, `smallmoney`, `tinyint`
Дата и время	`date`, `datetime2`, `datetime`, `datetimeoffset`, `smalldatetime`, `time`
Текст	`char`, `nchar`, `ntext`, `nvarchar`, `text`, `varchar`
Бинарные	`binary`, `image`, `varbinary`
Другие	`cursor`, `hierarchyid`, `sql_variant`, `table`, `rowversion`, `uniqueidentifier`, `xml`

Есть тип данных xml, но нет типа json. Для хранения данных в формате JSON используйте тип nvarchar. В T-SQL также поддерживаются географические пространственные типы данных geometry и geography.

Документирование и комментирование кода

Чтобы закомментировать часть строки, используйте символы `--`, что эквивалентно `//`.

Чтобы закомментировать большой фрагмент кода или добавить многострочный комментарий, указывайте сочетание символов `/*` в начале и `*/` в конце, как и в C#.

Объявление переменных

Имена локальных переменных сопровождает префикс `@`. Они определяются с помощью операторов `SET`, `SELECT` или `DECLARE`:

```
DECLARE @WholeNumber INT; -- Объявление переменной и указание ее типа
SET @WholeNumber = 3; -- Присвоение переменной конкретного значения
SET @WholeNumber = @WholeNumber + 1; -- Приращение значения переменной
SELECT @WholeNumber = COUNT(*) FROM Employees; -- Указание количества сотрудников
SELECT @WholeNumber = EmployeeId FROM Employees WHERE FirstName = 'Janet';
```

Глобальные переменные имеют префикс `@@`. Например, значение переменной `@@ROWCOUNT` будет в зависимости от контекста возвращать количество строк, затронутых оператором в текущей области видимости, в частности число обновленных или удаленных строк.

Указание типов данных

Большинство типов имеют фиксированный размер. Например, `int` занимает четыре байта, `smallint` — два байта, а `tinyint` — один байт.

Для текстовых и бинарных типов вы можете указывать префикс `var` или `nvar` (то есть переменный размер). В этом случае их размер будет меняться автоматически (в пределах максимального) в зависимости от присвоенного значения, например так: `varchar(40)`. Либо можете ограничить количество выделяемых символов — `char(40)`.

Для текстовых типов префикс `n` указывает на стандарт Юникод, то есть символ занимает два байта. Текстовые типы без префикса `n` используют один байт на символ.

Управление потоком

В T-SQL есть ключевые слова управления потоком, аналогичные C#, например `BREAK`, `CONTINUE`, `GOTO`, `IF...ELSE`, `CASE`, `THROW`, `TRY...CATCH`, `WHILE` и `RETURN`. Основное отличие — в использовании операторов `BEGIN` и `END` для обозначения начала и конца блока вместо фигурных скобок в C#.

Операторы

В T-SQL применимы такие же операторы, как и в C#, например = (присваивание), +, -, *, /, %, <, >, <=, ==, !=, &, |, ^ и т. д. Поддерживаются логические операторы AND, OR, NOT и LINQ-подобные операторы ANY, ALL, SOME, EXISTS, BETWEEN и IN.

Для сопоставления текстовых шаблонов используется ключевое слово LIKE. При составлении шаблонов поддерживается символ % для обозначения любого количества символов, символ _ для обозначения одного символа и скобки [] для указания диапазона и набора допустимых символов, например [0-9A-Z.-,]. Напоминает упрощенный синтаксис регулярных выражений, хотя это не так.

Если имя таблицы или столбца содержит пробелы, его необходимо заключать в квадратные скобки, например [Order Details]. SQL-сценарии для создания БД Northwind содержат команду set quoted_identifier on, поэтому допустимы и двойные кавычки, например "Order Details". Одинарные кавычки используются для литерального текста, например 'USA'.

Язык манипулирования данными (DML)

Язык манипулирования (управления) данными (Data Manipulation Language, *DML*) используется для запросов и изменения данных.

В этом языке наиболее востребован оператор SELECT, используемый для извлечения данных из одной или нескольких таблиц. Оператор SELECT чрезвычайно сложен, потому что он очень мощный. Эта книга не нацелена на обучение языку T-SQL, поэтому для формирования представления об операторе SELECT я решил привести несколько примеров, показанных в табл. 2.3.

Таблица 2.3. Примеры операторов SELECT с описанием

Пример	Описание
`SELECT *` `FROM Employees`	Извлекает все столбцы из таблицы Employees
`SELECT FirstName, LastName` `FROM Employees`	Извлекает столбцы с именами и фамилиями всех сотрудников из таблицы Employees
`SELECT emp.FirstName, emp.LastName` `FROM Employees AS emp`	Предоставляет псевдоним для имени таблицы. Если используется только одна таблица, префиксы имен таблиц не нужны. А если в нескольких таблицах есть столбцы с одинаковыми именами, например Customers.CustomerId и Orders.CustomerId, префиксы будут весьма кстати

Пример	Описание
`SELECT emp.FirstName, emp.LastName` `FROM Employees emp`	Предоставляет псевдоним имени таблицы без ключевого слова `AS`
`SELECT FirstName, LastName AS Surname` `FROM Employees`	Предоставляет псевдоним для имени столбца

`SELECT FirstName, LastName`
`FROM Employees`
`WHERE Country = 'USA'`
	Отфильтровывает в результатах сотрудников в США

`SELECT DISTINCT Country` `FROM Employees`	Извлекает список стран, перечисленных в столбце `Country` таблицы `Employees`, без дубликатов
`SELECT UnitPrice * Quantity AS Subtotal` `FROM [Order Details]`	Подсчитывает промежуточный итог для каждой строки детализации заказа

`SELECT OrderId,`
`  SUM(UnitPrice * Quantity) AS Total`
`FROM [Order Details]`
`GROUP BY OrderId`
`ORDER BY Total DESC`
	Подсчитывает общую сумму по каждому заказу и отсортировывает по уменьшению суммы заказа

`SELECT CompanyName`
`FROM Customers`
`UNION`
`SELECT CompanyName`
`FROM Suppliers`
	Возвращает все названия компаний всех клиентов и поставщиков

`SELECT CategoryName, ProductName` `FROM Categories, Products`	Сопоставляет *все* категории со *всеми* товарами путем декартова произведения (перекрестного соединения) и выводит их названия (обычно не требуется!). 616 строк (8 категорий × 77 товаров)

`SELECT CategoryName, ProductName`
`FROM Categories c, Products p`
`WHERE c.CategoryId = p.CategoryId`
	Сопоставляет *все* товары с соответствующими категориями посредством оператора `WHERE` для столбца `CategoryId` в каждой таблице и выводит названия категорий и товаров.
	77 строк

`SELECT CategoryName, ProductName`
`FROM Categories c`
`INNER JOIN Products p`
`ON c.CategoryId = p.CategoryId`
	Сопоставляет *все* товары с соответствующими категориями с помощью оператора `INNER JOIN...ON` для столбца `CategoryId` в каждой таблице и выводит названия категорий и товаров. Это современный синтаксис, альтернативный оператору `WHERE`, допускающий операции внешнего соединения. Результаты будут включать несовпадающие данные.
	77 строк

В копилку знаний: полную документацию об операторе SELECT ищите по адресу https://learn.microsoft.com/ru-ru/sql/t-sql/queries/select-transact-sql.

Откройте инструмент для отправки запросов к БД, например панель Server Explorer (Обозреватель серверов) в программе Visual Studio или расширение `mssql` в Visual Studio Code. Подключитесь к БД Northwind и отправьте несколько запросов (рис. 2.11, 2.12).

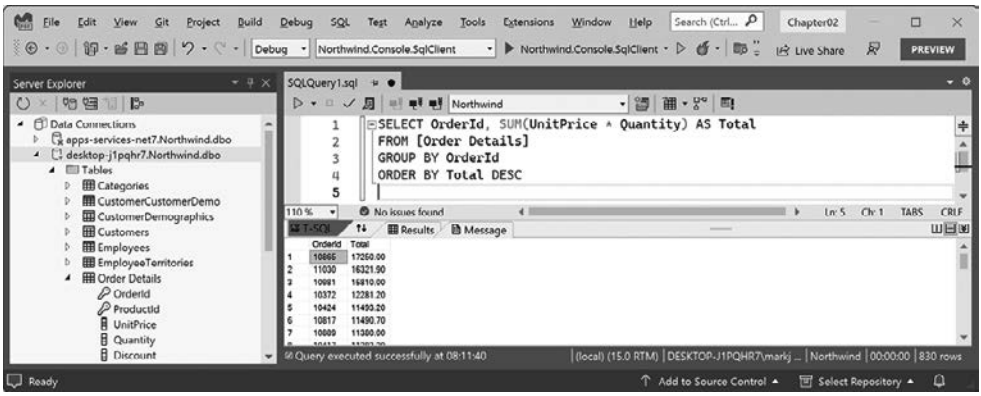


Рис. 2.11. Выполнение запросов T-SQL с помощью панели Server Explorer в Visual Studio

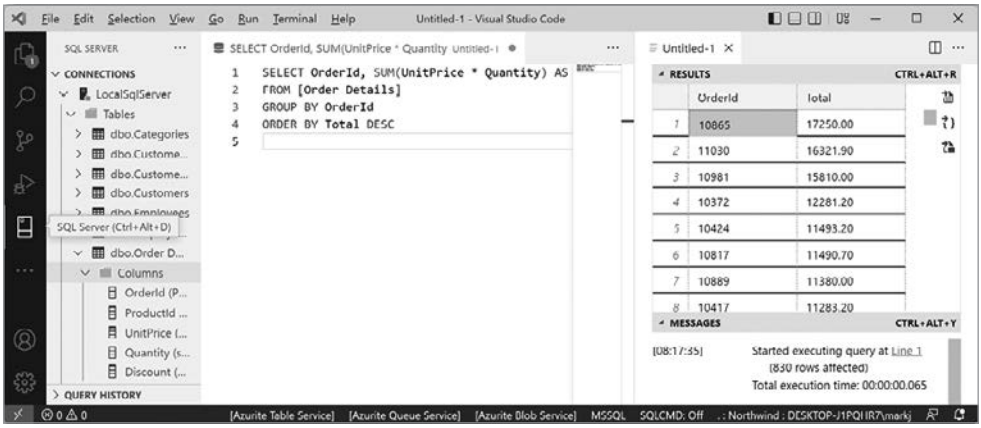


Рис. 2.12. Выполнение запросов T-SQL с помощью расширения mssql в Visual Studio Code

DML для добавления, обновления и удаления данных

DML-операторы для добавления, обновления и удаления данных перечислены в табл. 2.4.

Таблица 2.4. Примеры DML-операторов с описанием

Пример	Описание
`INSERT Employees(FirstName, LastName)` `VALUES('Mark', 'Price')`	Добавляет новую строку в таблицу `Employees`. Значение первичного ключа `EmployeeId` присваивается автоматически. Для извлечения этого значения используется переменная `@@IDENTITY`

`UPDATE Employees`
`SET Country = 'UK'`
`WHERE FirstName = 'Mark'`
`AND LastName = 'Price'`
	Изменяет строки таблицы `Employees`, присваивая полю `Country` значение `UK`

`DELETE Employees`
`WHERE FirstName = 'Mark'`
`AND LastName = 'Price'`
	Удаляет из таблицы `Employees` строку с указанным сотрудником

`DELETE Employees`	Удаляет из таблицы `Employees` все строки и вносит соответствующую запись в журнал транзакций
`TRUNCATE TABLE Employees`	Эффективнее удаляет из таблицы `Employees` все строки, поскольку не регистрирует удаление отдельных строк

В приведенных выше примерах используется таблица Employees в БД Northwind. У этой таблицы есть ограничения ссылочной целостности, поэтому, к примеру, удаление всех строк из таблицы невозможно, так как у нее есть связанные данные в других таблицах, в частности Orders.

Язык описания данных (DDL)

Операторы *языка описания данных* (Data Definition Language, *DDL*) изменяют структуру БД, в том числе отвечают за создание новых объектов, таких как таблицы, функции и хранимые процедуры. В рамках знакомства примеры DDL-операторов приведены в табл. 2.5. Обратите внимание, что эти простые примеры нельзя выполнить в БД Northwind.

Таблица 2.5. Примеры DDL-операторов с описанием

Пример	Описание

`CREATE TABLE dbo.Shippers (`
`  ShipperId INT PRIMARY KEY CLUSTERED,`
`  CompanyName NVARCHAR(40)`
`);`
	Создает таблицу `Shippers`

`ALTER TABLE Shippers` `ADD Country NVARCHAR(40)`	Добавляет в таблицу столбец
`CREATE NONCLUSTERED INDEX IX_Country` `ON Shippers(Country)`	Добавляет к столбцу в таблице некластеризованный индекс

`CREATE INDEX IX_FullName`
`ON Employees(LastName, FirstName DESC)`
`WITH (DROP_EXISTING = ON)`
	Изменяет агрегированный индекс нескольких столбцов и настраивает способ сортировки

`DROP TABLE Employees`	Удаляет таблицу `Employees`. Если она не существует, возникнет ошибка
`DROP TABLE IF EXISTS Employees`	Удаляет таблицу `Employees`, если та существует. Эта команда позволяет избежать потенциальной ошибки, возникающей с несуществующими таблицами при использовании оператора из предыдущего примера
`IF OBJECT_ID(N'Employees', N'U')` `  IS NOT NULL`	Проверяет наличие таблицы. Префикс `N` перед литеральным значением означает Юникод. `'U'` указывает на пользовательскую таблицу, а не системную

Управление данными с помощью низкоуровневых API

Пакет `Microsoft.Data.SqlClient` обеспечивает приложениям .NET подключение к БД SQL Server. Он также известен как *драйвер ADO.NET* для SQL Server и Azure SQL Database.

В копилку знаний: GitHub-репозиторий ADO.NET находится по адресу https://github.com/dotnet/SqlClient.

Пакет `Microsoft.Data.SqlClient` поддерживает следующие платформы .NET:

- .NET Framework 4.6.2 и более поздние версии;
- .NET Core 3.1 и более поздние версии;
- .NET Standard 2.0 и более поздние версии.

Типы данных в ADO.NET

Набор классов ADO.NET определяет абстрактные типы, предоставляющие минимальные объекты для управления данными, такие как `DbConnection`, `DbCommand` и `DbDataReader`. Разработчики ПО для манипулирования данными могут наследовать от них конкретные реализации, оптимизированные под выпускаемые ими БД и расширяющие их функционал. Такой подход в компании Microsoft применили при выпуске системы SQL Server. Важнейшие типы с наиболее используемыми членами приведены в табл. 2.6.

Таблица 2.6. Важные типы в ADO.NET SqlClient

Тип	Свойства	Методы	Описание
`SqlConnection`	`ConnectionString`, `State`, `ServerVersion`	`Open`, `Close`, `CreateCommand`, `RetrieveStatistics`	Управляет подключением к БД
`SqlConnection-StringBuilder`	`InitialCatalog`, `DataSource`, `Encrypt`, `UserID`, `Password`, `ConnectTimeout` и т. д.	`Clear`, `ContainsKey`, `Remove`	Формулирует корректную строку подключения к БД SQL Server. После настройки пользовательских параметров извлекает свойство `ConnectionString`
`SqlCommand`	`Connection`, `CommandType`, `CommandText`, `Parameters`, `Transaction`	`ExecuteReader`, `ExecuteNonQuery`, `ExecuteXmlReader`, `CreateParameter`	Настраивает команды для выполнения
`SqlParameter`	`ParameterName`, `Value`, `DbType`, `SqlValue`, `SqlDbType`, `Direction`, `IsNullable`		Настраивает параметры команды
`SqlDataReader`	`FieldCount`, `HasRows`, `IsClosed`, `RecordsAffected`	`Read`, `Close`, `GetOrdinal`, `GetInt32`, `GetString`, `GetDecimal`, `GetFieldValue<T>`	Обрабатывает результаты выполнения запроса

У типа `SqlConnection` есть два полезных события: `StateChange` и `InfoMessage`.

Все методы `ExecuteXxx` типа `SqlCommand` выполняют команды. Выбор метода зависит от желаемого результата.

- Если команда содержит один или более операторов `SELECT`, возвращающих результаты, то для выполнения команды вызовите метод `ExecuteReader`.

Он возвращает производный объект `DbDataReader` для построчного считывания результатов.

- Если команда не содержит операторы `SELECT`, то эффективнее вызвать метод `ExecuteNonQuery`. Он возвращает целочисленное количество затронутых строк.
- Если команда содержит один или несколько операторов `SELECT` с командой `AS XML`, возвращающих XML-данные, вызовите метод `ExecuteXmlReader`.

Создание консольного приложения для работы с ADO.NET

Сначала мы создадим консольное приложение для работы с ADO.NET.

1. В редакторе кода создайте приложение со следующими настройками:
   - выберите шаблон проекта Console App (Консольное приложение);
   - в поле Solution name (Имя решения) укажите значение `Chapter02`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Console.SqlClient`;
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
   - снимите флажок Enable native AOT publishing (Включить публикацию Native AOT).

Дельный совет: все проекты из этой книги храните в папках с коротким файловым путем и не используйте в их именах символ #. В противном случае могут возникать ошибки компилятора, такие как RSG002: TargetPath not specified for additional file. Такое происходит, к примеру, с проектами в папке вида C:\My C# projects\!

2. В файл проекта добавьте строгую обработку предупреждений и подключите пакет `Microsoft.Data.SqlClient` последней версии. Статически импортируйте класс `System.Console` на глобальном уровне:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>
```

```
  <ItemGroup>
    <PackageReference Include="Microsoft.Data.SqlClient" Version="5.1.2" />
  </ItemGroup>

  <ItemGroup>
    <Using Include="System.Console" Static="true" />
  </ItemGroup>

</Project>
```

Узнать актуальную версию пакета Microsoft.Data.SqlClient можно по адресу https://www.nuget.org/packages/Microsoft.Data.SqlClient#versions-body-tab.

3. Выполните сборку проекта для восстановления зависимостей.
4. Создайте файл класса `Program.Helpers.cs`. Определите в нем метод для настройки консоли, добавляющий поддержку специальных символов, например евро, и устанавливающий текущую культуру, а также метод для вывода в консоль текста с указанным цветом (по умолчанию используется черный цвет):

```
using System.Globalization; // Для класса CultureInfo

partial class Program
{
  private static void ConfigureConsole(string culture = "en-US",
    bool useComputerCulture = false)
  {
      // Для включения в консоли символов Юникода, например символа евро
    OutputEncoding = System.Text.Encoding.UTF8;

    if (!useComputerCulture)
    {
      CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo(culture);
    }
    WriteLine($"CurrentCulture: {CultureInfo.CurrentCulture.DisplayName}");
  }

  private static void WriteLineInColor(string value,
    ConsoleColor color = ConsoleColor.White)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = color;
    WriteLine(value);
    ForegroundColor = previousColor;
  }
}
```

В показанном коде текст окрашивается в белый цвет, так как на большинстве компьютеров установлен черный цвет фона. На своем компьютере в консоли я настроил белый цвет фона для удобства создания снимков экрана для книги. Выбирайте любые основной и фоновый цвета на своем компьютере.

5. Создайте файл класса `Program.EventHandlers.cs` и определите в нем методы, обрабатывающие события при изменении состояния подключения к БД. Они должны отражать исходное и текущее состояния, а также обрабатывать сообщение `InfoMessage` БД:

```
using Microsoft.Data.SqlClient; // Для класса SqlInfoMessageEventArgs
using System.Data; // Для объекта StateChangeEventArgs

partial class Program
{
  private static void Connection_StateChange(
    object sender, StateChangeEventArgs e)
  {
    WriteLineInColor(
      $"State change from {e.OriginalState} to {e.CurrentState}.",
      ConsoleColor.DarkYellow);
  }

  private static void Connection_InfoMessage(
    object sender, SqlInfoMessageEventArgs e)
  {
    WriteLineInColor($"Info: {e.Message}.", ConsoleColor.DarkBlue);
  }
}
```

6. В файле `Program.cs` замените существующие операторы кодом для локального подключения к SQL Server, Azure SQL Database или SQL Edge. Применяйте проверку подлинности либо SQL с идентификатором пользователя и паролем, либо Windows без ввода пользовательских данных:

```
using Microsoft.Data.SqlClient; // Для объекта SqlConnection и т. д.

ConfigureConsole();

#region Set up the connection string builder

SqlConnectionStringBuilder builder = new()
{
  InitialCatalog = "Northwind",
  MultipleActiveResultSets = true,
  Encrypt = true,
  TrustServerCertificate = true,
```

```
  ConnectTimeout = 10 // По умолчанию 30 секунд
};

WriteLine("Connect to:");
WriteLine(" 1 - SQL Server on local machine");
WriteLine(" 2 - Azure SQL Database");
WriteLine(" 3 – Azure SQL Edge");
WriteLine();
Write("Press a key: ");

ConsoleKey key = ReadKey().Key;
WriteLine(); WriteLine();

switch (key)
{
  case ConsoleKey.D1 or ConsoleKey.NumPad1:
    builder.DataSource = ".";
    break;
  case ConsoleKey.D2 or ConsoleKey.NumPad2:
    builder.DataSource =
      // Укажите имя своего сервера Azure SQL Database
      "tcp:apps-services-book.database.windows.net,1433";
    break;
  case ConsoleKey.D3 or ConsoleKey.NumPad3:
    builder.DataSource = "tcp:127.0.0.1,1433";
    break;
  default:
    WriteLine("No data source selected.");
    return;
}

WriteLine("Authenticate using:");
WriteLine(" 1 – Windows Integrated Security");
WriteLine(" 2 – SQL Login, for example, sa");
WriteLine();
Write("Press a key: ");
key = ReadKey().Key;
WriteLine(); WriteLine();

if (key is ConsoleKey.D1 or ConsoleKey.NumPad1)
{
  builder.IntegratedSecurity = true;
}
else if (key is ConsoleKey.D2 or ConsoleKey.NumPad2)
{
  Write("Enter your SQL Server user ID: ");
  string? userId = ReadLine();
  if (string.IsNullOrWhiteSpace(userId))
```

```
    {
      WriteLine("User ID cannot be empty or null.");
      return;
    }

    builder.UserID = userId;

    Write("Enter your SQL Server password: ");
    string? password = ReadLine();
    if (string.IsNullOrWhiteSpace(password))
    {
      WriteLine("Password cannot be empty or null.");
      return;
    }

    builder.Password = password;
    builder.PersistSecurityInfo = false;
  }
  else
  {
    WriteLine("No authentication selected.");
    return;
  }

  #endregion

  #region Create and open the connection

  SqlConnection connection = new(builder.ConnectionString);

  WriteLine(connection.ConnectionString);
  WriteLine();

  connection.StateChange += Connection_StateChange;
  connection.InfoMessage += Connection_InfoMessage;

  try
  {
    WriteLine("Opening connection. Please wait up to {0} seconds...",
      builder.ConnectTimeout);
    WriteLine();
    connection.Open();

    WriteLine($"SQL Server version: {connection.ServerVersion}");
  }
  catch (SqlException ex)
  {
    WriteLineInColor($"SQL exception: {ex.Message}",
```

```
    ConsoleColor.Red);
  return;
}

#endregion

connection.Close();
```

Дельный совет: в этой задаче по программированию пользователю выводится запрос на ввод пароля для подключения к БД. В рабочем приложении вы, скорее всего, будете хранить пароль в переменной окружения или в безопасном хранилище типа Azure Key Vault. Совершенно недопустимо хранить пароли в исходном коде!

7. Запустите консольное приложение, выберите параметры, подходящие для вашей установки SQL Server, и проанализируйте полученные результаты, в том числе оповещения об изменении состояния:

```
Connect to:
  1 - SQL Server on local machine
  2 - Azure SQL Database
  3 - Azure SQL Edge

Press a key: 1

Authenticate using:
  1 - Windows Integrated Security
  2 - SQL Login, for example, sa

Press a key: 1

Data Source=.;Initial Catalog=Northwind;Integrated Security=True;Multiple
Active Result Sets=True;Connect Timeout=10;Encrypt=True;Trust Server
Certificate=True

Opening connection. Please wait up to 10 seconds...

State change from Closed to Open.
SQL Server version: 15.00.2101
State change from Open to Closed.
```

Далее демонстрируется процесс подключения к Azure SQL Database/Azure SQL Edge, где требуется ввести имя пользователя и пароль. Если вы подключаетесь к локальному экземпляру SQL Server через систему проверки подлинности Windows, пароль вводить не нужно.

8. Запустите консольное приложение, выберите вариант, требующий ввода идентификатора пользователя и пароля, например Azure SQL Database, и обратите внимание на результат:

```
Enter your SQL Server user ID: markjprice
Enter your SQL Server password: [censored]
Data Source=tcp:apps-services-book.database.windows.
net,1433;Initial Catalog=Northwind;Persist Security Info=False;User
ID=markjprice;Password=[censored];Multiple Active Result
Sets=True;Connect Timeout=10;Encrypt=True;Trust Server Certificate=True

Opening connection. Please wait up to 10 seconds...

State change from Closed to Open.
SQL Server version: 12.00.5168
State change from Open to Closed.
```

9. Запустите консольное приложение, выберите вариант, требующий ввода идентификатора пользователя и пароля, введите неверные данные и запишите результат:

```
Enter your SQL Server user ID: markjprice
Enter your SQL Server password: 123456
Data Source=tcp:apps-services-book.database.windows.
net,1433;Initial Catalog=Northwind;Persist Security Info=False;User
ID=markjprice;Password=123456;Multiple Active Result Sets=True;Connect
Timeout=10;Encrypt=True;Trust Server Certificate=True

Opening connection. Please wait up to 10 seconds...

SQL exception: Login failed for user 'markjprice'.
```

10. В файле `Program.cs` измените имя сервера (свойство `DataSource`) на неправильное.
11. Запустите консольное приложение и обратите внимание на результат (в зависимости от локации вашей БД текст исключения может слегка отличаться):

```
SQL exception: A network-related or instance-specific error
occurred while establishing a connection to SQL Server. The server
was not found or was not accessible. Verify that the instance
name is correct and that SQL Server is configured to allow
remote connections. (provider: TCP Provider, error: 0 - No such
host is known.)
```

При установлении соединения с SQL Server тайм-аут на случай проблем с подключением к серверу составляет по умолчанию 30 секунд, так что наберитесь терпения! Мы настроили тайм-аут равным 10 секундам, чтобы ждать не так долго.

Выполнение запросов и извлечение данных с помощью ADO.NET

Теперь, когда мы успешно подключились к БД SQL Server, можно извлекать строки из таблиц и обрабатывать результаты с помощью объекта `DataReader`.

1. В файле `Program.cs` импортируйте пространство имен для поддержки типов ADO.NET:

```
using System.Data; // Для свойства CommandType
```

Дельный совет: для краткости я буду использовать имена cmd и r для обозначения SQL-команд и SQL-объекта DataReader. В своих сценариях присваивайте переменным подходящие полноценные имена, например command и reader.

2. Перед оператором, закрывающим подключение, добавьте код для извлечения идентификатора, названия и цены единицы товара из таблицы `Products` и вывода этих данных с помощью объекта `DataReader`:

```
SqlCommand cmd = connection.CreateCommand();

cmd.CommandType = CommandType.Text;
cmd.CommandText = "SELECT ProductId, ProductName, UnitPrice FROM Products";

SqlDataReader r = cmd.ExecuteReader();

string horizontalLine = new string('-', 60);
WriteLine(horizontalLine);
WriteLine("| {0,5} | {1,-35} | {2,10} |",
  arg0: "Id", arg1: "Name", arg2: "Price");
WriteLine(horizontalLine);

while (r.Read())
{
  WriteLine("| {0,5} | {1,-35} | {2,10:C} |",
    r.GetInt32("ProductId"),
    r.GetString("ProductName"),
    r.GetDecimal("UnitPrice"));
}

WriteLine(horizontalLine);

r.Close();
```

Мы выводим цену единицы товара в формате C (Currency — денежная единица), учитывающем текущую культуру для форматирования денежных значений. Согласно вызову метода ConfigureConsole используется английская культура (США), поэтому в выводе программы у всех пользователей цена будет сопровождаться символом $. Для смены культуры, например, на Францию и вывода символа € измените вызов в начале файла Program.cs на ConfigureConsole("fr-FR");.

3. Запустите консольное приложение и обратите внимание на результат (ниже показан фрагмент):

```
-------------------------------------------------------
|   Id | Name                                |   Price |
-------------------------------------------------------
|    1 | Chai                                |  $18.00 |
|    2 | Chang                               |  $19.00 |
...
|   76 | Lakkalikööri                        |  $18.00 |
|   77 | Original Frankfurter grüne Soße     |  $13.00 |
-------------------------------------------------------
```

4. В файле `Program.cs` измените SQL-код, настроив вывод цены единицы товара и добавив возможность фильтрации товаров, цена которых превышает указанную:

```
Write("Enter a unit price: ");
string? priceText = ReadLine();

if(!decimal.TryParse(priceText, out decimal price))
{
  WriteLine("You must enter a valid unit price.");
  return;
}

SqlCommand cmd = connection.CreateCommand();

cmd.CommandType = CommandType.Text;

cmd.CommandText = "SELECT ProductId, ProductName, UnitPrice FROM Products"
  + " WHERE UnitPrice >= @minimumPrice";

cmd.Parameters.AddWithValue("minimumPrice", price);
```

5. Запустите консольное приложение, введите цену единицы товара, например `50`, и обратите внимание на результат (ниже показан фрагмент):

```
Enter a unit price: 50
-------------------------------------------------------
|   Id | Name                                |   Price |
-------------------------------------------------------
|    9 | Mishi Kobe Niku                     |  $97.00 |
|   18 | Carnarvon Tigers                    |  $62.50 |
|   20 | Sir Rodney's Marmalade              |  $81.00 |
|   29 | Thüringer Rostbratwurst             | $123.79 |
|   38 | Côte de Blaye                       | $263.50 |
|   51 | Manjimup Dried Apples               |  $53.00 |
|   59 | Raclette Courdavault                |  $55.00 |
-------------------------------------------------------
```

Вывод статистики

ADO.NET позволяет отслеживать существующие полезные статистические данные, в том числе те, что перечислены в табл. 2.7.

Таблица 2.7. Отслеживаемая статистика подключения

Ключ	Описание
`BuffersReceived`, `BuffersSent`, `BytesReceived`, `BytesSent`	Данные передаются в виде байтов, хранящихся в буферах
`CursorOpens`	Открытие динамических курсоров — это затратная процедура, поскольку требует дополнительных сетевых и программных ресурсов на сервере, чего по возможности следует избегать
`Prepares`, `PreparedExecs`, `UnpreparedExecs`	Количество подготовок (компиляций), выполненных подготовленных и неподготовленных команд
`SelectCount`, `SelectRows`	Количество операторов `SELECT` и строк, возвращаемых ими
`ServerRoundtrips`, `SumResultSets`, `Transactions`	Количество обходов сервера, наборов результатов и транзакций
`ConnectionTime`, `ExecutionTime`, `NetworkServerTime`	Время в миллисекундах, затраченное на подключение, выполнение команд или сетевые операции

Опробуем эту функцию и выведем некоторые статистические данные.

1. В файле `Program.Helpers.cs` импортируйте пространство имен для поддержки ADO.NET и коллекций общего назначения:

```
using Microsoft.Data.SqlClient; // Для объекта SqlConnection
using System.Collections;       // Для интерфейса IDictionary
```

2. В файле `Program.Helpers.cs`, в разделяемом классе `Program`, добавьте метод для вывода статистики о текущем подключении. Она должна быть представлена массивом строковых значений для выбора желаемых статистических данных:

```
private static void OutputStatistics(SqlConnection connection)
{
  // Удаление строковых значений для вывода полной статистики
  string[] includeKeys = {
    "BytesSent", "BytesReceived", "ConnectionTime", "SelectRows"
  };

  IDictionary statistics = connection.RetrieveStatistics();
```

```
    foreach (object? key in statistics.Keys)
    {
      if (!includeKeys.Any() || includeKeys.Contains(key))
      {
        if (int.TryParse(statistics[key]?.ToString(), out int value))
        {
          WriteLineInColor($"{key}: {value:N0}", ConsoleColor.Cyan);
        }
      }
    }
  }
```

3. В файле `Program.cs`, после оператора вывода версии SQL Server в консоль, добавьте оператор для включения сбора статистики о подключении:

```
WriteLine($"SQL Server version: {connection.ServerVersion}");

connection.StatisticsEnabled = true;
```

4. В файле `Program.cs` перед командой закрытия подключения добавьте оператор для вывода статистики о подключении:

```
OutputStatistics(connection);

connection.Close();
```

5. Запустите консольное приложение и обратите внимание на результат со статистическими данными (ниже показан фрагмент):

```
BytesReceived: 3,888
BytesSent: 336
SelectRows: 77
ExecutionTime: 25
```

Асинхронная работа с ADO.NET

Вы можете ускорить доступ к данным, сделав код асинхронным. Более подробно о том, как работают асинхронные операции, вы узнаете в главе 5. А пока введите показанный ниже код в соответствии с шагами.

Посмотрим, как изменить операторы, чтобы они работали асинхронно.

1. В файле `Program.cs` превратите оператор открытия подключения в асинхронный:

```
await connection.OpenAsync();.
```

2. В файле `Program.cs` превратите оператор выполнения команды в асинхронный:

```
SqlDataReader r = await cmd.ExecuteReaderAsync();.
```

3. В файле `Program.cs` превратите операторы считывания следующей строки и извлечения значений полей в асинхронные:

```
while (await r.ReadAsync())
{
  WriteLine("| {0,5} | {1,-35} | {2,8:C} |",
    await r.GetFieldValueAsync<int>("ProductId"),
    await r.GetFieldValueAsync<string>("ProductName"),
    await r.GetFieldValueAsync<decimal>("UnitPrice"));
}
```

4. В файле `Program.cs` превратите операторы закрытия объекта `DataReader` и подключения в асинхронные:

```
await r.CloseAsync();
await connection.CloseAsync();
```

5. Запустите консольное приложение и убедитесь, что оно приводит к тому же результату, что и раньше, но лучше работает в многопоточной системе, в частности не блокирует пользовательский интерфейс GUI-приложения и потоки ввода-вывода на сайте.

Выполнение хранимых процедур с помощью ADO.NET

Если вам требуется многократно отправлять одни и те же запросы или выполнять SQL-оператор несколько раз, можно создавать *хранимые процедуры*, в том числе параметризированные, предварительно скомпилированные и оптимизированные. У параметров хранимых процедур есть направление, определяющее тип значения: входное, выходное или возвращаемое.

Рассмотрим пример с параметрами всех трех направлений. Сначала создадим хранимую процедуру в БД.

1. В выбранной программе подключитесь к БД Northwind.
2. Далее создайте хранимую процедуру.
   - В SQL Server Management Studio: на панели Object Explorer (Обозреватель объектов) перейдите в раздел Databases ▸ Northwind ▸ Programmability (Базы данных ▸ Northwind ▸ Программируемая), щелкните правой кнопкой мыши на пункте Stored Procedures (Хранимые процедуры) и в контекстном меню выберите пункт New ▸ Stored Procedure (Новая ▸ Хранимая процедура).
   - В Visual Studio 2022: на панели Server Explorer (Обозреватель серверов) щелкните правой кнопкой мыши на пункте Stored Procedures (Хранимые процедуры) и выберите пункт Add New Stored Procedure (Добавить новую хранимую процедуру).

- В Visual Studio Code на панели SQL Server щелкните правой кнопкой мыши на профиле подключения и в контекстном меню выберите пункт New Query (Новый запрос).
- В JetBrains Rider на панели инструментов Database (БД) нажмите кнопку Jump to Query Console (Перейти в консоль запроса), а затем удалите имеющиеся операторы. Введите показанные ниже SQL-операторы, а также команду активации БД на Northwind: `USE Northwind GO`. Таким образом мы предотвратим создание хранимой процедуры в основной (`master`) БД!

3. С помощью SQL-операторов определите хранимую процедуру `GetExpensiveProducts` с двумя параметрами: входным — для минимальной цены единицы товара и выходным — для количества соответствующих товаров:

```
CREATE PROCEDURE [dbo].[GetExpensiveProducts]
  @price money,
  @count int OUT
AS
  PRINT 'Getting expensive products: ' +
    TRIM(CAST(@price AS NVARCHAR(10)))

  SELECT @count = COUNT(*)
  FROM Products
    WHERE UnitPrice >= @price

  SELECT *
  FROM Products
    WHERE UnitPrice >= @price

RETURN 0
```

Хранимая процедура включает два оператора SELECT. Первый присваивает выходному параметру @count значение счетчика совпадающих строк товара. Второй возвращает совпадающие строки товара. Она также содержит команду PRINT, инициирующую событие InfoMessage.

4. Щелкните правой кнопкой мыши на SQL-операторах и выберите в контекстном меню пункт Execute (Выполнить) или Execute Query (Выполнить запрос).
5. Щелкните правой кнопкой мыши на строке Stored Procedures (Хранимые процедуры) и выберите команду Refresh (Обновить). В JetBrains Rider этот пункт называется Routines (Сопрограммы).
6. Разверните код процедуры `GetExpensiveProducts` и обратите внимание на входной параметр `@price money`, входной/выходной параметр `@count int` и возвращаемое значение, как показано в окне программы SQL Server Management Studio на рис. 2.13.

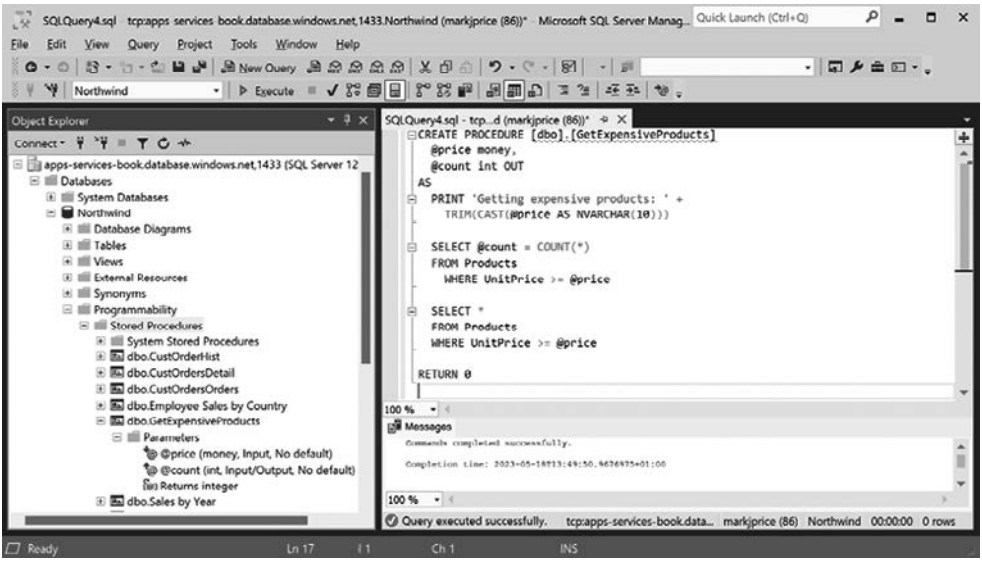


Рис. 2.13. Параметры хранимой процедуры GetExpensiveProducts

7. Закройте SQL-запрос без сохранения изменений.
8. В файле `Program.cs` добавьте операторы, предоставляющие пользователю выбор между запуском текстовой команды и хранимой процедуры. Добавьте операторы, определяющие хранимую процедуру и ее параметры, а затем выполните команду:

```
SqlCommand cmd = connection.CreateCommand();

WriteLine("Execute command using:");
WriteLine(" 1 - Text");
WriteLine(" 2 - Stored Procedure");
WriteLine();
Write("Press a key: ");

key = ReadKey().Key;
WriteLine(); WriteLine();

SqlParameter p1, p2 = new(), p3 = new();

if (key is ConsoleKey.D1 or ConsoleKey.NumPad1)
{
  cmd.CommandType = CommandType.Text;
  cmd.CommandText = "SELECT ProductId, ProductName, UnitPrice FROM Products"
    + " WHERE UnitPrice >= @minimumPrice";
```

```
    cmd.Parameters.AddWithValue("minimumPrice", price);
  }
  else if (key is ConsoleKey.D2 or ConsoleKey.NumPad2)
  {
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.CommandText = "GetExpensiveProducts";

    p1 = new()
    {
      ParameterName = "price",
      SqlDbType = SqlDbType.Money,
      SqlValue = price
    };

    p2 = new()
    {
      Direction = ParameterDirection.Output,
      ParameterName = "count",
      SqlDbType = SqlDbType.Int
    };

    p3 = new()
    {
      Direction= ParameterDirection.ReturnValue,
      ParameterName = "rv",
      SqlDbType = SqlDbType.Int
    };

    cmd.Parameters.AddRange(new[] { p1, p2, p3 });
  }

  SqlDataReader r = await cmd.ExecuteReaderAsync();
```

9. После оператора, закрывающего объект `DataReader`, добавьте команды для вывода выходного параметра и возвращаемого значения:

```
  await r.CloseAsync();

  if (key is ConsoleKey.D2 or ConsoleKey.NumPad2)
  {
    WriteLine($"Output count: {p2.Value}");
    WriteLine($"Return value: {p3.Value}");
  }

  await connection.CloseAsync();
```

Если хранимая процедура возвращает не только параметры, но и наборы результатов, то перед чтением параметров объект DataReader необходимо закрыть.

10. Запустите консольное приложение и обратите внимание на результаты. Ниже показан пример с введенной ценой. Обработчик события `InfoMessage` выведет текст темно-синим цветом.

```
Enter a unit price: 60
Execute command using:
  1 - Text
  2 - Stored Procedure

Press a key: 2

Info: Getting expensive products: 60.00.
-------------------------------------------------------
|    Id | Name                                |   Price |
-------------------------------------------------------
|     9 | Mishi Kobe Niku                     |  $97.00 |
|    18 | Carnarvon Tigers                    |  $62.50 |
|    20 | Sir Rodney's Marmalade              |  $81.00 |
|    29 | Thüringer Rostbratwurst             | $123.79 |
|    38 | Côte de Blaye                       | $263.50 |
-------------------------------------------------------
Output count: 5
Return value: 0
State change from Open to Closed.
```

Вывод потоков с помощью объекта DataReader

В рабочем приложении/сервисе вы, скорее всего, не будете выводить данные в консоль. Вероятно считывая строки по очереди с помощью объекта `DataReader`, вы будете передавать данные в поток, генерирующий из них HTML-код на веб-странице, или текстовые форматы, такие как XML и JSON.

Добавим возможность генерировать JSON-файл.

1. В файле `Program.cs` импортируйте пространство имен для эффективной работы с JSON и статически импортируйте классы `Environment` и `Path`:

```
using System.Text.Json; // Для метода Utf8JsonWriter класса JsonSerializer

using static System.Environment;
using static System.IO.Path;
```

2. В файле `Program.cs` перед конструкцией `while`, обрабатывающей объект `DataReader`, добавьте операторы для определения пути к JSON-файлу, создания файлового потока и генерации массива JSON. Затем в конструкции `while` напишите код JSON-объекта, представляющий каждую строку товара, и, наконец, завершите массив и закройте поток:

```
// Определение пути к файлу, в который будет производиться запись
string jsonPath = Combine(CurrentDirectory, "products.json");
```

```
await using (FileStream jsonStream = File.Create(jsonPath))
{
  Utf8JsonWriter jsonWriter = new(jsonStream);
  jsonWriter.WriteStartArray();

  while (await r.ReadAsync())
  {
    WriteLine("| {0,5} | {1,-35} | {2,10:C} |",
      await r.GetFieldValueAsync<int>("ProductId"),
      await r.GetFieldValueAsync<string>("ProductName"),
      await r.GetFieldValueAsync<decimal>("UnitPrice"));

    jsonWriter.WriteStartObject();

    jsonWriter.WriteNumber("productId",
      await r.GetFieldValueAsync<int>("ProductId"));
    jsonWriter.WriteString("productName",
      await r.GetFieldValueAsync<string>("ProductName"));
    jsonWriter.WriteNumber("unitPrice",
      await r.GetFieldValueAsync<decimal>("UnitPrice"));
    jsonWriter.WriteEndObject();
  }

  jsonWriter.WriteEndArray();
  jsonWriter.Flush();
  jsonStream.Close();
}

WriteLineInColor($"Written to: {jsonPath}", ConsoleColor.DarkGreen);
```

3. Запустите консольное приложение, введите цену единицы товара, равную `60`, и укажите путь к JSON-файлу:

```
Written to: C:\apps-services-net8\Chapter02\Northwind.Console.SqlClient\bin\
Debug\net8.0\products.json
```

4. Откройте файл `products.json`. Обратите внимание, что JSON-код пишется без пробелов в одной строке:

```
[{"productId":9,"productName":"Mishi Kobe Niku","unitPrice":97.0000},{
"productId":18,"productName":"Carnarvon Tigers","unitPrice":62.5000},
{"productId":20,"productName":"Sir Rodney\u0027s Marmalade","unitPrice":
81.0000},{"productId":29,"productName":"Th\u00FCringer Rostbratwurst",
"unitPrice":123.7900},{"productId":38,"productName":"C\u00F4te de
Blaye","unitPrice":263.5000}]
```

5. В программе Visual Studio 2022 можно щелкнуть правой кнопкой мыши и выбрать пункт Format Document (Форматирование документа). Теперь документ стал удобнее для чтения (рис. 2.14).

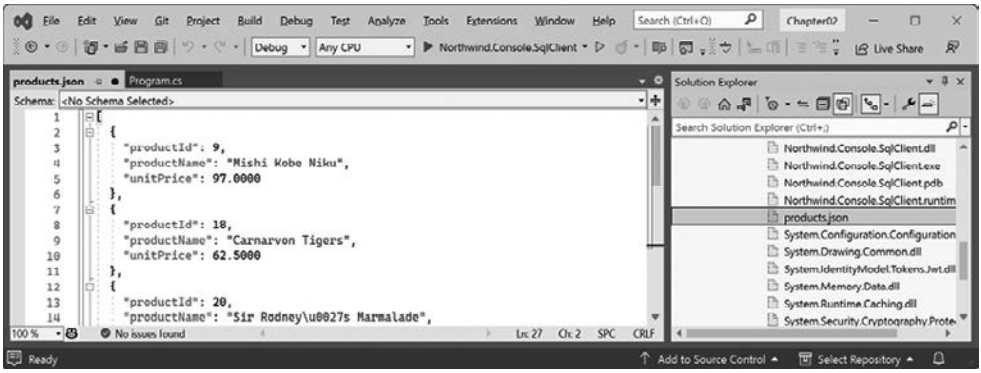


Рис. 2.14. Файл products.json, созданный с помощью объекта DataReader

Создание объектов с помощью объекта DataReader

Для максимальной гибкости мы можем преобразовать строки `DataReader` в экземпляры объектов, хранящиеся в массиве или коллекции. После этого граф объектов можно сериализовать любым удобным способом. В ADO.NET нет встроенной возможности сопоставить считанную строку данных с объектом, поэтому нам придется делать это вручную.

Рассмотрим пример.

1. Создайте файл класса `Product.cs` и измените его содержимое, добавив определение класса для представления трех столбцов, извлекаемых из каждой строки таблицы `Products`:

```
namespace Northwind.Models;

public class Product
{
  public int ProductId { get; set; }
  public string? ProductName { get; set; }
  public decimal? UnitPrice { get; set; }
}
```

Дельный совет: в этой задаче мы применяем описываемый тип к экземплярам, доступным исключительно для чтения, так что записи останутся неизменны. Так как позже нам потребуется изменять значения свойств после создания объекта, мы определяем класс.

2. В начале файла `Program.cs` импортируйте пространство имен `Northwind.Models`, чтобы использовать класс `Product`.

3. В файле `Program.cs` перед открытием файлового потока создайте список товаров с начальным хранилищем на 77 элементов (но это не предел), поскольку изначально БД Northwind содержит 77 товаров:

```
List<Product> products = new(capacity: 77);

await using (FileStream jsonStream = File.Create(jsonPath))
```

4. В конструкцию `while` добавьте операторы для создания экземпляра типа `Product` для каждой строки объекта `DataReader` и добавления его в список:

```
while (await r.ReadAsync())
{
  Product product = new()
  {
    ProductId = await r.GetFieldValueAsync<int>("ProductId"),
    ProductName = await r.GetFieldValueAsync<string>("ProductName"),
    UnitPrice = await r.GetFieldValueAsync<decimal>("UnitPrice")
  };
  products.Add(product);

  ...
}
```

5. Прежде чем закрыть объект `DataReader`, добавьте код статического метода `Serialize` класса `JsonSerializer` для вывода списка товаров в консоль:

```
WriteLineInColor(JsonSerializer.Serialize(products),
  ConsoleColor.Magenta);

await r.CloseAsync();
```

6. Запустите консольное приложение, введите цену единицы товара, например `60`, и обратите внимание на JSON-данные, сгенерированные из списка товаров:

```
Written to: C:\apps-services-net8\Chapter02\Northwind.Console.SqlClient\bin\
Debug\net8.0\products.json
[{"ProductId":9,"ProductName":"Mishi Kobe Niku","UnitPrice":97.0000},
{"ProductId":18,"ProductName":"Carnarvon Tigers","UnitPrice":62.5000},
{"ProductId":20,"ProductName":"Sir Rodney\u0027s Marmalade","UnitPrice":
81.0000},{"ProductId":29,"ProductName":"Th\u00FCringer
Rostbratwurst","UnitPrice":123.7900},{"ProductId":38,"ProductName":"C\u00F4te
de Blaye","UnitPrice":263.5000}]
```

Вместо создания экземпляров объектов вручную для еще большего упрощения мы можем использовать инструмент *объектно-реляционного отображения* (Object-Relational Mapping, ORM), например Dapper.

Управление данными с помощью Dapper

В ПО Dapper также используется технология ADO.NET при работе с SQL Server. Поскольку это технология более высокого уровня, этот способ не так функционален, как непосредственное использование ADO.NET, но зато отчасти проще. Dapper — это альтернативный инструмент объектно-реляционного отображения (ORM) для EF Core. Его преимущество в расширении низкоуровневого простейшего узкоспециализированного ADO.NET-интерфейса `IDbConnection`.

Расширение подключений с помощью Dapper

Dapper добавляет три метода расширения, `Query<T>`, `Query` и `Execute`, в классы, реализующие интерфейс `IDbConnection` (например, `SqlConnection`). По мере необходимости Dapper автоматически открывает и закрывает соответствующие подключения.

Метод расширения `Query<T>` наиболее востребован, поскольку он отвечает за выполнение всех SQL-команд и возвращает результаты в виде последовательности объектов `IEnumerable<T>`. Он предназначен для выполнения команд, извлекающих данные, например `SELECT`, и поддерживает несколько параметров, перечисленных в табл. 2.8.

Таблица 2.8. Параметры метода расширения Query<T> в Dapper

Параметр	Описание
`string sql`	Единственный обязательный параметр. Представляет собой либо SQL-команду, либо имя хранимой процедуры
`object param = null`	Сложный объект для передачи параметров, используемых в запросе. Это может быть анонимный тип
`IDbTransaction transaction = null`	Используется для управления распределенными транзакциями
`bool buffered = true`	По умолчанию буферизует все возвращаемые данные. При работе с большими наборами данных можно минимизировать затраты памяти и загружать объекты по мере необходимости, присвоив параметру `buffered` значение `false`
`int? commandTimeout = null`	Определяет длительность ожидания команды по умолчанию
`CommandType? commandType = null)`	Необходим для переключения на хранимую процедуру вместо используемой по умолчанию текстовой команды

Метод расширения `Query` — это слаботипизированный эквивалент, поэтому он используется реже.

Метод расширения `Execute` выполняет указанную SQL-команду и возвращает количество затронутых строк в виде целого числа. Предназначен для выполнения таких команд, как `INSERT`, `UPDATE` и `DELETE`. Поддерживает те же параметры, что и метод `Query<T>`.

Выполнение запросов с помощью Dapper

Рассмотрим простой пример, где вместо таблицы `Products` запрашивается таблица `Suppliers`.

1. В файле проекта `Northwind.Console.SqlClient` добавьте ссылку на пакет `Dapper`:

```
<ItemGroup>
  <PackageReference Include="Microsoft.Data.SqlClient" Version="5.1.2" />
  <PackageReference Include="Dapper" Version="2.1.21" />
</ItemGroup>
```

На момент подготовки книги актуальной была версия Dapper 2.1.66, выпущенная 6 февраля 2025 года. Проверить наличие свежей версии можно по адресу https://www.nuget.org/packages/Dapper.

2. Выполните сборку проекта для восстановления зависимостей.
3. Создайте файл `Supplier.cs` и определите в нем класс для представления четырех столбцов из каждой строки таблицы поставщиков (`Suppliers`):

```
namespace Northwind.Models;

public class Supplier
{
  public int SupplierId { get; set; }
  public string? CompanyName { get; set; }
  public string? City { get; set; }
  public string? Country { get; set; }
}
```

4. В конце файла `Program.cs` добавьте операторы для извлечения объектов поставщика в Германии (`Germany`), просмотра коллекции с выводом базовой информации о каждом из них, а затем сериализации коллекции в формате JSON в консоль:

```
WriteLineInColor("Using Dapper", ConsoleColor.DarkGreen);

connection.ResetStatistics(); // Таким образом можно выполнять
                              // сравнение с помощью Dapper
```

```
IEnumerable<Supplier> suppliers = connection.Query<Supplier>(
  sql: "SELECT * FROM Suppliers WHERE Country=@Country",
  param: new { Country = "Germany" });

foreach (Supplier s in suppliers)
{
  WriteLine("{0}: {1}, {2}, {3}",
    s.SupplierId, s.CompanyName, s.City, s.Country);
}

WriteLineInColor(JsonSerializer.Serialize(suppliers), ConsoleColor.Green);

OutputStatistics(connection);
```

5. Запустите консольное приложение и обратите внимание, что для выполнения Dapper-запроса использовалось одно и то же подключение. Во время него были инициированы соответствующие события, затем была выведена перечисляемая коллекция, а затем из списка поставщиков сгенерирован JSON-вывод:

```
Using Dapper
11: Heli Süßwaren GmbH & Co. KG, Berlin, Germany
12: Plutzer Lebensmittelgroßmärkte AG, Frankfurt, Germany
13: Nord-Ost-Fisch Handelsgesellschaft mbH, Cuxhaven, Germany
[{"SupplierId":11,
  "CompanyName":"Heli S\u00FC\u00DFwaren GmbH \u0026 Co. KG",
  "City":"Berlin","Country":"Germany"},
 {"SupplierId":12,
  "CompanyName":"Plutzer Lebensmittelgro\u00DFm\u00E4rkte AG",
  "City":"Frankfurt","Country":"Germany"},
 {"SupplierId":13,
  "CompanyName":"Nord-Ost-Fisch Handelsgesellschaft mbH",
  "City":"Cuxhaven","Country":"Germany"}]
BytesReceived: 1,430
BytesSent: 240
SelectRows: 3
ExecutionTime: 5
```

6. В конце файла `Program.cs` добавьте операторы для выполнения хранимой процедуры `GetExpensiveProducts` с передачей параметру `price` значения 100, перечисления коллекции с выводом основной информации о каждой из них, а затем сериализации коллекции в виде JSON-вывода в консоль:

```
IEnumerable<Product> productsFromDapper =
  connection.Query<Product>(sql: "GetExpensiveProducts",
  param: new { price = 100M, count = 0 },
  commandType: CommandType.StoredProcedure);

foreach (Product p in productsFromDapper)
{
  WriteLine("{0}: {1}, {2}",
```

```
    p.ProductId, p.ProductName, p.UnitPrice);
}

WriteLineInColor(JsonSerializer.Serialize(productsFromDapper),
  ConsoleColor.Green);
```

Внимание! В Dapper объект param необходимо передавать со всеми параметрами, даже если они используются только в качестве выходных. К примеру, необходимо определить параметр count, иначе будет выброшено исключение. Не забудьте также явно указать тип команды на хранимую процедуру!

7. Запустите консольное приложение и обратите внимание, что для выполнения Dapper-запроса с извлечением товаров ценой свыше `100` использовалось одно и то же подключение. Во время него были инициированы соответствующие события, затем была выведена перечисляемая коллекция, а затем из списка поставщиков сгенерирован JSON-вывод:

```
Info: Getting expensive products: 100.00.
29: Thüringer Rostbratwurst, 123.7900
38: Côte de Blaye, 263.5000
[{"ProductId":29,"ProductName":"Th\u00FCringer Rostbratwurst",
"UnitPrice":123.7900},{"ProductId":38,"ProductName":"C\u00F4te de
Blaye","UnitPrice":263.5000}]
```

В копилку знаний: дополнительную информацию о Dapper вы найдете по адресу https://github.com/DapperLib/Dapper/blob/main/Readme.md.

Удаление использовавшихся ресурсов

Завершив работу с БД SQL Server, необходимо удалить использовавшиеся ресурсы.

БД Northwind используется в большинстве глав книги, поэтому если вы планируете сразу же продолжить работу с другими главами после этой, не удаляйте Northwind! Если вы создали БД на своем локальном компьютере, то можете не трогать ее до окончания чтения книги.

Удаление ресурсов Azure

В целях экономии удалим ресурсы, использовавшиеся при подключении к SQL Database.

Внимание! Если вы не удалите ресурсы, требуемые БД Azure SQL, компания Microsoft выставит вам счет за их использование.

1. На портале Azure найдите группу ресурсов `apps-services-book`.
2. Нажмите кнопку Delete (Удалить).
3. Введите название группы ресурсов.
4. Нажмите кнопку Delete (Удалить).

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 2.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. На какой NuGet-пакет следует ссылаться в проекте .NET, чтобы добиться максимальной производительности при работе с БД SQL Server?
2. Какой способ конфигурации строки подключения к БД наиболее безопасен?
3. Что необходимо указывать в качестве префикса параметров и переменных T-SQL?
4. Что необходимо сделать перед считыванием выходного параметра команды, выполненной с помощью метода `ExecuteReader`?
5. К какому типу Dapper добавляет свои методы расширения?
6. Какие два наиболее часто используемых метода расширения предоставляет Dapper?

Упражнение 2.2. Дополнительная информация

Дополнительные материалы по темам этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-2---managing-relational-data-using-sql-server.

Упражнение 2.3. Альтернативные варианты хранения конфиденциальной информации

Такая уязвимая информация, как пароли и другие конфиденциальные данные, используемые в строках подключения к БД, например пароли для доступа к сервису, часто располагается в переменных окружения. В целях безопасности такие *секреты приложений* могут храниться в специальных защищенных хранилищах. Подробнее об этом вы можете узнать из статьи «Безопасное хранение секретов приложений во время разработки в ASP.NET Core», доступной по адресу https://learn.microsoft.com/ru-ru/aspnet/core/security/app-secrets.

О работе со строками подключения вы можете прочитать по адресу https://learn.microsoft.com/ru-ru/ef/core/miscellaneous/connection-strings.

Резюме

В этой главе вы узнали:

- как подключиться к существующей БД SQL Server;
- как выполнить простой запрос и обработать результаты с помощью быстрого и низкоуровневого инструмента ADO.NET;
- как выполнить простой запрос и обработать результаты с помощью Dapper.

В следующей главе вы узнаете, как использовать более мощный и сложный инструмент объектно-реляционного отображения Microsoft EF Core.

3 Построение моделей сущностей для SQL Server с помощью EF Core

Эта глава посвящена работе с реляционными данными, хранящимися в SQL Server, Azure SQL Database или Azure SQL Edge, с помощью технологии более высокого уровня — сопоставления объектов с хранилищем данных (ORM) *Entity Framework Core (EF Core)*. Затем вы узнаете, как хранить модели сущностей, использующие иерархии наследования, с помощью трех различных стратегий сопоставления. И наконец, создадите библиотеки классов для БД SQL Server, которые будут использоваться в примерах кода в остальных главах этой книги.

Управление данными с помощью EF Core

EF Core — это технология *объектно-реляционного отображения (ORM)*, которая строит более высокий уровень абстракции над ADO.NET и использует его возможности для работы с SQL Server. Поскольку это технология более высокого уровня, она не так эффективна, как ADO.NET непосредственно, но разработчикам может быть проще работать с ней, поскольку данные обрабатываются как объекты, а не строки разных таблиц. Для ООП-разработчика такой подход более интуитивен.

Фреймворк EF Core 8 ориентирован только на .NET 8, а версия EF Core 7 — на .NET 6, поэтому его можно использовать с релизом как *долгосрочным (Long Term Support, LTS)* .NET 6, так и со *стандартным сроком поддержки (Standard Term Support, STS)* .NET 7 (рис. 3.1).

Фреймворк EF Core 9, выпущенный в ноябре 2024 года, тоже ориентирован на платформу .NET 8 и более позднюю версию с долгосрочной поддержкой. В листингах следует обновить номера версий пакетов, и код, написанный для предыдущей версии EF Core, по-прежнему будет работать. Вся необходимая информация задокументирована разработчиками в спецификациях к релизу EF Core 9.

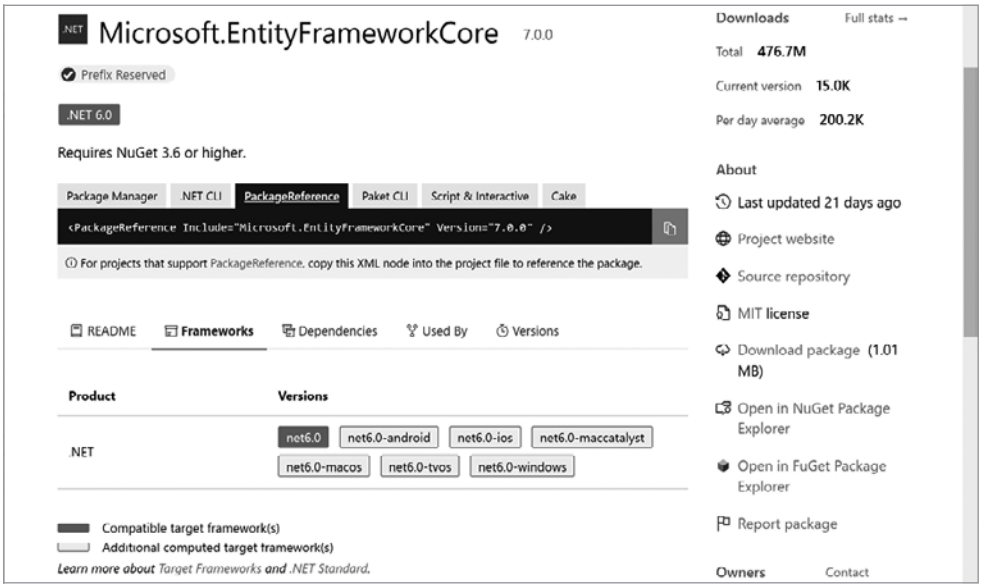


Рис. 3.1. EF Core 7 ориентирован на платформу .NET 6 или более позднюю версию

Обзор Entity Framework Core

Помимо традиционных РСУБД, таких как SQL Server, EF Core поддерживает современные облачные нереляционные хранилища данных без сконфигурированной структуры, такие как Azure Cosmos DB и MongoDB, в ряде случаев через сторонние инструменты.

Существует два подхода к работе с EF Core:

- *Database First* — БД уже существует, поэтому вы создаете модель, соответствующую структуре и функционалу БД;
- *Code First* — базы данных нет, поэтому вы создаете модель, а затем с помощью EF Core создаете БД, соответствующую структуре и функционалу БД.

Мы будем использовать EF Core с существующей БД, поскольку это наиболее распространенный сценарий. Позже в этой главе мы рассмотрим методику Code First, создавая БД после классов.

Жесткая привязка моделей на основе существующей БД

Жесткая привязка — это процесс создания классов, представляющих модель существующей БД путем обратной разработки. Функциональный инструмент привязки допускает расширение автоматически созданных классов и их дальнейшую генерацию без потери расширений.

Если вы не планируете генерировать классы с помощью инструмента жесткой привязки, меняйте код автоматически созданных классов по своему усмотрению. Сгенерированный инструментом код не всегда отражает ваши пожелания относительно его функционала.

Дельный совет: смело изменяйте сгенерированный инструментом код по своему усмотрению. Например, при использовании SQLite в сочетании БД Northwind столбцы даты/времени сопоставляются со строковыми свойствами, а столбцы с ценами — со свойствами типа double. В БД Northwind их лучше сопоставить с типами DateTime и decimal соответственно (в другой БД ситуация может быть совершенно иной). Еще один пример: в БД Northwind идентификаторы CustomerId должны состоять из пяти прописных букв. Инструмент не учитывает это правило по умолчанию, поэтому в рамках проверки следует использовать регулярное выражение. Не забывайте, что инструменты .NET постоянно совершенствуются, поэтому данные советы могут быть неактуальны к моменту чтения этой книги.

Настройка инструмента dotnet-ef

В .NET есть консольный инструмент `dotnet`. Его можно дополнить функционалом, который пригодится для работы с фреймворком EF Core. С его помощью можно создавать и осуществлять миграции данных из одной модели в другую, генерировать код модели на основе существующей БД и выполнять другие операции.

Инструмент `dotnet-ef` не устанавливается автоматически, поэтому *глобально* или *локально* установите соответствующий пакет. Если у вас уже установлена старая версия инструмента, необходимо удалить ее перед установкой новой.

Удостоверьтесь, что у вас установлена последняя версия инструмента.

1. В терминале проверьте, установлен ли глобальный инструмент `dotnet-ef`:

```
dotnet tool list --global
```

2. Проверьте наличие устаревшей версии инструмента, например для платформы .NET 7:

```
Package Id                          Version        Commands
----------------------------------------------------------------
dotnet-ef                           7.0.0          dotnet-ef
microsoft.web.librarymanager.cli    2.1.175        libman
redth.net.maui.check                0.5.6          maui-check
```

3. Если старая версия уже установлена, удалите ее следующей командой:

```
dotnet tool uninstall --global dotnet-ef
```

4. Теперь установите актуальную версию:

```
dotnet tool install --global dotnet-ef
```

5. При необходимости добавьте каталог `dotnet tools` в переменную окружения `PATH`, как описано в выводе инструмента `dotnet-ef` (инструкции различны в зависимости от используемой операционной системы).

Если вы хотите установить предварительную версию инструмента, например для пререлизов EF Core 9, укажите номер версии со звездочкой:

```
dotnet tool install --global dotnet-ef --version 9-*
```

6. Вместо удаления старой и установки новой версии пакета можно обновить имеющийся пакет следующей командой:

```
dotnet tool update --global dotnet-ef
```

Конфигурирование моделей данных EF Core

Для построения модели сущностей на этапе выполнения в EF Core используется комбинация *соглашений*, *атрибутов аннотаций* и инструкций *Fluent API*, чтобы действия, выполняемые над классами, автоматически преобразовывались в операции с реальной БД. Класс сущности представляет структуру таблицы, а экземпляр класса — строку в этой таблице.

Сначала мы рассмотрим три способа конфигурации модели с примерами кода, а затем создадим несколько классов.

Соглашения EF Core для конфигурации модели

В коде, который мы напишем, будут использоваться перечисленные ниже соглашения. Предполагается, что:

- имя таблицы совпадает с именем свойства `DbSet<T>` в классе `DbContext`, например `Products`;
- имена столбцов соответствуют именам свойств в классе модели сущности, например `ProductId`;
- строковый тип .NET соответствует типу `nvarchar` в БД;
- и в .NET, и в БД используется один и тот же тип `int`;

- первичный ключ — это свойство с именем `Id` или `ID`. Либо, если классу модели сущности присвоено имя `Product`, имя свойства — `ProductId` или `ProductID`. Если тип свойства — `int` или `Guid`, то предполагается, то оно является столбцом `IDENTITY` (тип столбца, автоматически присваивающий значение при вставке).

Дельный совет: существует множество других предопределенных соглашений, которые следует учитывать. Можно также внедрить и применять собственные, но это выходит за рамки данной книги. Вы можете прочитать о них по адресу https://learn.microsoft.com/ru-ru/ef/core/modeling/.

Атрибуты аннотаций EF Core для конфигурации модели

Зачастую соглашений недостаточно для полного сопоставления классов с объектами БД. Так, в ряде БД, например SQLite, используются столбцы динамических типов, поэтому инструмент должен «предполагать», какие типы свойств следует сопоставлять со столбцами согласно текущим данным в них.

Для «интеллектуализации» модели вы можете применить атрибуты аннотаций, показанные в табл. 3.1.

Таблица 3.1. Универсальные атрибуты аннотаций EF Core

Атрибут	Описание
`[Required]`	Определяет, что значение не равно `null`
`[StringLength(50)]`	Определяет, что длина значения не превышает 50 символов
`[RegularExpression(expression)]`	Определяет, что значение соответствует указанному регулярному выражению
`[Column(TypeName = "money", Name = "UnitPrice")]`	Определяет тип и имя столбца в таблице

Например, в базе данных максимальная длина названия товара составляет 40 символов, а значение не может быть `null`, как показано во фрагменте DDL-кода ниже, создающем таблицу `Products`:

```
CREATE TABLE Products (
    ProductId       INTEGER       PRIMARY KEY,
    ProductName     NVARCHAR (40) NOT NULL,
    SupplierId      "INT",
    ...
);
```

Для этого в классе `Product` применяются атрибуты следующим образом:

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

Дельный совет: при включенной проверке на обращение к значению null нет необходимости помечать ссылочный тип, не допускающий значение null, атрибутом [Required], как показано выше. Это связано с тем, что возможность присваивания значений null в C# будет передана модели EF Core. Свойство string будет обязательным; а свойство string? — нет, то есть, другими словами, nullable-типом. Подробнее об этом можно прочитать по адресу https://learn.microsoft.com/ru-ru/ef/core/modeling/entity-properties?tabs=data-annotations%2Cwith-nrt#required-and-optional-properties.

Без очевидного соответствия между типами .NET и типами БД можно использовать атрибут.

Например, столбцу `UnitPrice` таблицы `Products` назначен тип `money`. В .NET нет такого же типа, поэтому вместо него следует использовать тип `decimal`:

```
[Column(TypeName = "money")]
public decimal? UnitPrice { get; set; }
```

Другой пример — таблица `Categories`, показанная в следующем DDL-коде:

```
CREATE TABLE Categories (
    CategoryId   INTEGER       PRIMARY KEY,
    CategoryName NVARCHAR (15) NOT NULL,
    Description  "NTEXT",
    Picture      "IMAGE"
);
```

Размер столбца `Description` может превышать предельные 8000 символов, допустимые в переменной типа `nvarchar`, поэтому вместо него необходимо использовать тип `ntext`:

```
[Column(TypeName = "ntext")]
public string? Description { get; set; }
```

Конфигурация модели EF Core с помощью Fluent API

Последний способ конфигурации модели — применение Fluent API. Его можно использовать вместо атрибутов, а также в дополнение к ним. Подойдет, если модели сущностей необходимо определить в библиотеке классов .NET Standard 2.0 для поддержки устаревших платформ. В этом случае библиотека классов не должна содержать ссылки на библиотеки аннотаций данных. Другая причина заключается в том, что ваши разработчики могут придерживаться стратегии разделения необработанных моделей данных и правил проверки.

К примеру, для конфигурации свойства `ProductName` вместо использования двух атрибутов можно написать эквивалентный оператор Fluent API в методе `OnModelCreating` класса контекста БД:

```
modelBuilder.Entity<Product>()
  .Property(product => product.ProductName)
  .IsRequired() // необходимо только в том случае, если вы отключили
                // проверку на присваивание значения null
  .HasMaxLength(40);
```

Так класс модели сущности упрощается. Мы рассмотрим пример на практике чуть позже.

Инициализация базы данных с помощью Fluent API

Еще одно преимущество Fluent API — предоставление исходных данных для инициализации БД. EF Core автоматически определяет, какие операции вставки, обновления или удаления должны быть выполнены.

Например, если необходимо убедиться, что в новой БД есть хотя бы одна строка в таблице `Product`, можно вызвать метод `HasData` следующим образом:

```
modelBuilder.Entity<Product>()
  .HasData(new Product
  {
    ProductId = 1,
    ProductName = "Chai",
    UnitPrice = 8.99M
  });
```

Вызовы `HasData` действуют либо во время миграции данных, выполняемой обновлением БД командой `dotnet ef database update`, либо при вызове метода `Database.EnsureCreated`.

Наша модель будет привязана к уже инициализированной БД с данными, поэтому нам не понадобится этот прием.

Конфигурирование модели БД Northwind

Для представления БД будет использоваться класс `Northwind`. Чтобы использовать EF Core, класс должен наследоваться от класса `DbContext`. Благодаря этому система способна взаимодействовать с БД и динамически генерировать SQL-операторы для запросов и манипуляций с данными.

Класс, производный от `DbContext`, должен иметь переопределенный метод `OnConfiguring`, настраивающий строку подключения к БД. В этом классе необходимо определить не менее одного свойства типа `DbSet<T>`. Эти свойства представляют таблицы. Для уведомления EF Core о столбцах в таблице свойства `DbSet<T>` используют обобщения, указывая класс, представляющий строку таблицы. Класс модели сущности имеет свойства, представляющие столбцы.

Класс, производный от `DbContext`, может переопределить метод `OnModelCreating` для использования Fluent API вместо атрибутов в классах сущностей. Это улучшает читаемость и удобство сопровождения кода, поскольку вся конфигурация модели находится в одном месте. Можно сочетать оба подхода, но это снижает преимущества.

Если база данных Northwind не создана или была удалена, создайте ее, следуя инструкциям в главе 2.

Давайте построим модель для БД Northwind в консольном приложении.

1. В редакторе кода создайте консольное приложение со следующими настройками:
   - выберите шаблон проекта Console App (Консольное приложение);
   - в поле Solution name (Имя решения) укажите значение `Chapter03`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Console.EFCore`;
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
   - снимите флажок Enable native AOT publishing (Включить публикацию Native AOT).
2. В файле проекта `Northwind.Console.EFCore` добавьте ссылки на пакет EF Core для работы с провайдером БД SQL Server. Также статически импортируйте класс `System.Console` на глобальном уровне:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Design"
    Version="8.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="8.0.0" />
</ItemGroup>

<ItemGroup>
  <Using Include="System.Console" Static="true" />
</ItemGroup>
```

3. Выполните сборку проекта для восстановления зависимостей.

Для выполнения следующего шага необходимо, чтобы аутентификация в строке подключения к БД для локального экземпляра SQL Server осуществлялась с помощью системы проверки подлинности Windows. При необходимости подключения к Azure SQL Database или Azure SQL Edge настройте проверку подлинности, указав идентификатор пользователя и пароль.

4. В терминале, находясь в каталоге `Northwind.Console.EFCore`, создайте модель для всех таблиц в папке `Models`:

```
dotnet ef dbcontext scaffold "Data Source=.;Initial
Catalog=Northwind;Integrated Security=true;TrustServerCertificate=true;"
Microsoft.EntityFrameworkCore.SqlServer --output-dir Models –namespace
Northwind.Models --data-annotations --context NorthwindDb
```

В интерфейсах консольных инструментов команды необходимо вводить в одну строку. В программе dotnet-ef команды часто довольно длинные. Я рекомендую вам набирать код из бумажной книги или копировать/вставлять длинные команды типа этой из электронной книги в обычный текстовый редактор, такой как Блокнот. Далее проверяйте, оформлена ли команда в виде одной строки с правильными пробелами, прежде чем выполнять ее в терминале. Команды из электронной книги будут копироваться с символами новой строки (переносами), пробелы или дефисы могут быть пропущены и т. д. Такая команда не будет работать! Все команды из книги доступны для копирования в правильном формате на странице по адресу https://github.com/markjprice/apps-services-net8/blob/main/docs/command-lines.md.

Особенности реализации

- Выполняемая команда: `dbcontext scaffold`.
- Вид строки подключения зависит от того, подключаетесь вы к локальному SQL Server (с именем экземпляра или без) или к Azure SQL Database.
- Провайдер БД: `Microsoft.EntityFrameworkCore.SqlServer`.
- Выходная папка: `--output-dir Models`.
- Пространство имен: `--namespace Northwind.Models`.
- Аннотации данных, а также Fluent API: `--data-annotations`.
- Переименование контекста из `[`*имя_БД*`]Context: --context NorthwindDb`.

5. Обратите внимание на сообщения и предупреждения о сборке:

```
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.
```

Дельный совет: не фиксируйте проект в Git, пока не устраните такие предупреждения. Если использовались имя пользователя и пароль для подключения к БД SQL Server, эти данные теперь в исходном коде! Мы заменим строку подключения динамической, считывающей данные из переменных окружения.

6. Откройте папку `Models` и обратите внимание на 28 автоматически сгенерированных файлов классов.
7. Откройте файл `Category.cs` и обратите внимание, что он представляет строку в таблице `Categories`:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace Northwind.Models;

[Index("CategoryName", Name = "CategoryName")]
public partial class Category
{
  [Key]
  public int CategoryId { get; set; }

  [StringLength(15)]
  public string CategoryName { get; set; } = null!;

  [Column(TypeName = "ntext")]
  public string? Description { get; set; }

  [Column(TypeName = "image")]
  public byte[]? Picture { get; set; }

  [InverseProperty("Category")]
  public virtual ICollection<Product> Products { get; set; }
    = new List<Product>();
}
```

Особенности реализации

- В коде используется атрибут `[Index]` для класса сущности, который был введен в EF Core. Этот атрибут указывает, какие свойства должны иметь индекс в таблице. Ранее индексы определялись только с помощью Fluent API. Поскольку мы работаем с существующей БД, атрибут не нужен. Однако если мы создадим новую БД, эта информация будет использована для создания индексов в таблице `Categories`.
- Имя таблицы в БД — `Categories`, но инструмент `dotnet-ef` использует стороннюю библиотеку `Humanizer`, чтобы автоматически преобразовать имя класса в `Category`, более подходящее для сущности.
- Класс сущности объявляется с ключевым словом `partial`, чтобы можно было добавлять дополнительный код в разделяемый класс. Это позволяет

повторно генерировать класс сущности без потери изменений в разделяемом классе.

- Свойство `CategoryId` сопровождается атрибутом `[Key]` для явного указания, что это первичный ключ, хотя его имя также соответствует принятому соглашению.
- У свойства `Products` есть атрибут `[InverseProperty]` для указания связи внешнего ключа со свойством `Category` в классе сущности `Product`.

8. Откройте файл `ProductsAboveAveragePrice.cs` и обратите внимание, что он хранит строку, возвращаемую представлением БД, а не таблицей, поэтому он сопровождается атрибутом `[Keyless]`.
9. В файле `NorthwindDb.cs` взгляните на класс, показанный ниже в отредактированном листинге:

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

namespace Northwind.Models;

public partial class NorthwindDb : DbContext
{
  public NorthwindDb()
  {
  }

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

  public virtual DbSet<AlphabeticalListOfProduct>
    AlphabeticalListOfProducts { get; set; }

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

...

  public virtual DbSet<Territory> Territories { get; set; }

  protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)

/* Внимание: для защиты потенциально конфиденциальной информации в строке
подключения ее следует вынести из исходного кода. Чтобы избежать жесткой привязки
строки подключения, можно использовать синтаксис Name=. Чтобы читать ее из
конфигурации, см. https://go.microsoft.com/fwlink/?linkid=2131148.
```

```
Дополнительные рекомендации по хранению строк подключения см. на сайте http://
go.microsoft.com/fwlink/?LinkId=723263.*/

      => optionsBuilder.UseSqlServer("Data Source=.;Initial Catalog=Northwind;
➥ Integrated Security=true;TrustServerCertificate=true;");

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<AlphabeticalListOfProduct>(entity =>
    {
      entity.ToView("Alphabetical list of products");
    });

...

    OnModelCreatingPartial(modelBuilder);
  }

  partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}
```

Особенности реализации

- У класса контекста данных `NorthwindDb` указан модификатор `partial`, чтобы его можно было расширять и регенерировать в будущем. Мы задали имя `NorthwindDb`, потому что `Northwind` используется как пространство имен.
- У класса `NorthwindDb` есть два конструктора: стандартный без параметров и позволяющий передавать параметры. Это полезно в приложениях, где строка подключения указывается на этапе выполнения.
- Свойства `DbSet<T>` представляют таблицы типа `Categories`.
- В методе `OnConfiguring`, если параметры не были указаны в конструкторе, по умолчанию используется строка подключения, определенная в процессе привязки. В нем также присутствует предупреждение компилятора, напоминающее, что не следует жестко кодировать конфиденциальную информацию в строке подключения.
- В методе `OnModelCreating` Fluent API используется для конфигурирования классов сущностей, после чего вызывается разделяемый метод `OnModelCreatingPartial`. Он реализуется в собственном разделяемом классе `Northwind` с собственной конфигурацией Fluent API, не теряемой при регенерации классов моделей.

10. В начале файла `NorthwindDb.cs` укажите оператор импорта пространства имен для работы с типами ADO.NET:

```
using Microsoft.Data.SqlClient; // Для класса SqlConnectionStringBuilder
```

11. Настройте метод `OnConfiguring` так, чтобы строка соединения задавалась динамически, а параметры конфигурации устанавливались через переменные окружения:

```
protected override void OnConfiguring(
  DbContextOptionsBuilder optionsBuilder)
{
  if (!optionsBuilder.IsConfigured)
  {
    SqlConnectionStringBuilder builder = new();

    builder.DataSource = "."; // Синтаксис "Сервер\Экземпляр",
                              // например @".\sqlexpress"
    builder.InitialCatalog = "Northwind";
    builder.TrustServerCertificate = true;
    builder.MultipleActiveResultSets = true;

    // Чтобы сбои происходили быстрее.
    // По умолчанию это значение составляет 15 секунд.
    builder.ConnectTimeout = 3;

    // Если используется встроенная система проверки подлинности Windows
    builder.IntegratedSecurity = true;

    // Если используется система проверки подлинности SQL Server
    // builder.UserID = Environment.GetEnvironmentVariable("MY_SQL_USR");
    // builder.Password = Environment.GetEnvironmentVariable("MY_SQL_PWD");

    optionsBuilder.UseSqlServer(builder.ConnectionString);
  }
}
```

12. Закройте автоматически сгенерированные файлы классов.

Запрос модели Northwind

Теперь мы можем запросить модель.

1. В файле `Program.cs` удалите текущий код и добавьте инструкции для создания экземпляра контекста данных `NorthwindDb`, используя его для запроса товаров, цена которых выше заданного значения:

```
using Microsoft.Data.SqlClient; // Для класса SqlConnectionStringBuilder
using Microsoft.EntityFrameworkCore; // Методы ToQueryString, GetConnectionString
using Northwind.Models; // Для класса NorthwindDb

SqlConnectionStringBuilder builder = new();
```

```
builder.InitialCatalog = "Northwind";
builder.MultipleActiveResultSets = true;
builder.Encrypt = true;
builder.TrustServerCertificate = true;
builder.ConnectTimeout = 10;

WriteLine("Connect to:");
WriteLine("  1 - SQL Server on local machine");
WriteLine("  2 - Azure SQL Database");
WriteLine("  3 - Azure SQL Edge");
WriteLine();
Write("Press a key: ");

ConsoleKey key = ReadKey().Key;
WriteLine(); WriteLine();

if (key is ConsoleKey.D1 or ConsoleKey.NumPad1)
{
  builder.DataSource = "."; // Локальный SQL-сервер
  // @".\apps-services-book"; // Локальный SQL-сервер с именем экземпляра
}
else if (key is ConsoleKey.D2 or ConsoleKey.NumPad2)
{
  builder.DataSource = // Azure SQL Edge
    "tcp:apps-services-book.database.windows.net,1433";
}
else if (key is ConsoleKey.D3 or ConsoleKey.NumPad3)
{
  builder.DataSource = "tcp:127.0.0.1,1433"; // БД Azure SQL
}
else
{
  WriteLine("No data source selected.");
  return;
}

WriteLine("Authenticate using:");
WriteLine(" 1 - Windows Integrated Security");
WriteLine(" 2 - SQL Login, for example, sa");
WriteLine();
Write("Press a key: ");

key = ReadKey().Key;
WriteLine(); WriteLine();

if (key is ConsoleKey.D1 or ConsoleKey.NumPad1)
{
  builder.IntegratedSecurity = true;
}
else if (key is ConsoleKey.D2 or ConsoleKey.NumPad2)
{
  Write("Enter your SQL Server user ID: ");
```

```
  string? userId = ReadLine();
  if (string.IsNullOrWhiteSpace(userId))
  {
    WriteLine("User ID cannot be empty or null.");
    return;
  }

  builder.UserID = userId;

  Write("Enter your SQL Server password: ");
  string? password = ReadLine();
  if (string.IsNullOrWhiteSpace(password))
  {
    WriteLine("Password cannot be empty or null.");
    return;
  }

  builder.Password = password;
  builder.PersistSecurityInfo = false;
}
else
{
  WriteLine("No authentication selected.");
  return;
}

DbContextOptionsBuilder<NorthwindDb> options = new();
options.UseSqlServer(builder.ConnectionString);

using (NorthwindDb db = new(options.Options))
{
  Write("Enter a unit price: ");
  string? priceText = ReadLine();

  if (!decimal.TryParse(priceText, out decimal price))
  {
    WriteLine("You must enter a valid unit price.");
    return;
  }

  // Необходимо использовать тип var, потому что проекция
  // выполняется на анонимный тип
  var products = db.Products
    .Where(p => p.UnitPrice > price)
    .Select(p => new { p.ProductId, p.ProductName, p.UnitPrice });

  WriteLine("-------------------------------------------------------");
  WriteLine("| {0,5} | {1,-35} | {2,8} |", "Id", "Name", "Price");
  WriteLine("-------------------------------------------------------");

  foreach (var p in products)
  {
```

```
    WriteLine("| {0,5} | {1,-35} | {2,8:C} |",
      p.ProductId, p.ProductName, p.UnitPrice);
  }

  WriteLine("--------------------------------------------------------");

  WriteLine(products.ToQueryString());
  WriteLine();
  WriteLine($"Provider:   {db.Database.ProviderName}");
  WriteLine($"Connection: {db.Database.GetConnectionString()}");
}
```

2. Запустите консольное приложение и проверьте вывод (показан фрагмент результата):

```
Enter a unit price: --
|    Id | Name                                |   Price--
|     9 | Mishi Kobe Niku                     |  £97.00 |
|    18 | Carnarvon Tigers                    |  £62.50 |
|    20 | Sir Rodney's Marmalade              |  £81.00 |
|    29 | Thüringer Rostbratwurst             | £123.79 |
|    38 | Côte de Blaye                       | £263.50--
DECLARE @__price_0 decimal(2) = 60.0;

SELECT [p].[ProductId], [p].[ProductName], [p].[UnitPrice]
FROM [Products] AS [p]
WHERE [p].[UnitPrice] > @__price_0

Provider: Microsoft.EntityFrameworkCore.SqlServer
Connection: Data Source=tcp:apps-services-book.database.windows.net,1433;
Initial Catalog=Northwind;Persist Security Info=False;User ID=<censored>;
Password=<censored>;Multiple Active Result Sets=False;Encrypt=True;Trust Server
Certificate=False;Connection Timeout=10;
```

В вашем случае строка подключения будет выглядеть иначе.

Управление отслеживанием сущностей

Начнем с определения разрешения удостоверений сущности. EF Core определяет его с помощью уникального первичного ключа для каждого экземпляра. Так исключается двусмысленность в идентичности сущностей и их отношениях.

EF Core отслеживает только сущности с ключами, так как ключ используется для уникальной идентификации в базе данных. Сущности без ключей, такие как представления, не отслеживаются.

По умолчанию EF Core отслеживает сущности в локальной памяти, чтобы при вызове метода `SaveChanges` все изменения (добавление, изменение или удаление сущностей) были применены к базе данных. Например, если при одном запросе вы извлекаете данные клиентов из Германии, а затем в том же контексте выполняете запрос клиентов с именами, начинающимися с буквы «А», то, если клиент уже существует в контексте, информация о нем не будет загружена повторно. Однако, если между запросами его данные (например, номер телефона) были обновлены в БД, отслеживаемая сущность в контексте не получит эти обновления.

Если вы хотите избежать отслеживания изменений или всегда загружать свежие данные при каждом запросе, можно отключить отслеживание. Для этого используйте в запросе метод `AsNoTracking`:

```
var products = db.Products
  .AsNoTracking()
  .Where(p => p.UnitPrice > price)
  .Select(p => new { p.ProductId, p.ProductName, p.UnitPrice });
```

Для отключения отслеживания запросов по умолчанию в контексте данных установите значение `NoTracking`:

```
db.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
```

Чтобы отключить отслеживание для отдельного запроса, но сохранить разрешение идентификации, используйте в запросе метод `AsNoTrackingWithIdentityResolution`:

```
var products = db.Products
  .AsNoTrackingWithIdentityResolution()
  .Where(p => p.UnitPrice > price)
  .Select(p => new { p.ProductId, p.ProductName, p.UnitPrice });
```

Чтобы отключить отслеживание, но оставить разрешение удостоверений по умолчанию, присвойте параметру отслеживания запросов значение `NoTrackingWithIdentityResolution`:

```
db.ChangeTracker.QueryTrackingBehavior =
  QueryTrackingBehavior.NoTrackingWithIdentityResolution;
```

Чтобы установить значения по умолчанию для всех новых экземпляров контекста данных, в методе `OnConfiguring` вызовите метод `UseQueryTrackingBehavior`:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
  optionsBuilder.UseSqlServer(connectionString)
    .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
}
```

Сценарий с отслеживанием по умолчанию

По умолчанию отслеживание изменений осуществляется с разрешением удостоверений. После загрузки сущности в контекст данных базовые изменения не отражаются, и существует только одна ее копия. Локальные изменения сущностей отслеживаются, и вызов метода `SaveChanges` обновляет БД (табл. 3.2).

Таблица 3.2. Сценарий отслеживания по умолчанию

Действие	Сущность в контексте данных	Строка в БД
Запрос клиентов из Германии	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-4567
Изменение телефона в БД	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-9876
Запрос клиентов с именами, начинающимися с буквы «А»	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-9876
Запрос клиентов из Германии	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-9876
Изменение телефона в сущности	Alfred's Futterkiste, 123-1928	Alfred's Futterkiste, 123-9876
Сохранение изменений	Alfred's Futterkiste, 123-1928	Alfred's Futterkiste, 123-1928

Тот же сценарий при отсутствии отслеживания

Без отслеживания и разрешения удостоверений каждый запрос загружает в контекст данных новый экземпляр строки БД, включая базовые изменения, что может привести к дублированию и смешиванию устаревших и обновленных данных. Изменения локальных сущностей не отслеживаются, поэтому метод `SaveChanges` не используется (табл. 3.3).

Таблица 3.3. Сценарий без отслеживания

Действие	Сущности в контексте данных	Строка в БД
Запрос клиентов из Германии	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-4567
Изменение телефона в БД	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-9876
Запрос клиентов с именами, начинающимися с буквы «А»	Alfred's Futterkiste, 123-4567; Alfred's Futterkiste, 123-9876	Alfred's Futterkiste, 123-9876

Действие	Сущности в контексте данных	Строка в БД
Запрос клиентов из Германии	Alfred's Futterkiste, 123-4567; Alfred's Futterkiste, 123-9876; Alfred's Futterkiste, 123-9876	Alfred's Futterkiste, 123-9876
Изменение телефона в сущности	Alfred's Futterkiste, 123-4567; Alfred's Futterkiste, 123-9876; Alfred's Futterkiste, 123-1928	Alfred's Futterkiste, 123-9876
Сохранение изменений	Alfred's Futterkiste, 123-4567; Alfred's Futterkiste, 123-9876; Alfred's Futterkiste, 123-1928	Alfred's Futterkiste, 123-9876

Тот же сценарий без отслеживания с разрешением идентификации

Отсутствие отслеживания при разрешении удостоверения. После загрузки сущности в контекст данных базовые изменения не отражаются, и существует только одна ее копия. Локальные изменения сущности не отслеживаются, поэтому метод `SaveChanges` не используется (табл. 3.4).

Таблица 3.4. Сценарий с разрешением удостоверения

Действие	Сущности в контексте данных	Строка в БД
Запрос клиентов из Германии	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-4567
Изменение телефона в БД	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-9876
Запрос клиентов с именами, начинающимися с буквы «А»	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-9876
Запрос клиентов из Германии	Alfred's Futterkiste, 123-4567	Alfred's Futterkiste, 123-9876
Изменение телефона в сущности	Alfred's Futterkiste, 123-1928	Alfred's Futterkiste, 123-9876
Сохранение изменений	Alfred's Futterkiste, 123-1928	Alfred's Futterkiste, 123-9876

Заключение

Выбор зависит от вашего конкретного сценария. В некоторых блогах можно найти советы по улучшению запросов EF Core с помощью метода `AsNoTracking`. Однако если запрос возвращает множество сущностей, а затем выполняется снова в том же

контексте, могут появиться дубликаты, что негативно скажется на ресурсах памяти и производительности. Понимание работы различных вариантов отслеживания поможет выбрать оптимальный подход для вашего случая. Далее мы обсудим, как работать с иерархиями наследования.

Работа с иерархиями наследования в EF Core

Допустим, у вас есть иерархия классов C# для хранения данных о студентах и сотрудниках, представляющих типы людей. У всех есть имя и идентификатор, у студентов — изучаемый предмет, а у сотрудников — дата приема на работу:

```
public abstract class Person
{
  public int Id { get; set; }
  public string? Name { get; set; }
}

public class Student : Person
{
  public string? Subject { get; set; }
}

public class Employee : Person
{
  public DateTime HireDate { get; set; }
}
```

По умолчанию EF Core использует стратегию *«одна таблица — одна иерархия» (Table-Per-Hierarchy, TPH)*, объединяя все типы в одну таблицу. В EF Core 5 добавлена поддержка стратегии *«одна таблица — один тип» (Table-Per-Type, TPT)*, а в EF Core 7 — *«одна таблица — конкретный тип» (Table-Per-Concrete-Type, TPC)*. Рассмотрим различия между этими подходами.

Стратегия сопоставления «одна таблица — одна иерархия» (TPH)

В стратегии TPH для иерархии `Person-Student-Employee` создается одна таблица с дискриминационным столбцом, определяющим тип записи — студент или сотрудник. При этом столбцы, относящиеся только к определенным типам, остаются пустыми для остальных:

```
CREATE TABLE [People] (
  [Id] int NOT NULL IDENTITY,
  [Name] nvarchar(max) NOT NULL,
  [Discriminator] nvarchar(max) NOT NULL,
  [Subject] nvarchar(max) NULL,
  [HireDate] nvarchar(max) NULL,
  CONSTRAINT [PK_People] PRIMARY KEY ([Id])
);
```

Некоторые данные в таблице могут выглядеть следующим образом (табл. 3.5).

Таблица 3.5. Фрагмент таблицы People

Id	Name	Discriminator	Subject	HireDate
1	Roman Roy	Student	History	NULL
2	Kendall Roy	Employee	NULL	02/04/2014
3	Siobhan Roy	Employee	NULL	12/09/2020

Стратегия TPH использует столбец дискриминатора (`Discriminator`) для хранения имени класса в каждой строке. Она требует, чтобы столбцы свойств производных типов, таких как `Subject` и `HireDate`, поддерживали значение `null`, что может стать проблемой, если эти свойства обязательны. EF Core не решает это автоматически.

Главные преимущества TPH — простота и высокая производительность, поэтому этот подход используется по умолчанию.

Дельный совет: если дискриминатор имеет много значений, можно повысить производительность, создав для него индекс. Однако при небольшом количестве значений индекс может негативно сказаться на эффективности, замедляя обновления. В данном случае с двумя вариантами — Student и Employee — при 100 000 строк индекс мало повлияет на работу.

Стратегия сопоставления «одна таблица — один тип» (TPT)

В стратегии TPT для иерархии `Person-Student-Employee` создается отдельная таблица для каждого типа:

```
CREATE TABLE [People] (
  [Id] int NOT NULL IDENTITY,
  [Name] nvarchar(max) NOT NULL,
  CONSTRAINT [PK_People] PRIMARY KEY ([Id])
);

CREATE TABLE [Students] (
  [Id] int NOT NULL,
  [Subject] nvarchar(max) NULL,
  CONSTRAINT [PK_Students] PRIMARY KEY ([Id])
  CONSTRAINT [FK_Students_People] FOREIGN KEY ([Id]) REFERENCES [People] ([Id])
);

CREATE TABLE [Employees] (
  [Id] int NOT NULL,
  [HireDate] nvarchar(max) NULL,
  CONSTRAINT [PK_Employees] PRIMARY KEY ([Id])
  CONSTRAINT [FK_Employees_People] FOREIGN KEY ([Id]) REFERENCES [People] ([Id])
);
```

Фрагмент табличных данных показан ниже (табл. 3.6–3.8).

Таблица 3.6. Таблица People

Id	Name
1	Roman Roy
2	Kendall Roy
3	Siobhan Roy

Таблица 3.7. Таблица Students

Id	Subject
1	History

Таблица 3.8. Таблица Employees

Id	HireDate
2	02/04/2014
3	12/09/2020

Главное преимущество TPT — экономия места за счет полной нормализации данных. Однако этот подход замедляет работу, так как одна сущность распределяется по нескольким таблицам, усложняя запросы. Обычно TPT не лучший выбор, поэтому используйте этот подход только при уже нормализованной структуре, которую нельзя изменить.

Стратегия сопоставления «одна таблица — конкретный тип» (TPC)

В стратегии TPC для иерархии `Person-Student-Employee` создается отдельная таблица для каждого неабстрактного типа:

```
CREATE TABLE [Employees] (
  [Id] int NOT NULL DEFAULT (NEXT VALUE FOR [PersonIds]),
  [Name] nvarchar(40) NOT NULL,
  [HireDate] datetime2 NOT NULL,
  CONSTRAINT [PK_Employees] PRIMARY KEY ([Id])
);

CREATE TABLE [Students] (
  [Id] int NOT NULL DEFAULT (NEXT VALUE FOR [PersonIds]),
  [Name] nvarchar(40) NOT NULL,
  [Subject] nvarchar(max) NULL,
  CONSTRAINT [PK_Students] PRIMARY KEY ([Id])
);
```

Поскольку таблиц со столбцом IDENTITY для присвоения значений Id нет, можно добавить команду (NEXT VALUE FOR [PersonIds]) для определения последовательности, совместно используемой двумя таблицами, чтобы избежать присвоения одинаковых идентификаторов. Ниже приведен SQL-код для определения такой последовательности:

```
CREATE SEQUENCE [PersonIds] AS int START WITH 4 INCREMENT BY 1 NO
MINVALUE NO MAXVALUE NO CYCLE;
```

Фрагмент табличных данных показан в табл. 3.9, 3.10.

Таблица 3.9. Таблица Students

Id	Name	Subject
1	Roman Roy	History

Таблица 3.10. Таблица Employees

Id	Name	HireDate
2	Kendall Roy	02/04/2014
3	Siobhan Roy	12/09/2020

Основное преимущество стратегии TPC — высокая производительность, так как для запроса конкретного типа требуется только одна таблица. Благодаря этому отпадает необходимость в объединениях. Этот подход идеален для больших иерархий со множеством конкретных типов, каждый из которых имеет уникальные свойства.

Настройка стратегий сопоставления для иерархий наследования

Сначала нужно добавить все типы в модель:

```
public DbSet<Person> People { get; set; }
public DbSet<Student> Students { get; set; }
public DbSet<Employee> Employees { get; set; }
```

Для стратегии TPH ничего настраивать не нужно — она используется по умолчанию! Если хотите указать это явно, в методе `OnModelCreating` контекста данных примените стратегию к базовому классу. Так как `Person` — это базовый класс, вызовите `UseTphMappingStrategy` для него:

```
modelBuilder.Entity<Person>().UseTphMappingStrategy();
```

Для использования одной из двух других стратегий сопоставления вызовите соответствующий метод:

```
modelBuilder.Entity<Person>().UseTptMappingStrategy();
modelBuilder.Entity<Person>().UseTpcMappingStrategy();
```

Далее можно указать имя таблицы для каждого класса сущностей:

```
modelBuilder.Entity<Student>().ToTable("Students");
modelBuilder.Entity<Employee>().ToTable("Employees");
```

Стратегия TPC требует общей последовательности, поэтому ее тоже нужно настроить:

```
modelBuilder.HasSequence<int>("PersonIds");

modelBuilder.Entity<Person>().UseTpcMappingStrategy()
  .Property(e => e.Id).HasDefaultValueSql("NEXT VALUE FOR [PersonIds]");
```

Пример стратегий сопоставления иерархий

Теперь рассмотрим все вышеописанное на практике, воспользовавшись новой базой данных и проектом с именем `HierarchyMapping`.

1. В редакторе кода создайте консольное приложение со следующими настройками:
   - выберите шаблон проекта Console App (Консольное приложение);
   - в поле Solution name (Имя решения) укажите значение `Chapter03`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Console.HierarchyMapping`;
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
   - снимите флажок Enable native AOT publishing (Включить публикацию Native AOT).
2. Назначьте проект `Northwind.Console.HierarchyMapping` в качестве запускаемого.
3. В этом проекте добавьте ссылки на пакеты провайдера данных EF Core для SQL Server. Также статически импортируйте класс `System.Console` на глобальном уровне:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Design"
    Version="8.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="8.0.0" />
</ItemGroup>

<ItemGroup>
  <Using Include="System.Console" Static="true" />
</ItemGroup>
```

4. Выполните сборку проекта для восстановления зависимостей.
5. В том же проекте создайте папку `Models`.
6. В папке `Models` создайте файл класса `Person.cs` и измените его содержимое следующим образом:

```
using System.ComponentModel.DataAnnotations; // Для атрибута [Required]

namespace Northwind.Models;

public abstract class Person
{
  public int Id { get; set; }

  [Required]
  [StringLength(40)]
  public string? Name { get; set; }
}
```

7. В папке `Models` создайте файл класса `Student.cs` и измените его содержимое следующим образом:

```
namespace Northwind.Models;

public class Student : Person
{
  public string? Subject { get; set; }
}
```

8. В папке `Models` создайте файл класса `Employee.cs` со следующим содержимым:

```
namespace Northwind.Models;

public class Employee : Person
{
  public DateTime HireDate { get; set; }
}
```

9. В папке `Models` создайте файл класса `HierarchyDb.cs` и измените его содержимое следующим образом:

```
using Microsoft.EntityFrameworkCore; // Для атрибута DbSet<T>

namespace Northwind.Models;

public class HierarchyDb : DbContext
{
  public DbSet<Person>? People { get; set; }
  public DbSet<Student>? Students { get; set; }
  public DbSet<Employee>? Employees { get; set; }
```

```
  public HierarchyDb(DbContextOptions<HierarchyDb> options) : base(options)
  {
  }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Person>()
      .UseTphMappingStrategy();

    // Заполнение БД примерами данных

    Student p1 = new() { Id = 1, Name = "Roman Roy",
      Subject = "History" };

    Employee p2 = new() { Id = 2, Name = "Kendall Roy",
      HireDate = new(year: 2014, month: 4, day: 2) };

    Employee p3 = new() { Id = 3, Name = "Siobhan Roy",
      HireDate = new(year: 2020, month: 9, day: 12) };

    modelBuilder.Entity<Student>().HasData(p1);
    modelBuilder.Entity<Employee>().HasData(p2, p3);
  }
}
```

10. В файле `Program.cs` удалите существующие операторы. Добавьте операторы для настройки строки подключения для контекста данных `HierarchyDb`. Затем используйте этот контекст для удаления и последующего создания базы данных `HierarchyMapping` (не `Northwind`!). Просмотрите автоматически сгенерированный сценарий SQL и выведите данные о людях, включая студентов и сотрудников:

```
using Microsoft.Data.SqlClient; // Для класса SqlConnectionStringBuilder
using Microsoft.Extensions.Options;
using Microsoft.EntityFrameworkCore; // Метод GenerateCreateScript()
using Northwind.Models; // HierarchyDb, Person, Student, Employee

DbContextOptionsBuilder<HierarchyDb> options = new();

SqlConnectionStringBuilder builder = new();

builder.DataSource = "."; // "ServerName\InstanceName",
                          // например @".\sqlexpress"
builder.InitialCatalog = "HierarchyMapping";
builder.TrustServerCertificate = true;
builder.MultipleActiveResultSets = true;

// Чтобы сбои происходили быстрее.
// По умолчанию это значение составляет 15 секунд.
builder.ConnectTimeout = 3;
```

```
// Если используется встроенная система проверки подлинности Windows
builder.IntegratedSecurity = true;

// Если используется система проверки подлинности SQL Server
// builder.UserID = Environment.GetEnvironmentVariable("MY_SQL_USR");
// builder.Password = Environment.GetEnvironmentVariable("MY_SQL_PWD");

options.UseSqlServer(builder.ConnectionString);

using (HierarchyDb db = new(options.Options))
{
  bool deleted = await db.Database.EnsureDeletedAsync();
  WriteLine($"Database deleted: {deleted}");

  bool created = await db.Database.EnsureCreatedAsync();
  WriteLine($"Database created: {created}");

  WriteLine("SQL script used to create the database:");
  WriteLine(db.Database.GenerateCreateScript());

  if (db.Students is null || !db.Students.Any())
  {
    WriteLine("There are no students.");
  }
  else
  {
    foreach (Student student in db.Students)
    {
      WriteLine("{0} studies {1}",
      student.Name, student.Subject);
    }
  }

  if (db.Employees is null || !db.Employees.Any())
  {
    WriteLine("There are no employees.");
  }
  else
  {
    foreach (Employee employee in db.Employees)
    {
      WriteLine("{0} was hired on {1}",
        employee.Name, employee.HireDate);
    }
  }

  if (db.People is null || !db.People.Any())
  {
    WriteLine("There are no people.");
  }
  else
```

```
  {
    foreach (Person person in db.People)
    {
      WriteLine("{0} has ID of {1}",
        person.Name, person.Id);
    }
  }
}
```

11. Запустите консольное приложение и обратите внимание на результаты, включая единственную созданную таблицу с именем `People`:

```
Database deleted: False
Database created: True

SQL script used to create the database:
CREATE TABLE [People] (
    [Id] int NOT NULL IDENTITY,
    [Name] nvarchar(40) NOT NULL,
    [Discriminator] nvarchar(8) NOT NULL,
    [HireDate] datetime2 NULL,
    [Subject] nvarchar(max) NULL,
    CONSTRAINT [PK_People] PRIMARY KEY ([Id])
);
GO

IF EXISTS (SELECT * FROM [sys].[identity_columns] WHERE [name] IN
(N'Id', N'Discriminator', N'Name', N'Subject') AND [object_id] = OBJECT_
ID(N'[People]'))
    SET IDENTITY_INSERT [People] ON;
INSERT INTO [People] ([Id], [Discriminator], [Name], [Subject])
VALUES (1, N'Student', N'Roman Roy', N'History');
IF EXISTS (SELECT * FROM [sys].[identity_columns] WHERE [name] IN
(N'Id', N'Discriminator', N'Name', N'Subject') AND [object_id] = OBJECT_
ID(N'[People]'))
    SET IDENTITY_INSERT [People] OFF;
GO

IF EXISTS (SELECT * FROM [sys].[identity_columns] WHERE [name] IN
(N'Id', N'Discriminator', N'HireDate', N'Name') AND [object_id] = OBJECT_
ID(N'[People]'))
    SET IDENTITY_INSERT [People] ON;
INSERT INTO [People] ([Id], [Discriminator], [HireDate], [Name])
VALUES (2, N'Employee', '2014-04-02T00:00:00.0000000', N'Kendall Roy'),
(3, N'Employee', '2020-09-12T00:00:00.0000000', N'Siobhan Roy');
IF EXISTS (SELECT * FROM [sys].[identity_columns] WHERE [name] IN
(N'Id', N'Discriminator', N'HireDate', N'Name') AND [object_id] = OBJECT_
ID(N'[People]'))
    SET IDENTITY_INSERT [People] OFF;
GO
```

```
Roman Roy studies History
Kendall Roy was hired on 02/04/2014 00:00:00
Siobhan Roy was hired on 12/09/2020 00:00:00
Roman Roy has ID of 1
Kendall Roy has ID of 2
Siobhan Roy has ID of 3
```

12. В программе для работы с базами данных просмотрите содержимое таблицы `People` (рис. 3.2).

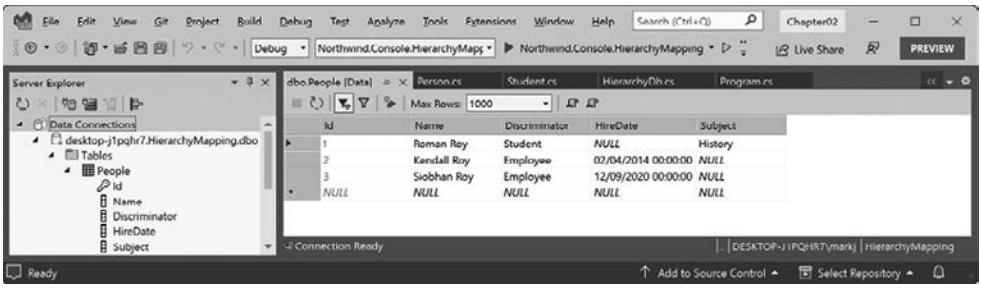


Рис. 3.2. Таблица People при применении стратегии сопоставления TPH

13. Закройте соединение с БД `HierarchyMapping`.
14. В файле `HierarchyDb.cs` закомментируйте вызов метода, который настраивает стратегию TPH, и добавьте вызов метода для настройки стратегии TPT:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<Person>()
    // .UseTphMappingStrategy();
    .UseTptMappingStrategy();
```

15. Запустите консольное приложение и обратите внимание на результаты, в том числе на три созданные таблицы — `People`, `Students` и `Employees` (показан фрагмент):

```
Database deleted: True
Database created: True
SQL script used to create the database:
CREATE TABLE [People] (
    [Id] int NOT NULL IDENTITY,
    [Name] nvarchar(40) NOT NULL,
    CONSTRAINT [PK_People] PRIMARY KEY ([Id])
);
GO
CREATE TABLE [Employees] (
    [Id] int NOT NULL,
    [HireDate] datetime2 NOT NULL,
    CONSTRAINT [PK_Employees] PRIMARY KEY ([Id]),
```

```
    CONSTRAINT [FK_Employees_People_Id] FOREIGN KEY ([Id]) REFERENCES
    [People] ([Id])
);
GO
CREATE TABLE [Students] (
    [Id] int NOT NULL,
    [Subject] nvarchar(max) NULL,
    CONSTRAINT [PK_Students] PRIMARY KEY ([Id]),
    CONSTRAINT [FK_Students_People_Id] FOREIGN KEY ([Id]) REFERENCES
[People] ([Id])
);
GO
```

16. В программе для работы с базами данных просмотрите содержимое таблиц (рис. 3.3).

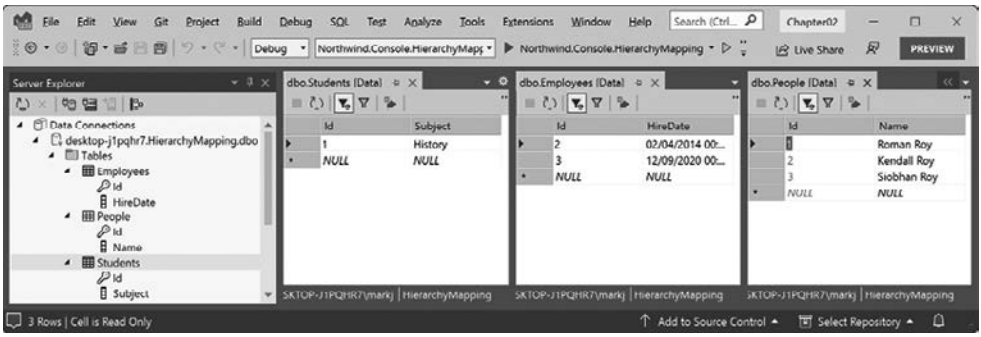


Рис. 3.3. Таблицы при использовании стратегии сопоставления TPT

17. Закройте соединение с БД `HierarchyMapping`.
18. В файле `HierarchyDb.cs` закомментируйте вызов метода, настраивающего TPT, и добавьте вызов метода, реализующий стратегию TPC. Также настройте последовательность для отслеживания присвоенных идентификаторов. Так как мы добавим три записи, начните отсчет значений идентификаторов с 4:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<Person>()
    // .UseTphMappingStrategy();
    // .UseTptMappingStrategy();
    .UseTpcMappingStrategy()
    .Property(person => person.Id)
    .HasDefaultValueSql("NEXT VALUE FOR [PersonIds]");

  modelBuilder.HasSequence<int>("PersonIds", builder =>
  {
    builder.StartsAt(4);
  });
```

19. Запустите консольное приложение и обратите внимание на результаты, в том числе на две созданные таблицы, `Students` и `Employees`, а также общую последовательность, начинающуюся с 4, как показано в выводе ниже (фрагмент):

```
CREATE SEQUENCE [PersonIds] AS int START WITH 4 INCREMENT BY 1 NO
MINVALUE NO MAXVALUE NO CYCLE;
GO

CREATE TABLE [Employees] (
    [Id] int NOT NULL DEFAULT (NEXT VALUE FOR [PersonIds]),
    [Name] nvarchar(40) NOT NULL,
    [HireDate] datetime2 NOT NULL,
    CONSTRAINT [PK_Employees] PRIMARY KEY ([Id])
);
GO

CREATE TABLE [Students] (
    [Id] int NOT NULL DEFAULT (NEXT VALUE FOR [PersonIds]),
    [Name] nvarchar(40) NOT NULL,
    [Subject] nvarchar(max) NULL,
    CONSTRAINT [PK_Students] PRIMARY KEY ([Id])
);
GO
```

20. В программе для работы с базами данных просмотрите содержимое таблиц (рис. 3.4).

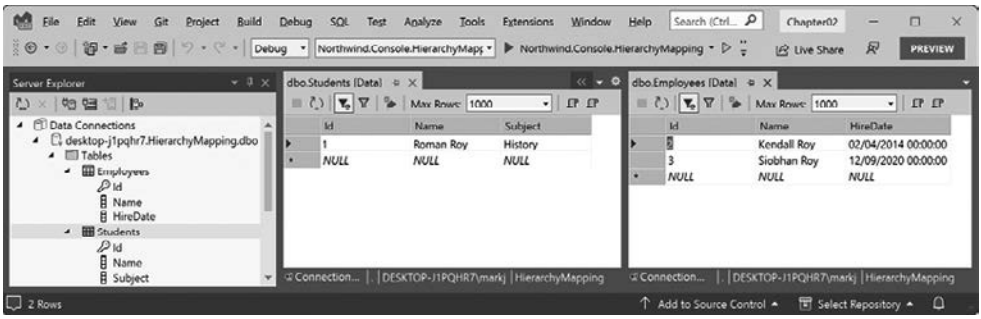


Рис. 3.4. Таблицы при использовании стратегии сопоставления TPC

21. Закройте соединение с БД `HierarchyMapping`.
22. В файле `Program.cs` после оператора вывода сценария создания базы данных в консоль напишите код для добавления двух новых людей, используя текущий контекст базы данных:

```
WriteLine(db.Database.GenerateCreateScript());

if ((db.Employees is not null) && (db.Students is not null))
```

```
{
  db.Students.Add(new Student { Name = "Connor Roy",
    Subject = "Politics" });

  db.Employees.Add(new Employee { Name = "Kerry Castellabate",
    HireDate = DateTime.UtcNow });

  int result = db.SaveChanges();
  WriteLine($"{result} people added.");
}
```

23. Запустите консольное приложение и обратите внимание на результаты, в том числе на двух персон, добавленных через контекст БД с идентификаторами, начинающимися на 4 (ниже показан фрагмент):

```
2 people added.
Roman Roy studies History
Connor Roy studies Politics
Kendall Roy was hired on 02/04/2014 00:00:00
Siobhan Roy was hired on 12/09/2020 00:00:00
Kerry Castellabate was hired on 19/05/2023 10:13:53
Kendall Roy has ID of 2
Siobhan Roy has ID of 3
Kerry Castellabate has ID of 4
Roman Roy has ID of 1
Connor Roy has ID of 5
```

Итак, с помощью ORM-инструмента EF Core можно определять иерархию наследования объектов и сопоставлять ее с БД тремя разными способами, используя связанные таблицы. Также мы убедились в удобстве подхода Code First в рамках удаления и создания БД заново при каждом запуске проекта.

Построение многократно используемой модели данных сущностей

Практическим приложениям часто требуется доступ к данным в реляционной БД или в другом хранилище. Ранее в этой главе мы создавали модели EF Core прямо в консольном приложении, где они использовались.

Теперь мы создадим многократно используемую модель данных сущностей для БД Northwind в виде двух библиотек классов. Одна из них будет содержать определения сущностей, таких как `Product` и `Customer`. Вторая — настройку таблиц БД, параметры подключения по умолчанию и конфигурацию модели с помощью Fluent API. Эти библиотеки можно будет использовать в разных приложениях и сервисах, которые мы создадим в следующих главах.

Дельный совет: для моделей данных сущностей нужно создать отдельный проект библиотеки классов, чтобы упростить обмен данными между внутренними веб-серверами и внешними клиентами, включая классические, мобильные и Blazor-приложения.

Создание библиотеки классов для моделей сущностей в SQL Server

Создадим модели сущностей с помощью инструмента `dotnet-ef`.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта Class Library (Библиотека классов);
   - в поле Project name (Имя проекта) укажите значение `Northwind.Common.EntityModels.SqlServer`;
   - в поле Solution name (Имя решения) укажите значение `Chapter03`.
2. В файле созданного проекта добавьте строгую обработку предупреждений и ссылки на пакеты для провайдера БД SQL Server и поддержки режима разработки EF Core:

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.0" />
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Design" Version="8.0.0">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;
        buildtransitive</IncludeAssets>
    </PackageReference>
  </ItemGroup>

</Project>
```

В копилку знаний: если вам не знаком процесс управления ресурсами в пакетах типа Microsoft.EntityFrameworkCore.Design, посетите веб-страницу https://learn.microsoft.com/ru-ru/nuget/consume-packages/package-references-in-project-files#controlling-dependency-assets.

3. Удалите файл `Class1.cs`.
4. Выполните сборку проекта `Northwind.Common.EntityModels.SqlServer`.
5. Откройте терминал и перейдите в папку `Northwind.Common.EntityModels.SqlServer`.

На следующем шаге предполагается, что в строке подключения к БД для локального SQL Server настроена встроенная проверка подлинности Windows. При необходимости измените этот параметр, если вы пользуетесь Azure SQL Database или Azure SQL Edge, указав имя пользователя и пароль.

6. В терминале создайте модели классов сущностей для всех таблиц:

```
dotnet ef dbcontext scaffold "Data Source=.;Initial
Catalog=Northwind;Integrated Security=true;TrustServerCertificate=True;"
Microsoft.EntityFrameworkCore.SqlServer --namespace Northwind.EntityModels
--data-annotations
```

   Обратите внимание на следующее.
   - Выполняемая команда: `dbcontext scaffold`.
   - Строка подключения: `"Data Source=.;Initial Catalog=Northwind;Integrated Security=true;TrustServerCertificate=True;"`.
   - Провайдер БД: `Microsoft.EntityFrameworkCore.SqlServer`.
   - Пространство имен для сгенерированных классов: `--namespace Northwind.EntityModels`.
   - Параметр для использования аннотаций данных наряду с Fluent API: `--data-annotations`.
7. Обратите внимание, что было сгенерировано 28 классов, начиная с `AlphabeticalListOfProduct.cs` и заканчивая `Territory.cs`.
8. В файле `NorthwindContext.cs` удалите метод `OnConfiguring` вместе с предупреждением и строкой подключения.
9. В файле `Customer.cs` инструмент `dotnet-ef` правильно определил, что столбец `CustomerId` — это первичный ключ, значение которого ограничено пятью символами. Однако нужно, чтобы значения были представлены символами в верхнем регистре. Для этого добавьте показанное ниже регулярное выражение, проверяющее значение первичного ключа. В качестве значений будут допускаться только прописные латинские буквы:

```
[Key]
[StringLength(5)]
[RegularExpression("[A-Z]{5}")]
public string CustomerId { get; set; } = null!;
```

Создание библиотеки классов для контекста данных с помощью SQL Server

Далее вы перенесете модель контекста, представляющую БД, в отдельную библиотеку классов.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта Class Library (Библиотека классов);
   - в поле Project name (Имя проекта) укажите значение `Northwind.Common.DataContext.SqlServer`;
   - в поле Solution name (Имя решения) укажите значение `Chapter03`;
   - в файле созданного проекта добавьте ссылку на проект `EntityModels` и подключите пакет провайдера данных EF Core для SQL Server:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.0" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include="..\Northwind.Common.EntityModels
➥ .SqlServer\Northwind.Common.EntityModels.SqlServer.csproj" />
</ItemGroup>
```

Внимание! Ссылка на проект в файле не должна содержать символы разрыва строки.

2. В проекте `Northwind.Common.DataContext.SqlServer` удалите файл `Class1.cs`.
3. Выполните сборку проекта `Northwind.Common.DataContext.SqlServer`.
4. Переместите файл `NorthwindContext.cs` из каталога `Northwind.Common.EntityModels.SqlServer` в проект/папку `Northwind.Common.DataContext.SqlServer`.
5. В проекте `Northwind.Common.DataContext.SqlServer` добавьте класс `NorthwindContextExtensions.cs` и измените его, написав метод расширения, добавляющий контекст БД Northwind в коллекцию служб зависимостей:

```
using Microsoft.Data.SqlClient; // Класс SqlConnectionStringBuilder
using Microsoft.EntityFrameworkCore; // Метод UseSqlServer
using Microsoft.Extensions.DependencyInjection; // Интерфейс IServiceCollection

namespace Northwind.EntityModels;

public static class NorthwindContextExtensions
```

```
{
  /// <summary>
  /// Добавляет реализацию NorthwindContext в указанный
  /// интерфейс IServiceCollection.
  /// Использует провайдер БД SqlServer.
  /// </summary>
  /// <param name="services"> Коллекция сервисов.</param>
  /// <param name="connectionString"> Устанавливается
  /// для переопределения значения по умолчанию.</param>
  /// <returns> Интерфейс IServiceCollection, который может быть
  /// использован для добавления других сервисов.</returns>
  public static IServiceCollection AddNorthwindContext(
    this IServiceCollection services,
    string? connectionString = null)
  {
    if (connectionString == null)
    {
      SqlConnectionStringBuilder builder = new();

      builder.DataSource = ".";
      builder.InitialCatalog = "Northwind";
      builder.TrustServerCertificate = true;
      builder.MultipleActiveResultSets = true;

      // Если используется Azure SQL Edge.
      // builder.DataSource = "tcp:127.0.0.1,1433";
      // Чтобы сбои происходили быстрее.

      // По умолчанию это значение составляет 15 секунд
      builder.ConnectTimeout = 3;

      // Если используется встроенная система проверки подлинности Windows
      builder.IntegratedSecurity = true;

      // Если используется система проверки подлинности SQL Server
      // builder.UserID = Environment.GetEnvironmentVariable("MY_SQL_USR");
      // builder.Password = Environment.GetEnvironmentVariable("MY_SQL_PWD");

      connectionString = builder.ConnectionString;
    }

    services.AddDbContext<NorthwindContext>(options =>
    {
      options.UseSqlServer(connectionString);

      // Журналирование в консоль при выполнении команд EF Core
      options.LogTo(Console.WriteLine,
        new[] { Microsoft.EntityFrameworkCore
          .Diagnostics.RelationalEventId.CommandExecuting });
    },
    // Зарегистрируйте жизненный цикл как Transient во избежание проблем
    // конкурентности с проектами Blazor Server.
    contextLifetime: ServiceLifetime.Transient,
```

```
    optionsLifetime: ServiceLifetime.Transient);

    return services;
  }
}
```

6. Выполните сборку двух библиотек классов и исправьте возникшие во время компиляции ошибки.

Дельный совет: мы добавили в метод AddNorthwindContext дополнительный аргумент, позволяющий переопределять строку подключения к БД SQL Server. Это обеспечит бóльшую гибкость, например, для загрузки значений из файла конфигурации.

Свойства, вычисляемые при создании модели сущности

В EF Core 7 добавлен интерфейс `IMaterializationInterceptor`, перехватывающий данные до и после создания сущности, а также при инициализации ее свойств. Это полезно для работы с вычисляемыми значениями.

Пример: когда сервис или клиентское приложение запрашивает сущности для отображения пользователю, может возникнуть необходимость кэшировать копию сущности на определенный период времени. Для этого требуется информация о времени последнего обновления сущности. Было бы удобно, если бы эта информация автоматически генерировалась и сохранялась вместе с сущностями во время загрузки.

Для этого необходимо выполнить следующие шаги.

1. Сначала определите интерфейс с дополнительным свойством.
2. Реализуйте этот интерфейс в одном или нескольких классах моделей сущностей.
3. Создайте класс перехватчика, реализующий интерфейс `IMaterializationInterceptor`, с методом `InitializedInstance`, который будет вызываться для каждой сущности. Если сущность реализует интерфейс с дополнительным свойством, метод установит его значение.
4. Создайте экземпляр перехватчика и зарегистрируйте его в контексте данных.

Теперь реализуем это для сущностей `Employee` в БД Northwind.

1. В проекте `Northwind.Common.EntityModels.SqlServer` создайте файл `IHasLastRefreshed.cs` и определите в нем интерфейс следующим образом:

```
namespace Northwind.EntityModels;

public interface IHasLastRefreshed
{
  DateTimeOffset LastRefreshed { get; set; }
}
```

2. В том же проекте создайте файл `EmployeePartial.cs` и реализуйте в нем интерфейс, как показано ниже (выделенный фрагмент):

```
using System.ComponentModel.DataAnnotations.Schema; // [NotMapped]

namespace Northwind.EntityModels;

public partial class Employee : IHasLastRefreshed
{
  [NotMapped]
  public DateTimeOffset LastRefreshed { get; set; }
}
```

Дельный совет: добавьте код расширения, аналогичный приведенному, в отдельный файл класса разделяемой сущности. Благодаря этому вы позже сможете вновь сгенерировать инструментом dotnet-ef файл Employee.cs, не внося изменения в свой код.

3. В проекте `Northwind.Common.DataContext.SqlServer` создайте файл `SetLastRefreshedInterceptor.cs` и добавьте в него код перехватчика, показанный ниже:

```
// Интерфейс IMaterializationInterceptor,
// структура MaterializationInterceptionData
using Microsoft.EntityFrameworkCore.Diagnostics;

namespace Northwind.EntityModels;

public class SetLastRefreshedInterceptor : IMaterializationInterceptor
{
  public object InitializedInstance(
    MaterializationInterceptionData materializationData,
    object entity)
  {
    if (entity is IHasLastRefreshed entityWithLastRefreshed)
    {
      entityWithLastRefreshed.LastRefreshed = DateTimeOffset.UtcNow;
    }
    return entity;
  }
}
```

4. В том же проекте в файле `NorthwindContext.cs` удалите имеющийся метод `OnConfiguring`.
5. Там же создайте файл `NorthwindContextPartial.cs`. В методе `OnConfiguring` объявите и зарегистрируйте перехватчик:

```
using Microsoft.Data.SqlClient; // Класс SqlConnectionStringBuilder
using Microsoft.EntityFrameworkCore; // Класс DbContext
```

```
namespace Northwind.EntityModels;

public partial class NorthwindContext : DbContext
{
  private static readonly SetLastRefreshedInterceptor
    setLastRefreshedInterceptor = new();

  protected override void
➥ OnConfiguring(DbContextOptionsBuilder optionsBuilder)
  {
    if (!optionsBuilder.IsConfigured)
    {
      SqlConnectionStringBuilder builder = new();

      builder.DataSource = ".";
      builder.InitialCatalog = "Northwind";
      builder.TrustServerCertificate = true;
      builder.MultipleActiveResultSets = true;

      // Чтобы сбои происходили быстрее.
      // По умолчанию это значение составляет 15 секунд.
      builder.ConnectTimeout = 3;

      // Если используется встроенная система проверки подлинности Windows
      builder.IntegratedSecurity = true;

      // Если используется система проверки подлинности SQL Server
      // builder.UserID = Environment.GetEnvironmentVariable("MY_SQL_USR");
      // builder.Password = Environment.GetEnvironmentVariable("MY_SQL_PWD");

      optionsBuilder.UseSqlServer(builder.ConnectionString);
    }
    optionsBuilder.AddInterceptors(setLastRefreshedInterceptor);
  }
}
```

6. Сохраните изменения.

Тестовый проект для проверки интеграции библиотек классов

Поскольку в этой главе не планируется создание клиентского проекта с привлечением модели EF Core, мы подготовим тестовый проект, чтобы проверить правильность интеграции контекста БД и моделей сущностей.

1. Откройте решение `Chapter03` в среде разработки и создайте проект xUnit Test Project (Тестовый проект xUnit) на языке C#. Присвойте ему имя `Northwind.Common.EntityModels.Tests`.

2. В файле `Northwind.Common.EntityModels.Tests.csproj` настройте строгую обработку предупреждений и добавьте группу элементов со ссылкой на проект `Northwind.Common.DataContext.SqlServer`:

```
<ItemGroup>
  <ProjectReference Include="..\Northwind.Common.DataContext.
➥ SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
</ItemGroup>
```

Внимание! Ссылка на проект в файле не должна содержать символы разрыва строки.

3. Соберите проект `Northwind.Common.EntityModels.Tests` и восстановите его зависимости.

Модульные тесты для моделей сущностей

Качественный модульный тест обычно состоит из трех частей.

- *Подготовка* — на этом этапе объявляются и инициализируются переменные для ввода и вывода данных.
- *Действие* — выполняется тестируемый модуль, например, вызывается метод, который необходимо протестировать.
- *Проверка* — на этом этапе происходит проверка результата, обычно с использованием утверждений. Если утверждения не выполняются, тест считается неудачным. Например, при сложении 2 и 2 результат должен быть равен 4.

Теперь мы напишем несколько модульных тестов для классов `NorthwindContext` и модели сущностей.

1. Переименуйте файл `UnitTest1.cs` в `NorthwindEntityModelsTests.cs`, а затем откройте его.
2. В программе Visual Studio Code измените имя класса на `NorthwindEntityModelsTests`. (Visual Studio предложит переименовать класс при переименовании файла.)
3. Чтобы изменить класс `NorthwindEntityModelsTests`, необходимо выполнить следующие шаги:
   1) импортировать пространство имен `Northwind.EntityModels`;
   2) добавить несколько тестовых методов для проверки:
      - подключения к контексту;
      - провайдера базы данных (должен быть SQL Server);
      - того, что первый товар имеет имя `Chai`.

Пример измененного кода класса `NorthwindEntityModelsTests` показан ниже:

```
using Northwind.EntityModels;

namespace Northwind.Common.EntityModels.Tests
{
  public class NorthwindEntityModelsTests
  {
    [Fact]
    public void CanConnectIsTrue()
    {
      using (NorthwindContext db = new()) // Подготовка
      {
        bool canConnect = db.Database.CanConnect(); // Действие

        Assert.True(canConnect); // Проверка
      }
    }

    [Fact]
  public void ProviderIsSqlServer()
  {
    using (NorthwindContext db = new())
    {
      string? provider = db.Database.ProviderName;
        Assert.Equal("Microsoft.EntityFrameworkCore.SqlServer", provider);
      }
    }

    [Fact]
    public void ProductId1IsChai()
    {
      using (NorthwindContext db = new())
      {
        Product product1 = db.Products.Single(p => p.ProductId == 1);

        Assert.Equal("Chai", product1.ProductName);
      }
    }

    [Fact]
    public void EmployeeHasLastRefreshedIn10sWindow()
    {
      using (NorthwindContext db = new())
      {
        Employee employee1 = db.Employees.Single(p => p.EmployeeId == 1);

        DateTimeOffset now = DateTimeOffset.UtcNow;

        Assert.InRange(actual: employee1.LastRefreshed,
          low: now.Subtract(TimeSpan.FromSeconds(5)),
          high: now.AddSeconds(5));
      }
    }
  }
}
```

Запуск модульных тестов в Visual Studio 2022

Теперь, когда вы написали модульные тесты, их можно запустить и изучить результаты.

1. В программе Visual Studio 2022 выполните команду меню Test ▸ Run All Tests (Тесты ▸ Запуск всех тестов).
2. Согласно показанным на панели Test Explorer (Обозреватель тестов) результатам, были успешно проведены четыре теста (рис. 3.5).

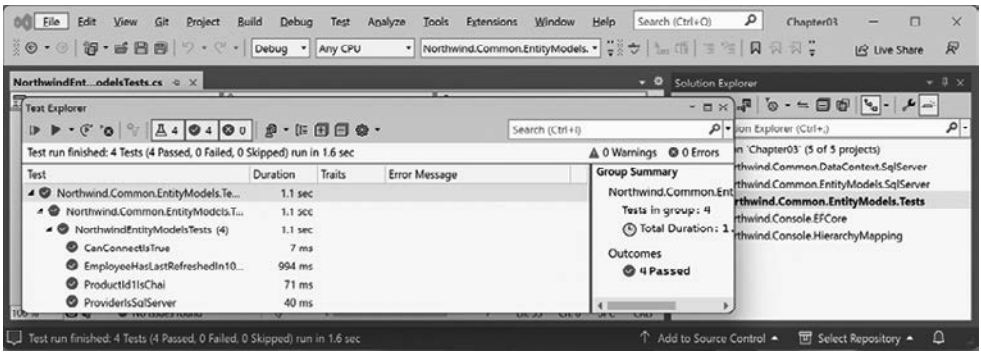


Рис. 3.5. Все модульные тесты пройдены

Запуск модульных тестов в Visual Studio Code

Теперь можно запустить модульные тесты и изучить результаты.

1. В Visual Studio Code выберите проект `Northwind.Common.EntityModels.Tests` и на панели Terminal (Терминал) запустите тесты следующей командой:

```
dotnet test
```

Если вы используете C# Dev Kit, то для выполнения сборки тестового проекта и запуска тестов перейдите в раздел Testing (Тестирование) панели Primary Side Bar (Основная боковая панель).

2. Обратите внимание, что, согласно выведенным результатам, были проведены четыре теста и все они завершились успешно.

В качестве дополнительного задания попробуйте придумать другие тесты правильности работы контекста БД и моделей сущностей.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 3.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. Для чего применяется инструмент `dotnet-ef`?
2. Какой тип данных применяется для свойства, представляющего таблицу, например для свойства `Products` в контексте данных?
3. Какой тип данных используется для свойства, представляющего отношение «один ко многим», например для свойства `Products` в сущности `Category`?
4. Расскажите о соглашении первичных ключей EF Core.
5. В чем заключаются преимущества Fluent API перед атрибутами аннотаций?
6. Для чего в типе сущности реализуется интерфейс `IMaterializationInterceptor`?

Упражнение 3.2. Практическое сравнение ADO.NET с EF Core

В решении `Chapter03` создайте консольное приложение `Ch03Ex02_ADONETvsEFCore`. Импортируйте в него библиотеку Benchmark.NET для сравнения извлечения всех товаров из БД Northwind с помощью ADO.NET (SqlClient) и EF Core.

Упражнение 3.3. Обзор решений в отношении производительности

Слой данных оказывает огромное влияние на общую производительность приложения или сервиса. См. https://learn.microsoft.com/ru-ru/ef/core/performance/.

Упражнение 3.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-3---building-entity-models-for-sql-server-using-ef-core.

Резюме

Из этой главы вы узнали:

- как выполнять простые запросы и обрабатывать результаты с помощью более объектно-ориентированного, но медленного инструмента EF Core;
- об аспектах выбора и настройки трех стратегий сопоставления иерархий типов;
- как реализовывать вычисляемые свойства при создании сущности.

В следующей главе вы узнаете, как работать с облачным нативным хранилищем данных Azure Cosmos DB.

4 Управление данными NoSQL с помощью Azure Cosmos DB

Эта глава посвящена управлению данными NoSQL с помощью Azure Cosmos DB. Вы познакомитесь с ключевыми концепциями, такими как доступные API, способы моделирования данных и настройка пропускной способности в целях снижения стоимости обслуживания. Вы создадите несколько ресурсов Cosmos DB в локальном эмуляторе и облаке Azure, а затем научитесь работать с более традиционными данными через API Core (SQL).

Основы работы с БД NoSQL

Два наиболее распространенных варианта хранения данных — это *реляционные системы управления БД* (*РСУБД*), такие как SQL Server, PostgreSQL, MySQL и SQLite, а также NoSQL-БД, например Azure Cosmos DB, Redis, MongoDB и Apache Cassandra.

Реляционные БД появились в 1970-х годах и для выполнения запросов к ним используется *язык SQL (Structured Query Language)*. В то время хранить данные было дорого, поэтому разработчики стремились минимизировать дублирование информации путем нормализации. Данные представляются в виде таблиц со строками и столбцами, а внесение изменений в их структуру после запуска в эксплуатацию бывает сложным и затратным. Масштабирование таких БД также требует значительных ресурсов.

NoSQL-БД не обязательно означают «без SQL», а, скорее, «не только SQL». Они появились в 2000-х годах с развитием Интернета и веб-технологий, унаследовав многие принципы ПО того времени. Такие базы данных обеспечивают высокую масштабируемость, производительность и гибкость. Они позволяют изменять схему данных в любое время, не накладывая жестких ограничений на их структуру, что упрощает процесс разработки.

Cosmos DB и поддерживаемые API

Azure Cosmos DB — это NoSQL-хранилище данных, поддерживающее несколько API. Главный API основан на SQL, но также доступны альтернативные варианты, такие как MongoDB, Cassandra и Gremlin.

Данные в Azure Cosmos DB хранятся в формате *atom-record-sequence (ARS)*, а взаимодействие с ними происходит через API, выбранный при создании БД.

- *API для MongoDB* поддерживает актуальные версии сетевого протокола MongoDB, позволяя клиентам работать с данными так, как если бы это была привычная MongoDB. Для переноса данных можно использовать инструменты `mongodump` и `mongorestore`. Дополнительную информацию вы найдете по адресу https://learn.microsoft.com/ru-ru/azure/cosmos-db/mongodb/introduction#how-the-api-works.
- *API для Cassandra* совместим с Cassandra Query Language (CQL) версии 4, благодаря чему взаимодействие с БД ничем не отличается от работы с классической Cassandra.
- *API Core (SQL)* рекомендуется для новых проектов, так как предоставляет гибкость и удобство запросов в стиле SQL
- *Альтернативные API* удобны для существующих проектов, поскольку позволяют работать со старыми клиентами и инструментами без необходимости обновления, а это снижает затраты на миграцию.
- *API Gremlin* подойдет для работы с графовыми данными, если между элементами данных есть сложные взаимосвязи, требующие анализа.

Дельный совет: если сложно выбрать подходящий API, лучше остановиться на Core (SQL) — он универсален и оптимален по умолчанию.

Мы будем работать именно с нативным API Core (SQL) для Cosmos DB, который позволяет запрашивать JSON-документы, используя синтаксис, похожий на SQL. Также для работы с данными используются система типов JSON и функции JavaScript.

Моделирование документов

JSON-документ, представляющий товар из БД Northwind (использованной в главе 2), при хранении в Azure Cosmos DB может выглядеть так:

```
{
  "id": "1",
  "productId": "1",
  "productName": "Chai",
  "supplier": {
    "supplierId": 1,
    "companyName": "Exotic Liquids",
```

```
    "contactName": "Charlotte Cooper",
    "Address": "49 Gilbert St.",
    "City": "London",
    "Country": "UK",
    "Phone": "(171) 555-2222"
  },
  "category": {
    "categoryId": 1,
    "categoryName": "Beverages",
    "description": "Soft drinks, coffees, teas, beers, and ales",
    "image": "https://myaccount.blob.core.windows.net/categories/beverages.png"
  },
  "quantityPerUnit": "10 boxes x 20 bags",
  "unitPrice": 18.0000,
  "unitsInStock": 39,
  "unitsOnOrder": 0,
  "reorderLevel": 10,
  "discontinued": false
}
```

В отличие от реляционной модели БД, в модели NoSQL часто *встраиваются* связанные данные, что приводит к их дублированию, это, например, категории и поставщики в нескольких записях товаров. Это полезно, когда связанные данные ограниченны. Например, у товара всегда только один поставщик и одна категория, поэтому эти отношения ограничиваются одной связью.

Если бы мы моделировали категорию и решили встроить в нее все товары, это было бы не лучшим решением, так как хранение всех данных о товарах в виде массива не поддерживает ограничений. Вместо этого можно хранить только уникальный идентификатор для каждого товара и ссылаться на данные о товаре, которые хранятся в другом хранилище.

Стоит также учитывать, как часто обновляются связанные данные. Если обновления происходят часто, то лучше избегать встраивания. Если редко, то встраивание может принести пользу. Преднамеренная денормализация данных может уменьшить количество запросов и обновлений, что снижает затраты как по времени, так и по производительности.

Аккуратная *денормализация* данных — это способ снизить количество запросов и обновлений в процессе работы, что, в свою очередь, уменьшает затраты как ресурсов, так и времени.

Используйте встраивание (денормализованные данные) в следующих случаях:

- связанные данные являются неотъемлемой частью основной сущности, например, если информация о человеке включает его адрес или номер телефона;
- отношения «один к одному» или «один ко многим», то есть связанные данные ограниченны;
- связанные данные обновляются редко;
- связанные данные часто/всегда должны быть включены в результаты запросов.

Дельный совет: денормализованные модели данных обеспечивают лучшую производительность при чтении, но могут снизить ее при записи.

Представьте, что нужно смоделировать статью и комментарии к ней на новостном сайте, где количество комментариев неограниченно и добавляться они будут особенно часто в первые часы — дни после публикации статьи. Или представьте инвестора, торгующего акциями, чьи цены обновляются регулярно.

В таких случаях вы захотите *нормализовать* связанные данные полностью или частично. Например, вы можете разместить самые популярные комментарии в верхней части списка прямо под статьей, а остальные хранить отдельно, ссылаясь на них через первичные ключи. Для долгосрочных инвестиций можно хранить такие данные, как цена покупки и цена на первый день каждого месяца, а для краткосрочных инвестиций, таких как дневная торговля, использовать ссылки на актуальную информацию.

Используйте ссылки (нормализованные данные), когда:

- отношения «один ко многим» или «многие ко многим» неограниченны;
- связанные данные обновляются часто.

Дельный совет: для нормализованных данных требуется больше запросов, что может ухудшить производительность при чтении, но повышает ее при записи.

Подробнее о моделировании документов в Azure Cosmos DB вы можете прочитать по адресу https://learn.microsoft.com/ru-ru/azure/cosmos-db/nosql/modeling-data.

Уровни согласованности

Azure Cosmos DB развернута в нескольких дата-центрах по всему миру и эластично масштабируется в зависимости от нагрузки. Данные могут храниться и реплицироваться в разных регионах, обеспечивая низкую задержку при доступе, высокую отказоустойчивость и бесперебойную работу даже в случае сбоя в одном из центров обработки данных. Для этого требуется соблюсти баланс между согласованностью, доступностью и задержкой при репликации данных по всему миру.

Для удобства разработчиков важно обеспечить полную согласованность данных. Если данные изменены в одном дата-центре, при всех последующих операциях чтения данных это изменение должно быть учтено. Наивысший уровень согласованности, известный как *линеаризуемость*, увеличивает задержку записи и сни-

жает доступность чтения, так как необходимо дождаться глобальной репликации, то есть глобального распространения обновлений данных.

Более мягкие уровни согласованности уменьшают задержку и повышают доступность, но нередко усложняют жизнь разработчика, поскольку данные могут оказаться несогласованными.

Большинство БД NoSQL поддерживают лишь два уровня согласованности: строгий и итоговый. В Azure Cosmos DB предусмотрено пять уровней, из числа которых вы можете выбрать вариант, оптимальный для конкретного проекта.

Вы самостоятельно выбираете уровень согласованности данных, и качество его работы гарантируется *соглашением об уровне услуг (Service-Level Agreement, SLA)*. Доступные уровни согласованности перечислены ниже, в порядке от самого строгого к наиболее мягкому.

- При *сильной согласованности* гарантируется, что данные во всех регионах остаются полностью синхронизированными (линеаризуемыми). Все остальные уровни считаются *мягкими*.

  Почему же не использовать сильную согласованность всегда?

  Если вы знакомы с реляционными БД, то вам должны быть знакомы *уровни изоляции транзакций*. Концептуально они похожи на уровни согласованности NoSQL. Самый строгий — `SERIALIZABLE`, а более слабые — `READ UNCOMMITTED` или `REPEATABLE READ`. Аналогично в NoSQL самая сильная согласованность снижает производительность, так как требует глобальной синхронизации данных, поэтому не стоит использовать его во всех случаях. В реальных проектах разработчики выбирают уровень согласованности, исходя из баланса *производительности и точности данных*.

  - Одни *начинают с самой сильной* и ослабляют ее, если система работает медленно.
  - Другие *начинают с самой слабой* и усиливают, если сталкиваются с проблемами несогласованных данных.

  Со временем вы научитесь быстро определять, *какой уровень лучше всего подходит для конкретной задачи*.

- *Согласованность с ограниченным устареванием* гарантирует доступность считываемых данных в регионе записи, а также монотонность считывания (значения не изменяются хаотично, а передаются в последовательном порядке). Кроме того, используется постоянный префикс (механизм, гарантирующий последовательное считывание данных) и задано ограничение на устаревание данных, выражающееся в числе версий, из-за которых чтение может отставать от записи в течение определенного времени. Например, если временной интервал составляет десять минут, а версий — три, то в течение этого времени может быть выполнено не более трех записей, прежде чем эти изменения будут учтены при чтении.

- *Согласованность сеанса* обеспечивает возможность чтения только сеансовых данных в пределах того же региона записи, соблюдаются монотонность чтения и постоянный префикс.
- *Согласованность с постоянным префиксом* гарантирует лишь порядок, в котором данные считываются.
- *Итоговая согласованность* не гарантирует совпадения порядка записи и чтения. Если запись приостанавливается, чтение будет догонять ее по мере синхронизации всех реплик. Это значит, что клиент может считывать данные, которые были актуальны до последнего обновления, то есть более старые значения, чем те, которые он считывал раньше. Probabilistic Bounded Staleness (PBS) — это показатель текущего допустимого отклонения согласованности, доступный для отслеживания на портале Azure.

Более подробную информацию об уровнях согласованности вы найдете по адресу https://learn.microsoft.com/ru-ru/azure/cosmos-db/consistency-levels.

Иерархия компонентов

Рассмотрим иерархию компонентов для Azure Cosmos DB.

- *Учетные записи* — на портале Azure можно создать до 50 учетных записей.
- *Базы данных* — в одной учетной записи может быть неограниченное количество баз данных. Мы создадим базу данных с именем `Northwind`.
- *Контейнеры* — в каждой БД можно создать неограниченное количество контейнеров. Мы создадим контейнер `Products`.
- *Разделы* — создаются в контейнере и управляются автоматически, количество неограниченно. Разделы могут быть логическими и физическими. *Логический раздел* включает элементы с одинаковым ключом раздела и определяет область действия транзакций. Несколько логических разделов могут быть сопоставлены с *физическим разделом*. Небольшим контейнерам достаточно одного физического раздела. Физическими разделами не нужно управлять, так как это происходит автоматически. Главное — подобрать ключ раздела, поскольку именно он определяет, какие элементы будут храниться в логическом разделе.
- *Элементы* — сущности, хранимые в контейнере. В качестве примера мы добавим элементы, представляющие товары, такие как чай `Chai`.

Термин «элемент» в API Core (SQL) обозначает JSON-документ, но также может использоваться и в других API. В них встречаются более специфические термины:

- в Cassandra — *строка*;
- в MongoDB — *документ*;
- в графовых БД, таких как Gremlin, — *вершины* и *ребра*.

Обеспечение пропускной способности

Пропускная способность в Azure Cosmos DB измеряется в единицах запросов в секунду (ЕЗ/с), где единица запроса (ЕЗ) соответствует стоимости выполнения `GET`-запроса для документа размером 1 Кбайт с уникальным идентификатором. На операции создания, обновления и удаления данных уходит больше ЕЗ: так, запрос данных может обойтись в 46,54 ЕЗ, а удаление — в 14,23 ЕЗ.

Пропускная способность определяется заранее, но при необходимости ее можно увеличивать/уменьшать с шагом в 100 ЕЗ/с. Стоимость использования пропускной способности в Azure Cosmos DB рассчитывается и соответствующий счет выставляется каждый час.

Стоимость использования пропускной способности в единицах ЕЗ/с можно узнать с помощью свойства RequestCharge. Подробнее — по ссылке https://learn.microsoft.com/ru-ru/azure/cosmos-db/sql/find-request-unit-charge. Данное значение мы будем анализировать во всех примерах в книге.

Пропускная способность необходима для выполнения CRUD-операций (создание, чтение, обновление и удаление). Предварительно следует оценить необходимую пропускную способность, учитывая количество операций, которое предстоит обрабатывать в течение года. В частности, на коммерческих сайтах пропускную способность может потребоваться увеличить в праздничные дни, такие как День Победы в России или День холостяков в Китае.

Настройка пропускной способности может быть выполнена на уровне контейнера или БД, и тогда параметры будут общими для всех контейнеров. Пропускная способность будет распределена между разделами контейнера поровну.

Если ресурсы пропускной способности исчерпаны, Cosmos DB начнет ограничивать скорость запросов и вам потребуется повторно отправлять запросы через некоторое время. Решить проблему можно автоматизированным путем с помощью .NET SDK для Cosmos DB, который считывает заголовок ответа `retry-after` и повторно отправляет запрос по истечении указанного времени.

На портале Azure можно настроить пропускную способность в вилке от 400 до 250 000 ЕЗ/с. Минимальный объем 400 ЕЗ/с на момент написания книги оценивался примерно в 35 долларов США в месяц. Также необходимо оплачивать стоимость хранения данных, которая зависит от объема хранилища (к примеру, 5 долларов США за несколько гигабайтов).

На бесплатном аккаунте Cosmos DB предоставляется не более 1000 ЕЗ/с и 25 Гбайт пространства в хранилище. Рассчитать необходимую пропускную способность можно с помощью калькулятора по ссылке https://cosmos.azure.com/capacitycalculator/.

Факторы, влияющие на затраты пропускной способности:

- *размер элемента* — документ размером 2 Кбайт обойдется в два раза дороже, чем документ размером 1 Кбайт;
- *индексированные свойства* — индексирование всех свойств элемента обходится дороже, чем индексирование только подмножества свойств;
- *согласованность* — строгая согласованность обойдется в два раза дороже, чем слабая;
- *сложность запросов* — количество предикатов (фильтров), количество результатов, количество пользовательских функций, проекций, размер набора данных и т. д. — все это увеличивает затраты пропускной способности.

Стратегии разбиения

Грамотный выбор ключа раздела позволяет Cosmos DB эффективно масштабироваться, поддерживать высокую производительность запросов и транзакционных операций. Ключ раздела задается на этапе создания контейнера и не может быть изменен в дальнейшем.

Подходящий ключ раздела равномерно распределяет нагрузку по БД, предотвращая появление «горячих» разделов — перегруженных участков, которые обрабатывают больше запросов, чем остальные. В идеале ключ должен быть связан с предметной областью и часто использоваться в запросах. Например, в США таким ключом может быть номер социального страхования. Однако ключ раздела не обязательно должен быть уникальным. Его значение объединяется с идентификатором элемента, создавая уникальный ключ элемента.

Разделы в Cosmos DB создаются и управляются автоматически. Если нагрузка растет, система динамически выделяет новые разделы, не влияя на работоспособность приложений. Каждый раздел поддерживает объем до 20 Гбайт, после чего Cosmos DB автоматически разбивает его на более мелкие части.

Ключ раздела должен обладать следующими свойствами:

- *высокая кардинальность* — чтобы данные распределялись по разделам равномерно;
- *равномерное распределение запросов* — чтобы избежать перегрузки отдельных разделов;
- *равномерное распределение хранилища* — чтобы эффективно использовать емкость БД.

Такой подход позволяет поддерживать баланс нагрузки и высокую производительность хранилища.

Проектирование хранилищ данных

В реляционных БД схемы жестко фиксированы и не обладают должной гибкостью. В БД Northwind все товары относятся к продуктам питания, поэтому структура меняется незначительно. Но если разрабатывать коммерческую систему для компании, торгующей одеждой, электроникой, книгами и прочими разнообразными товарами, лучше использовать полуструктурированное хранилище данных. Например:

- одежда: размеры (S, M, L, XL), производитель, цвет;
- обувь: размеры (7, 8, 9), производитель, цвет;
- телевизоры: диагональ (40", 52"), тип экрана (OLED, LCD), производитель;
- книги: количество страниц, автор, издательство.

В Azure Cosmos DB отсутствует жесткая схема, поэтому можно с легкостью добавлять новые типы товаров с разными свойствами, создавая записи с нужной структурой. В коде в этой главе это будет продемонстрировано наглядно.

Перенос данных в Cosmos DB

Инструмент миграции данных Azure Cosmos DB с открытым исходным кодом позволяет импортировать данные из различных источников, включая хранилище таблиц Azure, SQL-БД, MongoDB, текстовые файлы (JSON, CSV), HBase и др.

В книге мы не будем использовать этот инструмент. Если он вам понадобится, ознакомиться с ним можно по адресу https://github.com/Azure/azure-documentdb-datamigrationtool.

На этом с теорией пока все. Давайте перейдем к практике и посмотрим, как создавать ресурсы Cosmos DB.

Создание ресурсов Cosmos DB

Чтобы начать работу с Azure Cosmos DB, сначала необходимо создать ресурсы. Это можно сделать вручную на портале Azure или с помощью ПО .NET SDK. Ресурсы, развернутые в облаке, оплачиваются, за исключением пробных и бесплатных учетных записей.

Можно также развернуть Azure Cosmos DB локально в бесплатном эмуляторе. На момент написания книги эмуляция Azure Cosmos DB официально поддерживалась только в Windows. В операционной системе Linux доступна предварительная версия эмулятора, а в macOS Azure Cosmos DB можно эмулировать на виртуальной машине с Windows.

Эмуляция ресурсов Azure Cosmos DB в Windows

Если у вас нет компьютера с ОС Windows, прочитайте этот раздел, не выполняя практические шаги. В следующем разделе вы создадите ресурсы Azure Cosmos DB, воспользовавшись порталом Azure.

Сейчас мы настроим эмулятор Azure Cosmos DB для ОС Windows и создадим ресурсы Azure Cosmos DB, такие как БД и контейнер.

1. Загрузите и установите последнюю версию эмулятора Azure Cosmos DB на свой Windows-компьютер по ссылке (прямая ссылка на дистрибутив в формате MSI) https://aka.ms/cosmosdb-emulator.

На момент написания книги актуальная версия эмулятора — 2.14.12, выпущенная 20 марта 2023 года. Предыдущие версии эмулятора больше не поддерживаются разработчиками. Если у вас установлена старая версия, удалите ее и установите актуальную.

2. Запустите эмулятор Azure Cosmos DB.
3. Пользовательский интерфейс *эмулятора Azure Cosmos DB* должен открыться автоматически. В противном случае запустите браузер и перейдите по адресу https://localhost:8081/_explorer/index.html.
4. Обратите внимание, что эмулятор Azure Cosmos DB работает локально на порте 80. Для безопасного подключения к сервису потребуется первичный ключ (поле Primary Key (Первичный ключ)) (рис. 4.1).
5. На панели навигации в левой части интерфейса перейдите на вкладку Explorer (Обозреватель) и нажмите кнопку New Container (Новый контейнер).

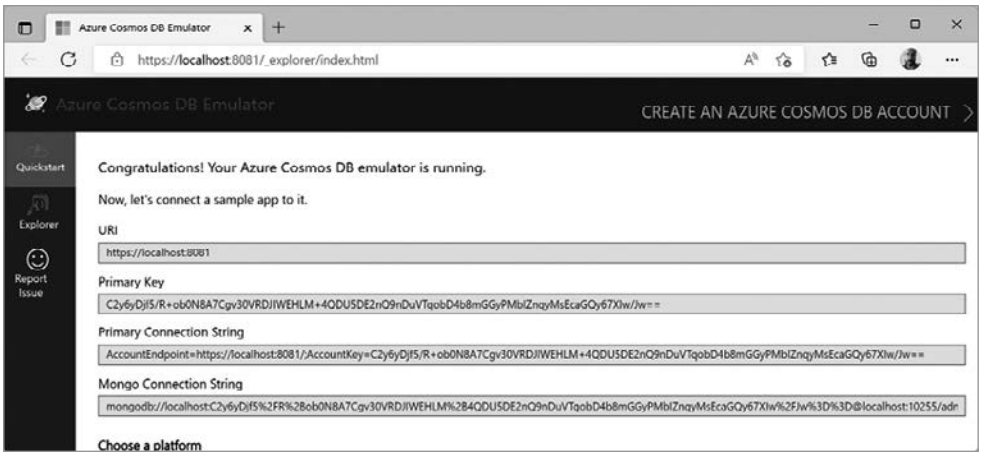


Рис. 4.1. Пользовательский интерфейс эмулятора Azure Cosmos DB в операционной системе Windows

По умолчанию в эмуляторе Azure Cosmos DB используется одинаковый первичный ключ. Чтобы указать собственное значение, запустите эмулятор из терминала с параметром /key. Подробнее об этом можно узнать по адресу https://learn.microsoft.com/ru-ru/azure/cosmos-db/emulator-windows-arguments.

6. Выполните следующие шаги.
   - Установите переключатель Database id (Идентификатор БД) в положение Create new (Создать новую) и введите значение `Northwind` в поле ниже.
   - Установите флажок Share throughput across containers (Разделить пропускную способность между контейнерами).
   - Установите переключатель Database throughput (Пропускная способность БД) в положение Autoscale (Автомасштабирование).
   - Присвойте параметру Database max RU/s (Предельное значение ЕЗ/с БД) значение `4000`. Так мы настраиваем пропускную способность на 400 ЕЗ/с с автоматическим масштабированием в случае необходимости до 4000 ЕЗ/с.
   - В поле Container id (Идентификатор контейнера) введите слово `Products`.
   - В поле Partition key (Ключ раздела) укажите значение `/productId`.
7. Нажмите кнопку OK.
8. В дереве слева разверните БД Northwind, контейнер Products и выберите строку Items (Элементы) (рис. 4.2).

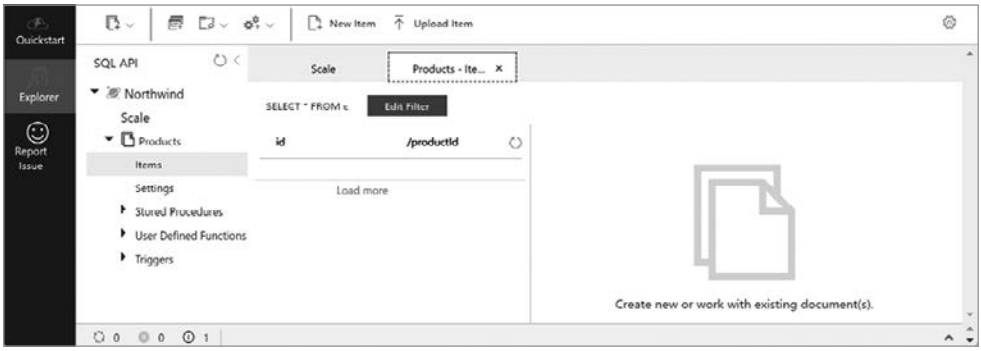


Рис. 4.2. Пустые элементы в контейнере Products БД Northwind

9. На панели инструментов нажмите кнопку New Item (Новый элемент).
10. Замените содержимое в окне редактора на JSON-код товара `Chai`, показанный ниже:

```
{
  "productId": 1,
  "productName": "Chai",
  "supplier": {
    "supplierId": 1,
```

```
    "companyName": "Exotic Liquids",
    "contactName": "Charlotte Cooper",
    "Address": "49 Gilbert St.",
    "City": "London",
    "Country": "UK",
    "Phone": "(171) 555-2222"
  },
  "category": {
    "categoryId": 1,
    "categoryName": "Beverages",
    "description": "Soft drinks, coffees, teas, beers, and ales"
  },
  "quantityPerUnit": "10 boxes x 20 bags",
  "unitPrice": 18,
  "unitsInStock": 39,
  "unitsOnOrder": 0,
  "reorderLevel": 10,
  "discontinued": false
}
```

11. На панели инструментов нажмите кнопку Save (Сохранить). Обратите внимание на дополнительные свойства, автоматически добавляемые к элементам, в частности `id`, `_etag` и `_ts`, как показано ниже в JSON-коде:

```
{
  "productId": 1,
  "productName": "Chai",
  "supplier": {
    "supplierId": 1,
    ...
  "reorderLevel": 10,
  "discontinued": false,
  "id": "2ad4c71d-d0e4-4ebd-a146-bcf052f8d7d6",
  "_rid": "bmAuAJ9o6I8BAAAAAAAAAA==",
  "_self": "dbs/bmAuAA==/colls/bmAuAJ9o6I8=/docs/bmAuAJ9o6I8BAAAAAAAAAA==/",
  "_etag": "\"00000000-0000-0000-8fc2-ec4d49ea01d8\"",
  "_attachments": "attachments/",
  "_ts": 1656952035
}
```

12. Нажмите кнопку New Item (Новый элемент).
13. Замените содержимое в окне редактора на JSON-код товара Chang, показанный ниже:

```
{
  "productId": 2,
  "productName": "Chang",
  "supplier": {
    "supplierId": 1,
```

```
    "companyName": "Exotic Liquids",
    "contactName": "Charlotte Cooper",
    "Address": "49 Gilbert St.",
    "City": "London",
    "Country": "UK",
    "Phone": "(171) 555-2222"
  },
  "category": {
    "categoryId": 1,
    "categoryName": "Beverages",
    "description": "Soft drinks, coffees, teas, beers, and ales"
  },
  "quantityPerUnit": "24 - 12 oz bottles",
  "unitPrice": 19,
  "unitsInStock": 17,
  "unitsOnOrder": 40,
  "reorderLevel": 25,
  "discontinued": false
}
```

14. Нажмите кнопку Save (Сохранить).
15. Нажмите кнопку New Item (Новый элемент).
16. Замените содержимое в окне редактора на JSON-код товара `Aniseed Syrup`, показанный ниже:

```
{
  "productId": 3,
  "productName": "Aniseed Syrup",
  "supplier": {
    "supplierId": 1,
    "companyName": "Exotic Liquids",
    "contactName": "Charlotte Cooper",
    "Address": "49 Gilbert St.",
    "City": "London",
    "Country": "UK",
    "Phone": "(171) 555-2222"
  },
  "category": {
    "categoryId": 2,
    "categoryName": "Condiments",
    "description": "Sweet and savory sauces, relishes, spreads, and seasonings"
  },
  "quantityPerUnit": "12 - 550 ml bottles",
  "unitPrice": 10,
  "unitsInStock": 13,
  "unitsOnOrder": 70,
  "reorderLevel": 25,
  "discontinued": false
}
```

17. Нажмите кнопку Save (Сохранить).
18. Щелкните на первом элементе в списке. Обратите внимание, что каждому элементу автоматически присвоены значения `GUID` для свойства `id` (рис. 4.3).

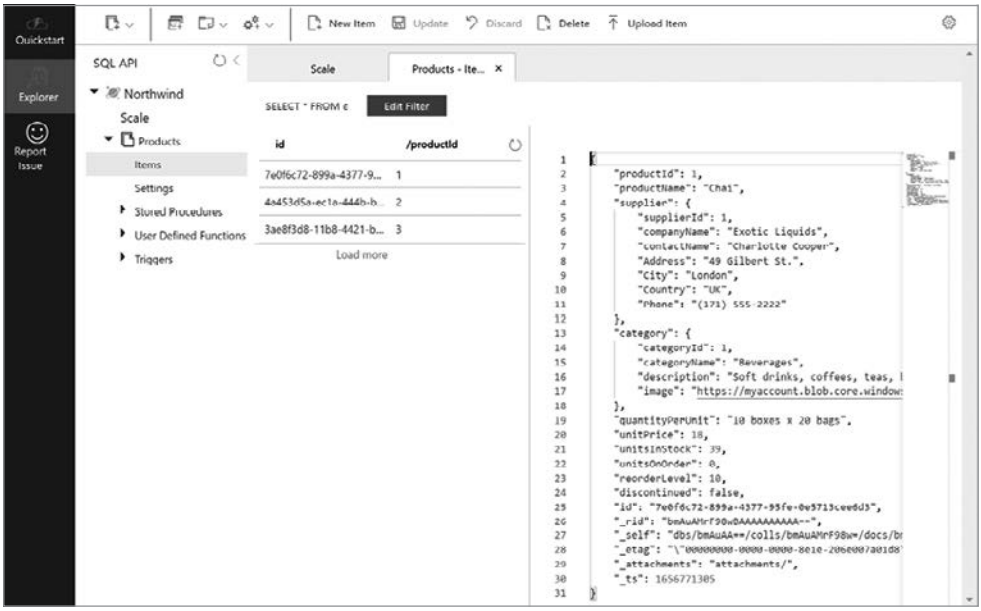


Рис. 4.3. Сохраненный JSON-код в эмуляторе Azure Cosmos DB

19. На панели инструментов нажмите кнопку New SQL Query (Новый SQL-запрос). Вы увидите запрос по умолчанию — `SELECT * FROM c`.
20. В теле запроса укажите команду для возврата всех товаров, поставляемых компанией Exotic Liquids. Для этого на панели инструментов нажмите кнопку Execute Query (Выполнить запрос) и укажите запрос, показанный на рис. 4.4 и в листинге ниже:

```
SELECT * FROM c WHERE c.supplier.companyName = "Exotic Liquids"
```

Обратите внимание, включены ли в массив результатов все три товара этого поставщика.

Ключевые слова нечувствительны к регистру, поэтому команда WHERE интерпретируется так же, как Where или where. А вот имена свойств чувствительны к регистру, поэтому значение CompanyName отличается от companyName и по такому запросу не будет найдено данных, соответствующих условиям.

Рис. 4.4. Запрос, возвращающий все товары поставщика Exotic Liquids

21. Измените текст запроса. Попробуйте вернуть все товары из категории 2:

    ```
    SELECT * FROM c WHERE c.category.categoryId = 2
    ```

22. Выполните запрос и обратите внимание, что в массив результатов попал лишь один товар.

Создание ресурсов Azure Cosmos DB на портале Azure

Если вы предпочитаете использовать только эмулятор Azure Cosmos DB во избежание затрат, то смело пропускайте этот раздел или прочитайте его, не выполняя практические шаги.

Теперь мы воспользуемся облачным порталом Azure и создадим ресурсы Azure Cosmos DB, такие как учетная запись, БД и контейнер.

1. Если у вас нет учетной записи Azure, вы можете бесплатно зарегистрироваться по адресу https://azure.microsoft.com/free/.
2. Перейдите на сайт Azure по адресу https://portal.azure.com/ и авторизуйтесь.
3. На главной странице сервиса Azure щелкните на ссылке + Create a resource (+ Создать ресурс).
4. В разделе Create a resource (Создать ресурс) нажмите кнопку Azure Cosmos DB, а затем на открывшейся странице нажмите кнопку Create (Создать).
5. На странице Which API best suits your workload? (Какой API лучше всего подходит для вашей рабочей нагрузки?) обратите внимание на описание в поле Azure

Cosmos DB for NoSQL (Azure Cosmos DB для NoSQL): Ядро Azure Cosmos DB или нативный API для работы с документами поддерживает быструю и удобную разработку с использованием языка запросов SQL, а также клиентских библиотек для .NET, JavaScript, Python и Java. Нажмите кнопку Create (Создать).

6. На вкладке Basics (Основы) сделайте следующее.
   - Выберите подписку (Subscription). Моя называется `Pay-As-You-Go`.
   - Выберите группу ресурсов (Resource Group) или создайте новую. Я использовал название `apps-services-book`.
   - Укажите имя учетной записи Azure Cosmos DB (Account Name). У меня — `apps-services-book`.
   - Выберите регион (Location). Я выбрал Великобританию, так как эта страна ко мне ближе всего.
   - Настройте режим пропускной способности, выбрав в раскрывающемся списке Capacity mode (Режим пропускной способности) пункт Provisioned throughput (Предоставляемая пропускная способность).
   - Параметру Apply Free Tier Discount (Применить бесплатный уровень обслуживания) присвойте значение Do not apply (Не применять).
   - Установите флажок Limit total account throughput (Ограничить общую пропускную способность учетной записи).

Дельный совет: на данном этапе применяйте бесплатный уровень обслуживания, только если не планируете создавать дополнительные учетные записи в текущей подписке. В противном случае рекомендую сохранить скидку для учетной записи, которая будет использоваться для реальных (а не пробных, рассматриваемых в данной книге) проектов. По скидке бесплатно доступно 1000 ЕЗ/с и 25 Гбайт хранилища, но только для одной учетной записи в подписке. Абонентская плата составит около 64 долларов в месяц.

7. Нажмите кнопку Next: Global Distribution (Далее: глобальное распределение) и просмотрите параметры, оставив их без изменений.
8. Нажмите кнопку Next: Networking (Далее: работа с сетью). Изучите настройки, но не меняйте их.
9. Нажмите кнопку Next: Backup Policy (Далее: политика резервного копирования) и ознакомьтесь с настройками, не изменяя их.
10. Нажмите кнопку Next: Encryption (Далее: шифрование) и изучите настройки, оставив их без изменений.
11. Нажмите кнопку Review + create (Обзор + создать).
12. Обратите внимание на сообщение Validation Success (Валидация прошла успешно), просмотрите сводку, а затем нажмите кнопку Create (Создать).
13. Дождитесь, пока завершится развертывание (это займет несколько минут).

14. Нажмите кнопку Go to resource (Перейти к ресурсу). Скорее всего, вы перейдете на страницу Quick Start (Быстрый старт) с шагами по созданию контейнера и т. д., в зависимости от того, новая у вас учетная запись Azure Cosmos DB или нет.
15. На навигационной панели слева перейдите на вкладку Overview (Обзор) и посмотрите информацию о своей учетной записи Azure Cosmos DB. Пример показан на рис. 4.5.

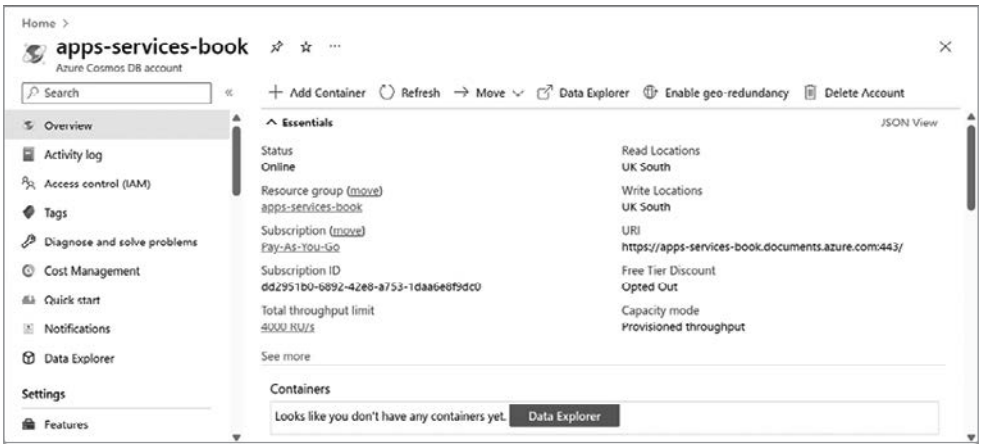


Рис. 4.5. Страница обзора учетной записи Azure Cosmos DB

16. На панели слева на вкладке Settings (Настройки) нажмите кнопку Keys (Ключи) и скопируйте значения из полей URI и PRIMARY KEY (Первичный ключ) (рис. 4.6). Их нужно будет указывать в коде для работы с текущей учетной записью Azure Cosmos DB.

Рис. 4.6. Ключи для работы с учетной записью Azure Cosmos DB

Дельный совет: в отличие от первичного ключа в эмуляторе Cosmos DB, универсального для всех разработчиков, первичные ключи Azure Cosmos DB уникальны и должны храниться в секрете. Я удалил учетную запись Cosmos DB, использованную при написании этой главы, поэтому ключи на снимке экрана выше больше недействительны.

17. На навигационной панели слева нажмите кнопку Data Explorer (Обозреватель данных) и, если появится тематический видеоролик, закройте его.
18. На панели инструментов нажмите кнопку New Container (Новый контейнер).
19. Выполните шаги, приведенные в подразделе «Эмуляция ресурсов Azure Cosmos DB в Windows», начиная с шага 6, с момента ввода информации о создаваемом контейнере.

Создание ресурсов Azure Cosmos DB с помощью приложения .NET

Далее мы создадим консольное приложение для управления ресурсами Azure Cosmos DB, с которым можно работать либо в локальном эмуляторе, либо в облаке, в зависимости от используемых URI-идентификатора и первичного ключа.

1. В редакторе кода создайте консольное приложение со следующими настройками:
   - выберите шаблон проекта Console App (Консольное приложение);
   - в поле Solution name (Имя решения) укажите значение `Chapter04`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.CosmosDb.SqlApi`.
2. В файл проекта добавьте строгую обработку предупреждений, подключите пакет Azure Cosmos и добавьте ссылку на проект контекста данных Northwind, созданный в главе 3. Также статически импортируйте класс `System.Console` на глобальном уровне:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Azure.Cosmos" Version="3.37.0" />
  </ItemGroup>
```

```
  <ItemGroup>
    <ProjectReference Include="..\..\Chapter03\Northwind.Common.DataContext
      .SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
    </ItemGroup>

  <ItemGroup>
    <Using Include="System.Console" Static="true" />
  </ItemGroup>

</Project>
```

3. Выполните сборку проекта `Northwind.CosmosDb.SqlApi` в терминале командой `dotnet build`.

Внимание! Сборка в программе Visual Studio 2022 проекта за пределами текущего решения приводит к следующей ошибке: NU1105 Не удается прочитать сведения о проекте для 'C:\apps-services-net8\Chapter03\Northwind.Common.DataContext.SqlServer\Northwind.Common.DataContext.SqlServer.csproj'. Если вы используете Visual Studio, проблема может быть связана с тем, что проект выгружен или не является частью текущего решения.

В этом случае следует собирать проект командой `dotnet build` в терминале. Вы можете щелкнуть правой кнопкой мыши на названии проекта на панели Solution Explorer (Обозреватель решений) и выбрать пункт Open in Terminal (Открыть в терминале) в контекстном меню.

4. Создайте файл класса `Program.Helpers.cs`, удалите из него весь код и добавьте операторы для определения разделяемого класса `Program` с методом, который выводит в консоль заголовок раздела в нужном формате. См. ниже:

```
// Определение выполняется в пустом пространстве имен по умолчанию,
// поэтому оно объединяется с разделяемым классом Program,
// сгенерированным SDK

partial class Program
{
  static void SectionTitle(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkYellow;
    WriteLine("*");
    WriteLine($"* {title}");
    WriteLine("*");
    ForegroundColor = previousColor;
  }
}
```

5. Создайте файл класса `Program.Methods.cs`.
6. Добавьте в созданный файл оператор импорта пространства имен для работы с Azure Cosmos. Добавьте в класс `Program` метод, который инициализирует клиента Cosmos и создает БД Northwind с контейнером Products в эмуляторе/облаке:

```
using Microsoft.Azure.Cosmos; // Для класса CosmosClient и т. д.
using System.Net;             // Для перечисления HttpStatusCode

// Определение выполняется в пустом пространстве имен по умолчанию,
// поэтому оно объединяется с разделяемым классом Program,
// сгенерированным SDK.

partial class Program
{
  // To use Azure Cosmos DB in the local emulator.
  private static string endpointUri = "https://localhost:8081/";
  private static string primaryKey = "C2y6yDjf5/R+ob0N8A7Cgv30VRDJIWEHLM+
➥ 4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==";

  /*
  // Для Azure Cosmos DB в локальном эмуляторе
  private static string account = "apps-services-book";
  // используйте свою учетную запись
  private static string endpointUri = $"https://{account}.documents.azure.com:443/";
  private static string primaryKey = "LGrx7H...gZw==";
  // используйте свой ключ
  */

  static async Task CreateCosmosResources()
  {
    SectionTitle("Creating Cosmos resources");

    try
    {
      using (CosmosClient client = new(
        accountEndpoint: endpointUri,
        authKeyOrResourceToken: primaryKey))
      {
        DatabaseResponse dbResponse = await client
          .CreateDatabaseIfNotExistsAsync(
            "Northwind", throughput: 400 /* E3/c */);

        string status = dbResponse.StatusCode switch
        {
          HttpStatusCode.OK => "exists",
          HttpStatusCode.Created => "created",
          _ => "unknown"
        };
```

```
        WriteLine("Database Id: {0}, Status: {1}.",
          arg0: dbResponse.Database.Id, arg1: status);

        IndexingPolicy indexingPolicy = new()
        {
          IndexingMode = IndexingMode.Consistent,
          Automatic = true, // Элементы индексируются, если они
                            // не исключены в явном виде
          IncludedPaths = { new IncludedPath { Path = "/*" } }
        };

        ContainerProperties containerProperties = new("Products",
          partitionKeyPath: "/productId")
        {
          IndexingPolicy = indexingPolicy
        };

        ContainerResponse containerResponse = await dbResponse.Database
          .CreateContainerIfNotExistsAsync(
            containerProperties, throughput: 1000 /* RU/s */);

        status = containerResponse.StatusCode switch
        {
          HttpStatusCode.OK => "exists",
          HttpStatusCode.Created => "created",
          _ => "unknown",
        };

        WriteLine("Container Id: {0}, Status: {1}.",
          arg0: containerResponse.Container.Id, arg1: status);

        Container container = containerResponse.Container;

        ContainerProperties properties = await container.ReadContainerAsync();
        WriteLine($" PartitionKeyPath: {properties.PartitionKeyPath}");
        WriteLine($" LastModified: {properties.LastModified}");
        WriteLine(" IndexingPolicy.IndexingMode: {0}",
          arg0: properties.IndexingPolicy.IndexingMode);
           WriteLine(" IndexingPolicy.IncludedPaths: {0}",
          arg0: string.Join(",", properties.IndexingPolicy
            .IncludedPaths.Select(path => path.Path)));
        WriteLine($" IndexingPolicy: {properties.IndexingPolicy}");
      }
    }
    catch (HttpRequestException ex)
    {
      WriteLine($"Error: {ex.Message}");
      WriteLine("Hint: If you are using the Azure Cosmos Emulator then
➥ please make sure that it is running.");
    }
    catch (Exception ex)
```

```
      {
        WriteLine("Error: {0} says {1}",
          arg0: ex.GetType(),
          arg1: ex.Message);
      }
    }
  }
```

В приведенном коде обратите внимание на следующие моменты.

- В эмуляторе значения параметров endpointUri и primaryKey одинаковы для всех.
- Конструктору CosmosClient необходимы параметры endpointUri и primaryKey. Никогда не храните свой первичный ключ в исходном коде и не загружайте в публичный Git-репозиторий! Используйте переменные окружения или хранилище Azure Key Vault.
- При создании БД указываются имя и пропускная способность в единицах ЕЗ/с в секунду.
- При создании контейнера указываются имя, ключ раздела, политика индексирования и (при необходимости) нужная пропускная способность.
- В ответе на создание ресурса Azure Cosmos DB содержится код ответа HTTP (200 OK — уже существует, 201 Created — успешно создан) и данные о ресурсе, включая его идентификатор.

7. Удалите содержимое файла `Program.cs`, а затем добавьте вызов метода создания ресурсов Azure Cosmos:

```
await CreateCosmosResources();
```

8. Запустите консольное приложение и обратите внимание на результаты:

```
*
* Creating Cosmos resources
*
Database Id: Northwind, Status: exists.
Container Id: Products, Status: exists.
  PartitionKeyPath: /productId
  LastModified: 04/07/2022 11:11:31
  IndexingPolicy.IndexingMode: Consistent
  IndexingPolicy.IncludedPaths: /*
```

9. В эмуляторе Azure Cosmos DB/на портале Azure перейдите на вкладку Data Explorer (Обозреватель данных) и удалите БД Northwind. (Нужно установить указатель мыши на БД, а затем нажать кнопку с многоточием — ...) Появится запрос ввода имени БД для подтверждения удаления, поскольку эту операцию нельзя отменить.

На этом этапе обязательно нужно удалить БД Northwind. Позже в этой главе вы добавите 77 товаров из БД SQL Server Northwind в Cosmos DB Northwind. Если в контейнере Products останутся три товара, которые мы добавили ранее, то у вас возникнут проблемы.

10. Запустите консольное приложение. Поскольку БД была только что удалена, код создал ее заново, что видно в выводе ниже:

```
*
* Creating Cosmos resources
*
Database Id: Northwind, Status: created.
Container Id: Products, Status: created.
  PartitionKeyPath: /productId
  LastModified: 04/07/2022 11:11:31
  IndexingPolicy.IndexingMode: Consistent
  IndexingPolicy.IncludedPaths: /*
```

Вам не нужно удалять БД снова, потому что она и так пуста.

База данных Cosmos DB создана, и теперь с ней можно работать как в локальном эмуляторе, так и в облаке Azure. Давайте разберемся, как выполнять CRUD-операции с помощью SQL API.

Управление данными через API Core (SQL)

Наиболее распространенный интерфейс API для работы с данными в Azure Cosmos DB — Core (SQL).

Полную документацию по API Core (SQL) вы найдете по адресу https://learn.microsoft.com/ru-ru/azure/cosmos-db/nosql/.

Выполнение CRUD-операций с помощью API Cosmos SQL

В Cosmos DB с помощью SQL API над JSON-документами можно выполнять CRUD-операции, вызывая следующие распространенные перегрузки методов экземпляра класса `Microsoft.Azure.Cosmos.Container`:

- `ReadItemAsync<T>(id, partitionKey)` — получает элемент типа *T* по его уникальному идентификатору (*id*) и значению ключа раздела (*partitionKey*);

- `ReadManyItemsAsync<T>(idsAndPartitionKeys)` — получает список элементов типа *T* только для чтения по их уникальным идентификаторам и значениям ключей разделов (*idsAndPartitionKeys*);
- `CreateItemAsync(object)` — добавляет новый элемент, переданный в *object*;
- `DeleteItemAsync<T>(id, partitionKey)` — удаляет элемент типа *T* по его уникальному идентификатору (*id*) и значению ключа раздела (*partitionKey*);
- `PatchItemAsync<T>(id, partitionKey, patchOperations)` — вносит изменения в элемент типа *T* по его уникальному идентификатору (*id*) и ключу раздела (*partitionKey*), используя список обновляемых свойств (*patchOperations*);
- `ReplaceItemAsync<T>(object, id)` — заменяет существующий элемент типа *T* на новый (*object*) по его уникальному идентификатору (*id*);
- `UpsertItemAsync<T>(object, id)` — вставляет новый элемент или обновляет существующий (*object*) по уникальному идентификатору (*id*).

Эти методы предоставляют основные операции для взаимодействия с документами в контейнере Cosmos DB через SQL API.

Дельный совет: в Cosmos DB для обмена данными применяется протокол HTTP, поэтому операции Patch и Replace реализованы через соответствующие HTTP-методы PATCH и PUT. Метод PATCH более эффективен, так как передает только измененные свойства, тогда как метод PUT требует отправки всего объекта целиком, даже если изменилась лишь небольшая часть данных.

Все методы возвращают ответы с общими свойствами:

- `Resource` — элемент, который был создан, извлечен, обновлен или удален;
- `RequestCharge` — значение типа `double`, отражающее стоимость запроса в единицах ЕЗ/с;
- `StatusCode` — код ответа HTTP (например, `404`, если `ReadItemAsync<T>` не обнаружил элемент);
- `Headers` — словарь заголовков HTTP-ответа;
- `Diagnostics` — информация для диагностики;
- `ActivityId` — GUID, полезный для отслеживания активности через многоуровневые сервисы.

Теперь скопируем все товары из БД Northwind в SQL Server в Cosmos DB.

Поскольку классы сущностей EF Core для SQL Server отражают нормализованную структуру данных в SQL-БД Northwind, создадим отдельные классы для работы с Cosmos. Они будут включать встроенные связанные данные и следовать соглашениям об оболочке JSON, так как представляют собой JSON-документы.

1. В проекте `Northwind.CosmosDb.SqlApi` создайте папку `Models`.
2. В папке `Models` создайте файл класса `CategoryCosmos.cs`.

3. Удалите его содержимое и определите класс `CategoryCosmos`:

```
namespace Northwind.CosmosDb.Items;

public class CategoryCosmos
{
  public int categoryId { get; set; }
  public string categoryName { get; set; } = null!;
  public string? description { get; set; }
}
```

Мы сознательно отходим от стандартных соглашений о регистре в .NET, так как не можем динамически управлять процессом сериализации. Итоговый JSON-документ должен быть оформлен в camelCase.

4. В папке `Models` создайте файл `SupplierCosmos.cs` и измените его содержимое, определив класс `SupplierCosmos`:

```
namespace Northwind.CosmosDb.Items;

public class SupplierCosmos
{
  public int supplierId { get; set; }
  public string companyName { get; set; } = null!;
  public string? contactName { get; set; }
  public string? contactTitle { get; set; }
  public string? address { get; set; }
  public string? city { get; set; }
  public string? region { get; set; }
  public string? postalCode { get; set; }
  public string? country { get; set; }
  public string? phone { get; set; }
  public string? fax { get; set; }
  public string? homePage { get; set; }
}
```

5. В папке `Models` создайте файл класса `ProductCosmos.cs` и измените его содержимое, определив класс `ProductCosmos`:

```
namespace Northwind.CosmosDb.Items;

public class ProductCosmos
{
  public string id { get; set; } = null!;
  public string productId { get; set; } = null!;
  public string productName { get; set; } = null!;
  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; }
  public CategoryCosmos? category { get; set; }
  public SupplierCosmos? supplier { get; set; }
}
```

Дельный совет: все элементы JSON-документов в Cosmos DB должны содержать свойство id. Чтобы управлять его значением, лучше явно определить это свойство в модели. В противном случае система автоматически присвоит ему идентификатор GUID, как это происходило ранее в текущей главе при добавлении элемента вручную в разделе Data Explorer (Обозреватель данных) сайта Azure.

6. В файле `Program.Methods.cs` добавьте операторы импорта пространств имен, содержащих типы контекста и сущностей данных Northwind, классы для работы с Cosmos DB и расширения EF Core:

```
using Northwind.EntityModels;        // Для класса NorthwindContext и т. д.
using Northwind.CosmosDb.Items;      // Для класса ProductCosmos и т. д.
using Microsoft.EntityFrameworkCore; // Для метода расширения Include
```

7. В файле `Program.Methods.cs` добавьте код метода, который загружает все товары из БД Northwind SQL вместе с их категориями и поставщиками, а затем добавляет их в контейнер `Products` в Cosmos DB:

```
static async Task CreateProductItems()
{
  SectionTitle("Creating product items");
  double totalCharge = 0.0;

  try
  {
    using (CosmosClient client = new(
      accountEndpoint: endpointUri,
      authKeyOrResourceToken: primaryKey))
    {
      Container container = client.GetContainer(
        databaseId: "Northwind", containerId: "Products");

      using (NorthwindContext db = new())
      {
        if (!db.Database.CanConnect())
        {
          WriteLine("Cannot connect to the SQL Server database to " +
            " read products using database connection string: " +
            db.Database.GetConnectionString());
          return;
        }

        ProductCosmos[] products = db.Products
```

```
// Получение связанных данных для встраивания
.Include(p => p.Category)
.Include(p => p.Supplier)

// Отсев товаров с категорией или поставщиком null
// во избежание предупреждений о значении null
.Where(p => (p.Category != null) && (p.Supplier != null))

// Проектирование сущности EF Core в типы Cosmos JSON
.Select(p => new ProductCosmos
{
  id = p.ProductId.ToString(),
  productId = p.ProductId.ToString(),
  productName = p.ProductName,
  quantityPerUnit = p.QuantityPerUnit,

  // Если связанная категория равна null, сохраняется
  // значение null, иначе сохраняется категория,
  // сопоставленная с моделью Cosmos
  category = p.Category == null ? null :
    new CategoryCosmos
  {
    categoryId = p.Category.CategoryId,
    categoryName = p.Category.CategoryName,
    description = p.Category.Description
  },

  supplier = p.Supplier == null ? null :
    new SupplierCosmos
  {
    supplierId = p.Supplier.SupplierId,
    companyName = p.Supplier.CompanyName,
    contactName = p.Supplier.ContactName,
    contactTitle = p.Supplier.ContactTitle,
    address = p.Supplier.Address,
    city = p.Supplier.City,
    country = p.Supplier.Country,
    postalCode = p.Supplier.PostalCode,
    region = p.Supplier.Region,
    phone = p.Supplier.Phone,
    fax = p.Supplier.Fax,
    homePage = p.Supplier.HomePage
  },

  unitPrice = p.UnitPrice,
  unitsInStock = p.UnitsInStock,
  reorderLevel = p.ReorderLevel,
  unitsOnOrder = p.UnitsOnOrder,
  discontinued = p.Discontinued,
})
.ToArray();
```

```
        foreach (ProductCosmos product in products)
        {
          try
          {
            // Попытка прочитать элемент для проверки его существования
            ItemResponse<ProductCosmos> productResponse =
              await container.ReadItemAsync<ProductCosmos>(
              id: product.id, new PartitionKey(product.productId));

            WriteLine("Item with id: {0} exists. Query consumed {1} RUs.",
              productResponse.Resource.id, productResponse.RequestCharge);
            totalCharge += productResponse.RequestCharge;
          }
          catch (CosmosException ex)
            when (ex.StatusCode == HttpStatusCode.NotFound)
          {
            // Создание элемента, если он не существует
            ItemResponse<ProductCosmos> productResponse =
              await container.CreateItemAsync(product);

            WriteLine("Created item with id: {0}. Insert consumed {1} RUs.",
              productResponse.Resource.id, productResponse.RequestCharge);

            totalCharge += productResponse.RequestCharge;
          }
          catch (Exception ex)
          {
            WriteLine("Error: {0} says {1}",
              arg0: ex.GetType(),
              arg1: ex.Message);
          }
        }
      }
    }
  }
  catch (HttpRequestException ex)
  {
    WriteLine($"Error: {ex.Message}");
    WriteLine("Hint: If you are using the Azure Cosmos Emulator then
➥ please make sure it is running.");
  }
  catch (Exception ex)
  {
    WriteLine("Error: {0} says {1}",
      arg0: ex.GetType(),
      arg1: ex.Message);
  }

  WriteLine("Total requests charge: {0:N2} RUs", totalCharge);
}
```

8. В файле `Program.cs` закомментируйте строку, в которой вызывается метод для создания ресурсов Azure Cosmos. Затем добавьте вызов метода вставки всех товаров:

```
await CreateProductItems();
```

9. Запустите консольное приложение и обратите внимание на результаты: должно быть вставлено 77 элементов товаров (ниже показан фрагмент):

```
*
* Creating product items
*
Created item with id: 1. Insert consumed 14.29 RUs.
Created item with id: 2. Insert consumed 14.29 RUs.
Created item with id: 3. Insert consumed 14.29 RUs.
...
Created item with id: 76. Insert consumed 14.29 RUs.
Created item with id: 77. Insert consumed 14.48 RUs.
Total requests charge: 1,114.58 RUs
```

10. Запустите консольное приложение еще раз и обратите внимание на результаты: элементы товаров уже существуют, как показано в выводе ниже (фрагмент):

```
*
* Creating product items
*
Item with id: 1 exists. Query consumed 1 RUs.
Item with id: 2 exists. Query consumed 1 RUs.
Item with id: 3 exists. Query consumed 1 RUs.
...
Item with id: 76 exists. Query consumed 1 RUs.
Item with id: 77 exists. Query consumed 1 RUs.
Total requests charge: 77.00 RUs
```

11. Убедитесь, что в эмуляторе Azure Cosmos DB или в разделе Data Explorer (Обозреватель данных) сайта Azure в контейнере `Products` действительно содержится 77 элементов товаров.
12. В файле `Program.Methods.cs` добавьте метод для перечисления всех товаров в контейнере `Products`:

```
static async Task ListProductItems(string sqlText = "SELECT * FROM c")
{
  SectionTitle("Listing product items");

  try
  {
    using (CosmosClient client = new(
```

```
        accountEndpoint: endpointUri,
        authKeyOrResourceToken: primaryKey))
      {
        Container container = client.GetContainer(
          databaseId: "Northwind", containerId: "Products");

        WriteLine("Running query: {0}", sqlText);

        QueryDefinition query = new(sqlText);

        using FeedIterator<ProductCosmos> resultsIterator =
          container.GetItemQueryIterator<ProductCosmos>(query);

        if (!resultsIterator.HasMoreResults)
        {
          WriteLine("No results found.");
        }

        while (resultsIterator.HasMoreResults)
        {
          FeedResponse<ProductCosmos> products =
            await resultsIterator.ReadNextAsync();

          WriteLine("Status code: {0}, Request charge: {1} RUs.",
            products.StatusCode, products.RequestCharge);

          WriteLine($"{products.Count} products found.");

          foreach (ProductCosmos product in products)
          {
            WriteLine("id: {0}, productName: {1}, unitPrice: {2}",
              arg0: product.id, arg1: product.productName,
           arg2: product.unitPrice.ToString());

          }
        }
      }
    }
    catch (HttpRequestException ex)
    {
      WriteLine($"Error: {ex.Message}");
      WriteLine("Hint: If you are using the Azure Cosmos Emulator then
➥ please make sure it is running.");
    }
    catch (Exception ex)
    {
      WriteLine("Error: {0} says {1}",
        arg0: ex.GetType(),
        arg1: ex.Message);
    }
  }
```

13. В файле `Program.cs` добавьте операторы импорта пространств имен для работы с культурами и кодировками, установите французскую культуру, закомментируйте вызов метода создания элементов товаров, а затем добавьте вызов метода для получения списка элементов товаров:

```
using System.Globalization;  // Для класса CultureInfo
using System.Text;           // Для кодировки

OutputEncoding = Encoding.UTF8; // Для включения вывода символов евро

// Имитация французского культурного контекста для проверки вывода евро
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("fr-FR");

//await CreateCosmosResources();

//await CreateProductItems();

await ListProductItems();
```

14. Запустите консольное приложение и обратите внимание на результаты. Должно быть перечислено 77 элементов товаров (ниже показан фрагмент):

```
*
* Listing product items
*
Running query: SELECT * FROM c
Status code: OK, Request charge: 3.93 RUs.
77 products found.
id: 1, productName: Chai, unitPrice: 18,00 €
id: 2, productName: Chang, unitPrice: 19,00 €
id: 3, productName: Aniseed Syrup, unitPrice: 10,00 €
...
id: 76, productName: Lakkalikööri, unitPrice: 18,00 €
id: 77, productName: Original Frankfurter grüne Soße, unitPrice: 13,00 €
```

15. В файле `Program.Methods.cs` добавьте метод для удаления всех элементов из контейнера `Products`:

```
static async Task DeleteProductItems()
{
  SectionTitle("Deleting product items");

  double totalCharge = 0.0;

  try
  {
    using (CosmosClient client = new(
      accountEndpoint: endpointUri,
      authKeyOrResourceToken: primaryKey))
    {
```

```
        Container container = client.GetContainer(
          databaseId: "Northwind", containerId: "Products");

        string sqlText = "SELECT * FROM c";

        WriteLine("Running query: {0}", sqlText);

        QueryDefinition query = new(sqlText);

        using FeedIterator<ProductCosmos> resultsIterator =
          container.GetItemQueryIterator<ProductCosmos>(query);

        while (resultsIterator.HasMoreResults)
        {
          FeedResponse<ProductCosmos> products =
            await resultsIterator.ReadNextAsync();

          foreach (ProductCosmos product in products)
          {
            WriteLine("Delete id: {0}, productName: {1}",
              arg0: product.id, arg1: product.productName);

            ItemResponse<ProductCosmos> response =
              await container.DeleteItemAsync<ProductCosmos>(
              id: product.id, partitionKey: new(product.id));

            WriteLine("Status code: {0}, Request charge: {1} RUs.",
              response.StatusCode, response.RequestCharge);

            totalCharge += response.RequestCharge;

          }
        }
      }
    }
    catch (HttpRequestException ex)
    {
      WriteLine($"Error: {ex.Message}");
      WriteLine("Hint: If you are using the Azure Cosmos Emulator then
➥ please make sure it is running.");
    }
    catch (Exception ex)
    {
      WriteLine("Error: {0} says {1}",
        arg0: ex.GetType(),
        arg1: ex.Message);
    }

    WriteLine("Total requests charge: {0:N2} RUs", totalCharge);
  }
```

16. В файле `Program.cs` закомментируйте строку с вызовом метода для получения списка товаров. Затем добавьте строку с вызовом метода удаления товаров:

```
await DeleteProductItems();
```

17. Запустите консольное приложение и обратите внимание на результаты: должно быть удалено 77 элементов товаров (ниже показан фрагмент):

```
*
* Deleting product items
*
Running query: SELECT * FROM c
Delete id: 1, productName: Chai
Status code: NoContent, Request charge: 14.29 RUs.
...
Delete id: 77, productName: Original Frankfurter grüne Soße
Status code: NoContent, Request charge: 14.48 RUs.
Total requests charge: 1,128.87 RUs
```

18. В эмуляторе Azure Cosmos DB или в разделе Data Explorer (Обозреватель данных) сайта Azure убедитесь, что контейнер `Products` пуст.

Структура SQL-запросов

Azure Cosmos DB поддерживает SQL-запросы со следующими ключевыми словами:

- `SELECT` — выборка свойств элементов, поддерживает символ * для подстановки всех полей и `TOP` для ограничения количества результатов;
- `AS` — задание псевдонимов;
- `FROM` — определение источника данных. Например, `FROM c`, где `c` — псевдоним элементов контейнера. Можно использовать любые псевдонимы, например `FROM Items c` или `FROM p`;
- `WHERE` — фильтрация данных;
- `LIKE` — поиск по шаблону (`%` — любое количество символов, `_` — один символ, `[a-f]` или `[aeiou]` — один символ из диапазона или набора, `[^aeiou]` — исключая указанные символы);
- `IN`, `BETWEEN` — фильтрация по множеству значений или диапазону;
- `AND`, `OR`, `NOT` — логические операции;
- `ORDER BY` — сортировка результатов;
- `DISTINCT` — удаление дубликатов;
- `COUNT`, `AVG`, `SUM` и другие агрегатные функции.

Выполним SQL-запрос к контейнеру Products с помощью API Core (SQL):

```
SELECT p.id, p.productName, p.unitPrice FROM Items p
```

А теперь выполним SQL-запрос товаров.

1. В файле `Program.cs` закомментируйте вызов повторного создания элементов товаров и измените вызов метода `ListProductItems`, передав SQL-запрос, который фильтрует товары, оставляя только категорию `Beverages` с идентификатором, названием и ценой единицы товара:

```
//await CreateCosmosResources();
await CreateProductItems();
await ListProductItems("SELECT p.id, p.productName, p.unitPrice FROM
Items p WHERE p.category.categoryName = 'Beverages'");
//await DeleteProductItems();
```

2. Запустите приложение и убедитесь, что в выводе присутствуют 12 товаров из категории `Beverages`:

```
*
* Listing product items
*
Running query: SELECT p.id, p.productName, p.unitPrice FROM Items p WHERE
p.category.categoryName = 'Beverages'
Status code: OK, Request charge: 3.19 RUs.
12 products found.
id: 1, productName: Chai, unitPrice: 18
id: 2, productName: Chang, unitPrice: 19
id: 24, productName: Guaraná Fantástica, unitPrice: 4.5
id: 34, productName: Sasquatch Ale, unitPrice: 14
id: 35, productName: Steeleye Stout, unitPrice: 18
id: 38, productName: Côte de Blaye, unitPrice: 263.5
id: 39, productName: Chartreuse verte, unitPrice: 18
id: 43, productName: Ipoh Coffee, unitPrice: 46
id: 67, productName: Laughing Lumberjack Lager, unitPrice: 14
id: 70, productName: Outback Lager, unitPrice: 15
id: 75, productName: Rhönbräu Klosterbier, unitPrice: 7.75
id: 76, productName: Lakkalikööri, unitPrice: 18
```

3. В Azure Cosmos DB Emulator или в разделе Data Explorer (Обозреватель данных) сайта Azure выполните тот же SQL-запрос еще раз (рис. 4.7).
4. Нажмите кнопку Query Stats (Статистика запроса) и обратите внимание на следующую статистику:
   - стоимость запроса (3,19 ЕЗ/с);
   - количество записей (12);
   - размер выходного документа (752 байта) (рис. 4.8).

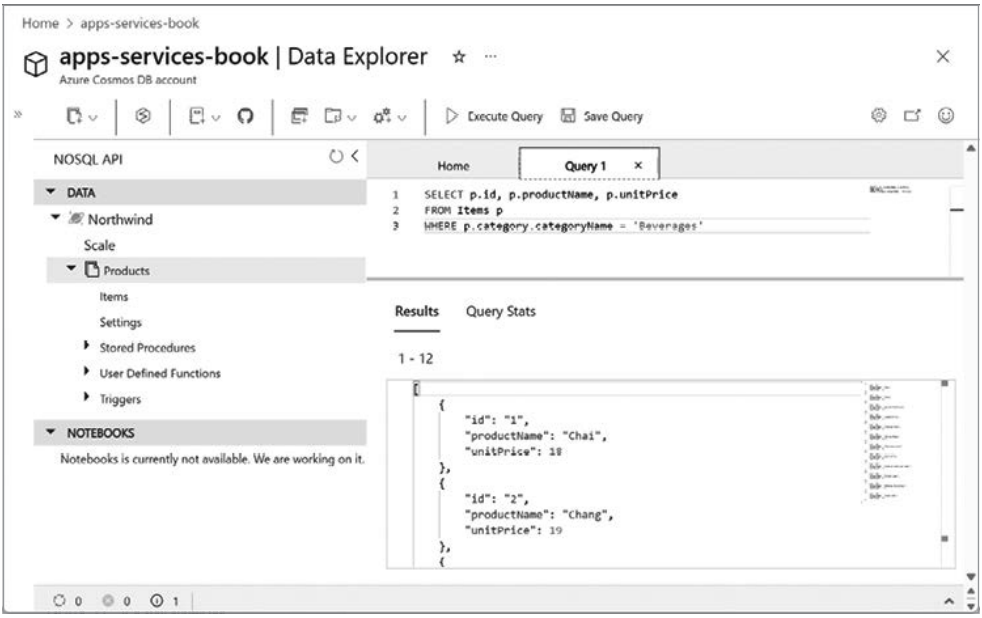


Рис. 4.7. Выполнение SQL-запроса в разделе Data Explorer

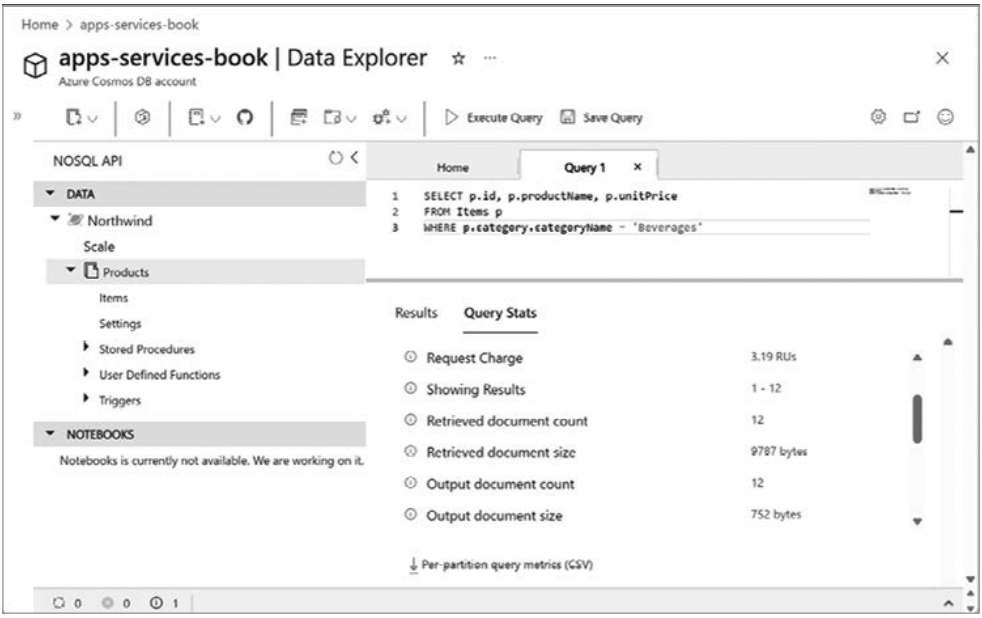


Рис. 4.8. Статистика запросов в разделе Data Explorer

Могут также пригодиться следующие статистические данные выполненного запроса:

- количество найденных документов в индексе;
- время поиска индекса;
- время загрузки и записи документа;
- время выполнения запроса.

Такой запрос позволяет быстро и эффективно фильтровать данные в Cosmos DB.

Полезные SQL-запросы в Cosmos DB

Выполните следующие запросы:

```
SELECT p.id, p.productName, p.unitPrice FROM Items p
WHERE p.unitPrice > 50

SELECT DISTINCT p.category FROM Items p

SELECT DISTINCT p.category.categoryName FROM Items p
WHERE p.discontinued = true

SELECT p.productName, p.supplier.city FROM Items p
WHERE p.supplier.country = 'Germany'

SELECT COUNT(p.id) AS HowManyProductsComeFromGermany FROM Items p
WHERE p.supplier.country = 'Germany'

SELECT AVG(p.unitPrice) AS AverageUnitPrice FROM Items p
```

Хотя строковые SQL-запросы — самый распространенный способ работы с Cosmos DB, можно также создавать на сервере хранимые объекты.

Серверное программирование в Cosmos DB

В Cosmos DB серверный код включает *хранимые процедуры*, *триггеры* и *функции, определяемые пользователем (User-Defined Functions, UDF)*, которые пишутся на языке JavaScript.

Реализация функций, определяемых пользователем

UDF-функции вызываются только в SQL-запросах и позволяют реализовать бизнес-логику, например расчет налогов.

Определим такую функцию для расчета налога с продаж.

1. В эмуляторе Azure Cosmos DB или в разделе Data Explorer (Обозреватель данных) сайта Azure создайте UDF-функцию (рис. 4.9).

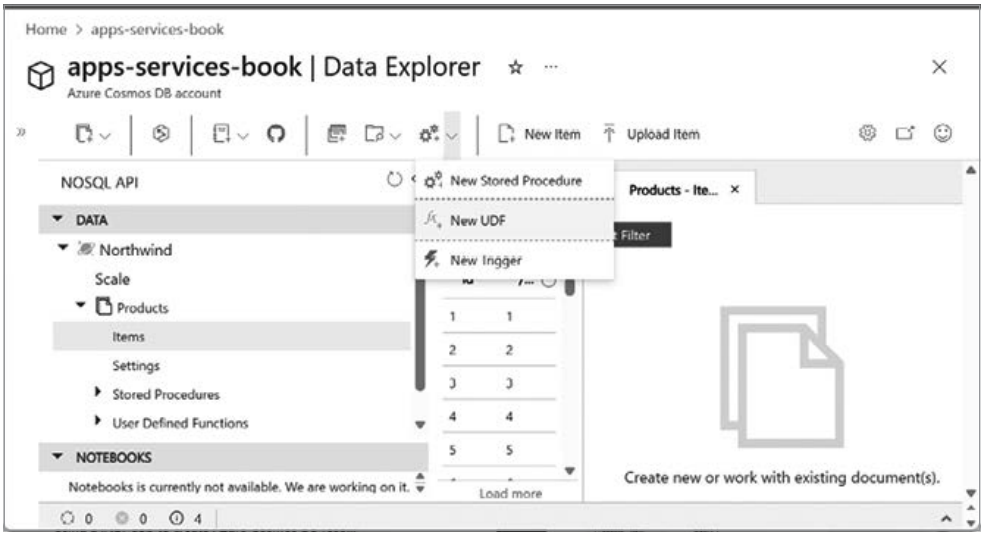


Рис. 4.9. Создание UDF-функции

2. В поле User Defined Function Id (Идентификатор функции, определяемой пользователем) введите значение `salesTax`.
3. В поле User Defined Function Body (Тело функции UDF) введите JavaScript-определение функции `salesTax`:

```
function salesTax(unitPrice){
  return unitPrice * 0.2;
}
```

4. На панели инструментов нажмите кнопку Save (Сохранить).
5. Создайте SQL-запрос, чтобы вывести цену единицы товара и рассчитанный налог для товаров дороже 100 единиц:

```
SELECT p.unitPrice cost, udf.salesTax(p.unitPrice) AS tax
FROM Items p WHERE p.unitPrice > 100
```

Обратите внимание, что ключевое слово AS для задания псевдонима в SQL-запросах указывать не обязательно, но оно улучшает читаемость.

6. Нажмите кнопку Save Query (Сохранить запрос).

Если программа использует облачные ресурсы вместо эмулятора, то по требованиям Microsoft запросы сохраняются в отдельной БД Cosmos в учетной записи Azure Cosmos DB. Примерная стоимость услуги — 0,77 доллара в сутки.

7. Выполните запрос и обратите внимание на результаты:

```
[
  {
    "cost": 123.79,
    "tax": 24.758000000000003
  },
  {
    "cost": 263.5,
    "tax": 52.7
  }
]
```

Реализация хранимых процедур

Использование хранимых процедур — единственный способ выполнять в Cosmos DB транзакции по стандарту *ACID* (*A*tomic, *C*onsistent, *I*solated, *D*urable — «атомарность, согласованность, изолированность, устойчивость»). Они позволяют объединять несколько операций в одну, которая фиксируется или отменяется целиком. Для транзакций внутри одного раздела можно воспользоваться клиентским кодом и транзакционными пакетными операциями, как описано на сайте https://learn.microsoft.com/ru-ru/azure/cosmos-db/nosql/transactional-batch.

Помимо этого, серверное выполнение кода увеличивает производительность, поскольку операции выполняются непосредственно рядом с данными.

Ранее мы создали UDF-функцию в разделе Data Explorer (Обозреватель данных) сайта Azure. Хранимую процедуру также можно определить с помощью кода.

1. В файле `Program.Methods.cs` добавьте оператор импорта пространства имен для работы с серверными объектами:

```
// Для StoredProcedureResponse и т. д.
using Microsoft.Azure.Cosmos.Scripts;
```

2. В тот же файл добавьте метод, который определяет хранимую процедуру для вставки нескольких товаров. Он должен выстраивать цепочку функций обратного вызова, пока все элементы массива не будут добавлены:

```
static async Task CreateInsertProductStoredProcedure()
{
  SectionTitle("Creating the insertProduct stored procedure");

  try
  {
    using (CosmosClient client = new(
      accountEndpoint: endpointUri,
      authKeyOrResourceToken: primaryKey))
```

```
    {
      Container container = client.GetContainer(
        databaseId: "Northwind", containerId: "Products");

      StoredProcedureResponse response = await container
        .Scripts.CreateStoredProcedureAsync(new StoredProcedureProperties
        {
          Id = "insertProduct",
          // __ — это псевдоним для методов getContext().getCollection()
          Body = """
            function insertProduct(product) {
              if (!product) throw new Error(
                "product is undefined or null.");

              tryInsert(product, callbackInsert);

              function tryInsert(product, callbackFunction) {
                var options = { disableAutomaticIdGeneration: false };

                // __ — псевдоним метода getContext().getCollection()
                var isAccepted = __.createDocument(
                  __.getSelfLink(), product, options, callbackFunction);

                if (!isAccepted)
                  getContext().getResponse().setBody(0);
              }

              function callbackInsert(err, item, options) {
                if (err) throw err;
                getContext().getResponse().setBody(1);
              }
            }
            """
        });

      WriteLine("Status code: {0}, Request charge: {1} RUs.",
        response.StatusCode, response.RequestCharge);
    }
  }
  catch (HttpRequestException ex)
  {
    WriteLine($"Error: {ex.Message}");
    WriteLine("Hint: If you are using the Azure Cosmos Emulator then
➥ please make sure it is running.");
  }
  catch (Exception ex)
  {
    WriteLine("Error: {0} says {1}",
      arg0: ex.GetType(),
      arg1: ex.Message);
  }
}
```

3. В файле `Program.cs` закомментируйте весь код и добавьте вызов нового метода:

```
await CreateInsertProductStoredProcedure();
```

4. Запустите консольное приложение и убедитесь, что хранимая процедура создана:

```
*
* Creating the insertProduct stored procedure
*
Status code: Created, Request charge: 6.29 RUs.
```

5. В файле `Program.Methods.cs` добавьте метод для выполнения хранимой процедуры:

```
static async Task ExecuteInsertProductStoredProcedure()
{
  SectionTitle("Executing the insertProduct stored procedure");

  try
  {
    using (CosmosClient client = new(
      accountEndpoint: endpointUri,
      authKeyOrResourceToken: primaryKey))
    {
      Container container = client.GetContainer(
        databaseId: "Northwind", containerId: "Products");

      string pid = "78";

      ProductCosmos product = new()
      {
        id = pid, productId = pid,
        productName = "Barista's Chilli Jam",
        unitPrice = 12M, unitsInStock = 10
      };

      StoredProcedureExecuteResponse<string>
        response = await container.Scripts
        .ExecuteStoredProcedureAsync<string>("insertProduct",
        new PartitionKey(pid), new[] { product });

      WriteLine("Status code: {0}, Request charge: {1} RUs.",
        response.StatusCode, response.RequestCharge);
    }
  }
  catch (HttpRequestException ex)
  {
    WriteLine($"Error: {ex.Message}");
    WriteLine("Hint: If you are using the Azure Cosmos Emulator then
➥ please make sure it is running.");
  }
```

```
  catch (Exception ex)
  {
    WriteLine("Error: {0} says {1}",
      arg0: ex.GetType(),
      arg1: ex.Message);
  }
}
```

6. В файле `Program.cs` закомментируйте код создания хранимой процедуры, добавьте вызов метода ее выполнения и выведите список товаров, у которых значение `productId` равно `78`:

```
//await CreateInsertProductStoredProcedure();
await ExecuteInsertProductStoredProcedure();
await ListProductItems("SELECT p.id, p.productName, p.unitPrice FROM
Items p WHERE p.productId = '78'");
```

7. Запустите приложение и убедитесь, что хранимая процедура успешно выполнена:

```
*
* Executing the insertProduct stored procedure
*
Status code: OK, Request charge: 10.23 RUs.
*
* Listing product items
*
Running query: SELECT p.id, p.productName, p.unitPrice FROM Items p WHERE
p.productId = '78'
Status code: OK, Request charge: 2.83 RUs.
1 products found.
id: 78, productName: Barista's Chilli Jam, unitPrice: €12.00
```

Очистка ресурсов Azure

После завершения работы с учетной записью Azure Cosmos DB важно удалить использованные ресурсы, иначе за них будет продолжать списываться абонентская плата. Можно удалить их по отдельности или группу ресурсов целиком, чтобы убрать все связанные компоненты.

При удалении учетной записи Azure Cosmos DB удаляются и все связанные БД и контейнеры. Для этого сделайте следующее.

1. Откройте портал Azure и перейдите в раздел All Resources (Все ресурсы).
2. В группе ресурсов `apps-services-book` выберите учетную запись Azure Cosmos DB.
3. Перейдите на вкладку Overview (Обзор) и нажмите кнопку Delete Account (Удалить учетную запись) на панели инструментов.

4. В поле Confirm the Account Name (Подтвердить имя учетной записи) укажите имя своей учетной записи.
5. Нажмите кнопку Delete (Удалить).

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 4.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. Какие пять API поддерживаются Azure Cosmos DB?
2. На каком уровне выбирается API: учетная запись, база данных, контейнер или раздел?
3. Что означает встраивание в контексте моделирования данных в Cosmos DB?
4. В каких единицах измеряется пропускная способность Cosmos DB и что такое одна единица?
5. Какой пакет нужен для работы с ресурсами Cosmos DB программным способом?
6. Какой язык используется для написания пользовательских функций и хранимых процедур в Cosmos DB API Core (SQL)?

Упражнение 4.2. Моделирование и разделение данных

В документации Microsoft приведен подробный пример моделирования и разбиения Azure Cosmos DB: https://learn.microsoft.com/ru-ru/azure/cosmos-db/nosql/model-partition-example.

Упражнение 4.3. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-4---managing-nosql-data-using-azure-cosmos-db.

Упражнение 4.4. Шпаргалки по запросам Cosmos DB

Шпаргалки по запросам Cosmos DB доступны для скачивания по адресу https://learn.microsoft.com/ru-ru/azure/cosmos-db/nosql/query-cheat-sheet.

Упражнение 4.5. Gremlin API

Для получения дополнительной информации о графовом API Gremlin прочитайте книгу по адресу https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html.

Упражнение 4.6. Дополнительные материалы по NoSQL и EF Core

В этой главе речь шла об Azure Cosmos DB. Если хотите узнать больше о NoSQL, таких как MongoDB, и их интеграции с EF Core, обратите внимание на следующие ссылки.

- *Использование NoSQL в качестве инфраструктуры хранения данных:* https://learn.microsoft.com/ru-ru/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/nosql-database-persistence-infrastructure.
- *Провайдеры БД документов для EF Core:* https://github.com/BlueshiftSoftware/EntityFrameworkCore.

Резюме

Из этой главы вы узнали:

- как в Azure Cosmos DB хранить данные с гибкой структурой;
- как создавать ресурсы Cosmos DB в эмуляторе и в облаке Azure;
- как управлять данными с помощью Cosmos SQL API;
- как писать серверный JavaScript-код в Cosmos DB.

В следующей главе вы узнаете, как с помощью типа `Task` повысить производительность своих приложений!

5 Многозадачность и конкурентность

В этой главе речь пойдет о том, как выполнять несколько задач одновременно, повышая производительность, масштабируемость и удобство работы с приложениями.

Процессы, потоки и задачи

Каждое консольное приложение — это *процесс*, который получает выделенные ресурсы, такие как память и потоки.

Поток выполняет инструкции кода последовательно. По умолчанию процесс имеет один поток, но, если задач несколько, это может стать узким местом. Потоки также хранят данные о текущем пользователе и настройках языка.

В Windows и других современных ОС используется *вытесняющая многозадачность* (псевдопараллельное выполнение задач). Это значит, что процессор поочередно выделяет *временной интервал* разным потокам.

1. Поток выполняется в течение своего временного интервала.
2. Когда время истекает, контекст потока сохраняется.
3. Загружается контекст другого потока, и он получает управление.

Переключение потоков затрачивает ресурсы, поэтому слишком много потоков могут замедлить работу вместо ускорения.

Если у вас немного сложных задач и вы хотите управлять ими вручную, можно создавать и управлять отдельными экземплярами потоков (`Thread`). Если же есть много небольших фоновых задач и основной поток, удобнее использовать класс `ThreadPool`. В этом случае задачи добавляются в очередь как экземпляры делегатов, а система автоматически распределяет их по потокам.

В этой главе мы будем управлять потоками через `Task`, поскольку этот механизм работает на более высоком уровне абстракции и упрощает многозадачность.

Когда несколько потоков пытаются одновременно получить доступ к общим ресурсам (переменным, файлам, БД), возникает *конкуренция*, которая может

приводить к задержкам и ошибкам. С этим процессом мы разберемся далее в главе.

Часто увеличение количества потоков не ускоряет решение задачи, а, наоборот, замедляет ее из-за борьбы за ресурсы (рис. 5.1).

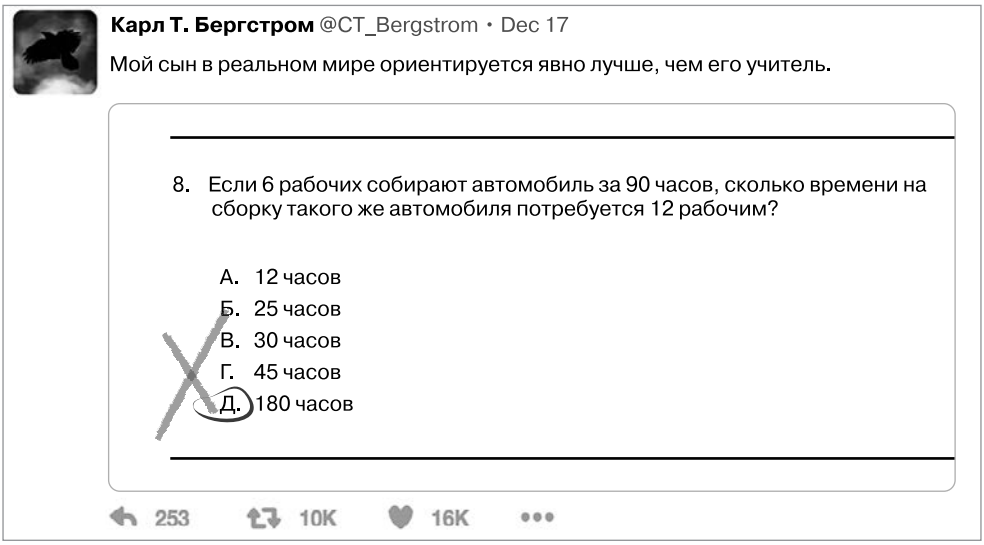


Рис. 5.1. Твит о задачах в реальном мире

Дельный совет: увеличение количества потоков не гарантирует рост производительности! Сначала протестируйте производительность кода в однопоточном режиме. Затем сравните ее с версией, где используются несколько потоков. Тестируйте код в среде, приближенной к рабочей, чтобы получить реальные результаты.

В следующем разделе разберем принципы асинхронного выполнения задач, позволяющие эффективнее управлять многозадачностью.

Асинхронное выполнение задач

Чтобы понять, как выполнять несколько задач *одновременно*, создадим консольное приложение, которое последовательно запустит три метода.

Каждый метод будет имитировать выполнение работы с разной продолжительностью:

- первый — 3 секунды;
- второй — 2 секунды;
- третий — 1 секунду.

Для этого воспользуемся классом `Thread`, чтобы приостановить выполнение потока на нужное время.

Синхронное выполнение нескольких действий

Сначала запустим методы *синхронно*, выполняя их один за другим.

1. В редакторе кода создайте консольное приложение со следующими настройками:
   - выберите шаблон проекта Console App (Консольное приложение);
   - в поле Solution name (Имя решения) укажите значение `Chapter05`;
   - в поле Project name (Имя проекта) укажите значение `WorkingWithTasks`;
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
   - снимите флажок Enable native AOT publishing (Включить публикацию Native AOT).
2. В файле проекта `WorkingWithTasks` статически импортируйте класс `System.Console` на глобальном уровне и добавьте строгую обработку предупреждений.
3. В том же проекте создайте файл класса `Program.Helpers.cs`.
4. В файле `Program.Helpers.cs` удалите весь код, а затем определите разделяемый класс `Program` с методами для вывода заголовков разделов и заголовков задач, а также для отображения информации о текущем потоке (с разными цветами, чтобы отличать их в консоли):

```
partial class Program
{
  private static void SectionTitle(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkYellow;
    WriteLine($"*** {title}");
    ForegroundColor = previousColor;
  }

  private static void TaskTitle(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.Green;
    WriteLine($"{title}");
    ForegroundColor = previousColor;
  }

  private static void OutputThreadInfo()
  {
    Thread t = Thread.CurrentThread;
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkCyan;
```

```
    WriteLine(
      "Thread Id: {0}, Priority: {1}, Background: {2}, Name: {3}",
      t.ManagedThreadId, t.Priority, t.IsBackground, t.Name ?? "null");

    ForegroundColor = previousColor;
  }
}
```

5. В проекте `WorkingWithTasks` создайте файл класса `Program.Methods.cs`.
6. В файле `Program.Methods.cs` удалите весь код и добавьте три метода, имитирующие выполнение работы:

```
partial class Program
{
  private static void MethodA()
  {
    TaskTitle("Starting Method A...");
    OutputThreadInfo();
    Thread.Sleep(3000); // Имитация трех секунд работы
    TaskTitle("Finished Method A.");
  }

  private static void MethodB()
  {
    TaskTitle("Starting Method B...");
    OutputThreadInfo();
    Thread.Sleep(2000); // Имитация двух секунд работы
    TaskTitle("Finished Method B.");
  }

  private static void MethodC()
  {
    TaskTitle("Starting Method C...");
    OutputThreadInfo();
    Thread.Sleep(1000); // Имитация одной секунды работы
    TaskTitle("Finished Method C.");
  }
}
```

7. Удалите содержимое файла `Program.cs`, а затем добавьте в него код:
   - для вывода информации о текущем потоке;
   - определения и запуска секундомера;
   - последовательного вызова трех методов;
   - вывода времени выполнения:

```
using System.Diagnostics; // Необходимо для работы секундомера

OutputThreadInfo();
Stopwatch timer = Stopwatch.StartNew();
```

```
SectionTitle("Running methods synchronously on one thread.");
MethodA();
MethodB();
MethodC();

WriteLine($"{timer.ElapsedMilliseconds:#,##0}ms elapsed.");
```

8. Запустите программу и дождитесь окончания выполнения всех трех методов, а затем просмотрите результат. Поскольку задачи выполняются последовательно, общее время работы составит чуть больше 6 секунд:

```
Thread Id: 1, Priority: Normal, Background: False, Name: null
*** Running methods synchronously on one thread.
Starting Method A...
Thread Id: 1, Priority: Normal, Background: False, Name: null
Finished Method A.
Starting Method B...
Thread Id: 1, Priority: Normal, Background: False, Name: null
Finished Method B.
Starting Method C...
Thread Id: 1, Priority: Normal, Background: False, Name: null
Finished Method C.
6,028ms elapsed.
```

Асинхронное выполнение нескольких действий с помощью задач

Класс `Thread` был добавлен еще в первую версию .NET (2002 год) и используется для создания потоков и управления ими, но работать с ним напрямую сложно и неудобно.

В 2010 году в .NET Framework 4.0 появился класс `Task`, который представляет собой асинхронную операцию. Задача — это высокоуровневая абстракция над потоком операционной системы, выполняющим операцию. Класс `Task` значительно упрощает создание и управление потоками, обернутыми в задачи.

Используя задачи, можно эффективно управлять несколькими потоками, и, таким образом, наш код будет выполняться *асинхронно*.

У каждого экземпляра `Task` есть два основных свойства:

- `Status` — показывает текущее состояние задачи;
- `CreationOptions` — определяет параметры, с которыми задача была создана.

Кроме того, у экземпляра `Task` есть метод `ContinueWith`, который позволяет настроить выполнение дополнительной операции после завершения задачи, а также можно использовать перечисление `TaskContinuationOptions` для более детальной настройки. Управление задачами может осуществляться через класс `TaskFactory`.

Начальные задания

Мы рассмотрим три способа выполнения методов с помощью экземпляров `Task`. В репозитории GitHub есть ссылки на статьи, в которых обсуждаются их преимущества и недостатки. Синтаксис каждого способа немного отличается, но все они создают экземпляр `Task` и запускают его.

1. В файле `Program.cs` закомментируйте вызовы методов A — C, а затем добавьте операторы для создания и запуска трех задач, по одному на метод:

```
Stopwatch timer = Stopwatch.StartNew();

/*
SectionTitle("Running methods synchronously on one thread.");
MethodA();
MethodB();
MethodC();
*/

SectionTitle("Running methods asynchronously on multiple threads.");

Task taskA = new(MethodA);
taskA.Start();
Task taskB = Task.Factory.StartNew(MethodB);
Task taskC = Task.Run(MethodC);

WriteLine($"{timer.ElapsedMilliseconds:#,##0}ms elapsed.");
```

Если вы не хотите отключать вызовы методов, то можете допустить их выполнение, но обязательно вызовите метод timer.Restart() после вывода заголовка нового раздела. Так вы сбросите тайминги для каждого раздела.

2. Выполните код и проанализируйте результат. Обратите внимание, что время в миллисекундах появляется почти сразу. Это происходит потому, что каждый из методов теперь выполняется в своем фоновом потоке, который выделяется из *пула потоков* (thread pool, TP).

```
*** Running methods asynchronously on multiple threads.
Starting Method A...
Thread Id: 4, Priority: Normal, Background: True, Name: .NET TP Worker
Starting Method C...
Thread Id: 7, Priority: Normal, Background: True, Name: .NET TP Worker
Starting Method B...
Thread Id: 6, Priority: Normal, Background: True, Name: .NET TP Worker
6ms elapsed.
```

Иногда консольное приложение может завершиться до того, как все задачи успеют запуститься и вывести данные!

Ожидание завершения задач

Иногда нужно дождаться завершения задачи, прежде чем продолжить выполнение. Для этого можно использовать метод `Wait` для экземпляра класса `Task` или статические методы `WaitAll` или `WaitAny` для нескольких задач (табл. 5.1).

Таблица 5.1. Методы Wait класса Task

Метод	Описание
`t.Wait()`	Ожидает завершения конкретной задачи `t`
`Task.WaitAny(Task[])`	Ожидает завершения любой задачи из переданного массива
`Task.WaitAll(Task[])`	Ожидает завершения всех задач в массиве

Использование методов wait в задачах

Посмотрим, как можно использовать методы ожидания, чтобы решить эту проблему в консольном приложении.

1. В файле `Program.cs` после создания трех задач и перед выводом прошедшего времени добавьте код, который собирает ссылки на эти задачи в массив и передает их методу `WaitAll`:

```
Task[] tasks = { taskA, taskB, taskC };
Task.WaitAll(tasks);
```

2. Выполните код и посмотрите на результат. Обратите внимание, что главный поток приостанавливается на вызове `WaitAll`, ожидая завершения всех трех задач, а затем выводит прошедшее время, составляющее чуть больше 3 секунд:

```
Starting Method A...
Starting Method B...
Thread Id: 4, Priority: Normal, Background: True, Name: .NET TP Worker
Thread Id: 6, Priority: Normal, Background: True, Name: .NET TP Worker
Starting Method C...
Thread Id: 7, Priority: Normal, Background: True, Name: .NET TP Worker
Finished Method C.
Finished Method B.
Finished Method A.
3,013ms elapsed.
```

Все три метода работают параллельно, но порядок их выполнения может различаться. Метод `MethodC`, который занимает всего 1 секунду, скорее всего, завершится первым, за ним — `MethodB` (2 секунды), а последним — `MethodA` (3 секунды).

Однако итоговое поведение программы зависит от характеристик центрального процессора. Он распределяет ресурсы и назначает временные интервалы для выполнения потоков, так что время завершения методов может варьироваться. Вы не можете управлять временем выполнения методов.

Выполнение задач продолжения

Если все три задачи могут выполняться параллельно, достаточно дождаться их завершения. Однако часто одна задача зависит от результата другой. В таких случаях используются *задачи продолжения*.

Мы создадим два метода, имитирующих вызов веб-сервиса, который возвращает денежную сумму, и выполнение запроса к БД, который определяет, сколько товаров стоит больше этой суммы. Результат первого метода будет передаваться второму. Чтобы реалистично моделировать задержку, вместо фиксированного времени ожидания воспользуемся классом `Random`, задавая случайный интервал от 2 до 4 секунд.

1. В файле `Program.Methods.cs` создайте два метода: один для имитации запроса к веб-сервису, второй — для работы с БД:

```
private static decimal CallWebService()
{
  TaskTitle("Starting call to web service...");
  OutputThreadInfo();
  Thread.Sleep(Random.Shared.Next(2000, 4000));
  TaskTitle("Finished call to web service.");
  return 89.99M;
}

private static string CallStoredProcedure(decimal amount)
{
  TaskTitle("Starting call to stored procedure...");
  OutputThreadInfo();
  Thread.Sleep((Random.Shared.Next(2000, 4000));
  TaskTitle("Finished call to stored procedure.");
  return $"12 products cost more than {amount:C}.";
}
```

2. В файле `Program.cs` закомментируйте код, создающий три предыдущие задачи, и добавьте новую последовательность: сначала вызов задачи веб-сервиса, затем передачу ее результата в задачу, выполняющую запрос к БД:

```
SectionTitle("Passing the result of one task as an input into another.");

Task<string> taskServiceThenSProc = Task.Factory
  .StartNew(CallWebService) // возвращает результат Task<decimal>
```

```
    .ContinueWith(previousTask => // возвращает результат Task<string>
      CallStoredProcedure(previousTask.Result));

WriteLine($"Result: {taskServiceThenSProc.Result}");

WriteLine($"{timer.ElapsedMilliseconds:#,##0}ms elapsed.");
```

3. Выполните код и проанализируйте результат:

```
Starting call to web service...
Thread Id: 4, Priority: Normal, Background: True, Name: .NET TP Worker
Finished call to web service.
Starting call to stored procedure...
Thread Id: 6, Priority: Normal, Background: True, Name: .NET TP Worker
Finished call to stored procedure.
Result: 12 products cost more than £89.99.
5,463ms elapsed.
```

Символ денежной единицы зависит от настроек культуры операционной системы. Например, у меня отображается £, а у вас может быть другой символ. Как управлять культурами, мы рассмотрим в главе 7.

При запуске кода можно заметить, что вызовы веб-сервиса и БД выполняются либо в разных потоках (например, 4 и 6), либо в одном, если предыдущий поток освободился и может быть использован повторно.

Вложенные и дочерние задачи

Помимо зависимых задач, можно определять вложенные и дочерние задачи. *Вложенная* — это задача, создаваемая внутри другой задачи. *Дочерняя* — вложенная задача, которая должна завершиться до окончания родительской.

Разберемся, как это работает.

1. В файле `Program.Methods.cs` добавьте два метода, один из которых создает и запускает задачу для выполнения другого метода:

```
private static void OuterMethod()
{
  TaskTitle("Outer method starting...");
  Task innerTask = Task.Factory.StartNew(InnerMethod);
  TaskTitle("Outer method finished.");
}
```

```
private static void InnerMethod()
{
  TaskTitle("Inner method starting...");
  Thread.Sleep(2000);
  TaskTitle("Inner method finished.");
}
```

2. В файле `Program.cs` добавьте код для запуска задачи с внешним методом и дождитесь ее завершения перед остановкой программы:

```
SectionTitle("Nested and child tasks");

Task outerTask = Task.Factory.StartNew(OuterMethod);
outerTask.Wait();
WriteLine("Console app is stopping.");
```

3. Выполните код и проанализируйте результат:

```
Outer method starting...
Inner method starting...
Outer method finished.
Console app is stopping.
```

Хотя мы и ожидаем завершения внешней задачи, ее внутренняя задача может остаться невыполненной. В результате консольное приложение может завершить работу раньше, чем начнется внутренняя задача:

```
Outer method starting...
Outer method finished.
Console app is stopping.
```

4. Чтобы связать вложенные задачи в цепочку «родительская — дочерняя», нужно указать специальный параметр. В файле `Program.Methods.cs` измените код, добавив в `TaskCreationOptions` значение `AttachedToParent`:

```
private static void OuterMethod()
{
  TaskTitle("Outer method starting...");

  Task innerTask = Task.Factory.StartNew(InnerMethod,
    TaskCreationOptions.AttachedToParent);

  TaskTitle("Outer method finished.");
}
```

5. Выполните код и обратите внимание, что теперь внутренняя задача завершается раньше внешней:

```
Outer method starting...
Inner method starting...
Outer method finished.
Inner method finished.
Console app is stopping.
```

6. Как вариант, внешний метод может завершиться раньше, чем запустится внутренний:

```
Outer method starting...
Outer method finished.
Inner method starting...
Inner method finished.
Console app is stopping.
```

Возможна и другая ситуация: внешний метод может завершиться раньше, чем начнется внутренняя задача. Но благодаря значению AttachedToParent консоль не закроется, пока не завершатся обе задачи.

Обертывание объектов в задачи

Иногда нужно сделать метод асинхронным, но его возвращаемый результат сам по себе не является задачей. В таких случаях можно обернуть результат в задачу, вернуть исключение или указать, что задача была отменена. Для этого используются статические методы класса `Task`, представленные в табл. 5.2.

Таблица 5.2. Методы создания задачи для различных сценариев

Метод	Описание
`FromResult<TResult>(TResult)`	Создает объект `Task<TResult>`, где `Result` — заданное значение, а `Status` — `RanToCompletion`
`FromException<TResult>(Exception)`	Создает задачу `Task<TResult>`, завершающуюся с указанным исключением
`FromCanceled<TResult>(CancellationToken)`	Создает задачу `ask<TResult>`, завершенную из-за отмены с заданным маркером

Эти методы пригодятся в следующих случаях:

- реализации асинхронного интерфейса, если сам метод синхронный (актуально для сайтов и сервисов);
- имитации асинхронных вызовов во время модульного тестирования.

Допустим, требуется создать метод для проверки XML-ввода, который должен соответствовать интерфейсу с `Task<T>` в качестве возвращаемого типа:

```
public interface IValidation
{
  Task<bool> IsValidXmlTagAsync(this string input);
}
```

Код в этом разделе приведен исключительно для примера. Вам не нужно добавлять его в свой проект.

В этом случае можно воспользоваться методами `FromX`, чтобы обернуть результат в задачу:

```
using System.Text.RegularExpressions;

namespace Packt.Shared;

public static class StringExtensions : IValidation
{
  public static Task<bool> IsValidXmlTagAsync(this string input)
  {
    if (input == null)
    {
      return Task.FromException<bool>(
      new ArgumentNullException($"Missing {nameof(input)} parameter"));
    }

    if (input.Length == 0)
    {
      return Task.FromException<bool>(
      new ArgumentException($"{nameof(input)} parameter is empty."));
    }

    return Task.FromResult(Regex.IsMatch(input,
      @"^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$"));
  }
}
```

Если метод возвращает значение типа `Task` (эквивалент `void` в синхронном коде), можно просто вернуть объект `Task.CompletedTask`:

```
public Task DeleteCustomerAsync()
{
  // ...
  return Task.CompletedTask;
}
```

При параллельном выполнении задач разным потокам может потребоваться доступ к одним и тем же ресурсам. Поэтому важно понимать, как безопасно работать с такими ресурсами, чтобы избежать конфликтов и ошибок.

Синхронизация доступа к общим ресурсам

При одновременном выполнении нескольких потоков существует риск, что они попытаются обратиться к одной и той же переменной или к другому ресурсу одновременно. Это может привести к выдаче некорректных данных, неожиданным ошибкам или даже к повреждению данных. Поэтому важно обеспечить *потокобезопасность* кода.

Один из простых способов — использовать объектную переменную в роли блокировки. Своего рода это «правило раковины» из книги Уильяма Голдинга «Повелитель мух».

В романе Ральф и Хрюша решили, что говорить может только тот, кто держит в руках раковину. По аналогии, если поток владеет «раковиной» (переменная `Conch` в моих проектах), другие потоки не должны изменять общий ресурс. Однако «раковина» — это не принудительная блокировка, а лишь способ организовать безопасный доступ к ресурсам.

В .NET есть несколько инструментов для синхронизации доступа к общим ресурсам:

- `Monitor` — позволяет потокам координировать доступ к общим ресурсам в пределах одного процесса;
- `Interlocked` — предоставляет атомарные операции над числовыми типами на уровне процессора, таким образом, это наиболее быстрый способ защиты простых переменных.

В зависимости от ситуации можно выбрать наиболее подходящий механизм, чтобы избежать ошибок при многопоточной работе.

Доступ к ресурсу из нескольких потоков

Давайте разберемся, как несколько потоков могут совместно использовать один и тот же ресурс, создав для этого консольное приложение.

1. В редакторе кода создайте консольное приложение `SynchronizingResourceAccess` и добавьте его в решение `Chapter05`.
2. Статически импортируйте класс `System.Console` на глобальном уровне и настройте строгую обработку предупреждений.
3. Создайте файл класса `SharedObjects.cs`.
4. Удалите из него весь код и определите статический класс с полем для хранения общего сообщения. Оно будет играть роль общего ресурса:

```
public static class SharedObjects
{
  public static string? Message; // общий ресурс
}
```

5. Создайте файл класса `Program.Methods.cs`.
6. Удалите все содержимое из этого файла. Затем определите два метода, каждый из которых выполняет цикл из пяти итераций, ожидает случайный интервал до 2 секунд и добавляет либо A, либо B в общее сообщение:

```
partial class Program
{
  private static void MethodA()
  {
    for (int i = 0; i < 5; i++)
    {
      // Имитация 2 секунд работы текущего потока
      Thread.Sleep(Random.Shared.Next(2000));

      // Добавление буквы A в общее сообщение
      SharedObjects.Message += "A";

      // Отображение активности в консольном выводе
      Write(".");
    }
  }

  private static void MethodB()
  {
    for (int i = 0; i < 5; i++)
    {
      Thread.Sleep(Random.Shared.Next(2000));
      SharedObjects.Message += "B";
      Write(".");
    }
  }
}
```

7. Удалите содержимое файла `Program.cs`. Добавьте оператор импорта пространства имен диагностических типов, таких как `Stopwatch`, и напишите код, который запускает оба метода в отдельных задачах, ожидает их завершения и выводит прошедшее время:

```
using System.Diagnostics; // Для типа Stopwatch

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($"Results: {SharedObjects.Message}.");
WriteLine($"{watch.ElapsedMilliseconds:N0} elapsed milliseconds.");
```

8. Выполните код и проанализируйте результат:

```
Please wait for the tasks to complete.
..........
Results: BABABAABBA.
5,753 elapsed milliseconds.
```

Вы увидите, что оба потока изменяют сообщение одновременно. В реальном приложении такая ситуация может привести к некорректной работе программы, например к потере данных или неожиданным ошибкам. Чтобы избежать этого, необходимо контролировать доступ к ресурсу. Мы можем наложить взаимоисключающую блокировку на объект `Conch` и добавить в методы проверку «раковины» перед изменением данных. В следующем разделе мы разберем, как это реализовать.

Применение взаимно исключающей блокировки к «раковине»

Теперь воспользуемся объектом `Conch` для ограничения доступа к общему ресурсу. Одновременно к нему должен обращаться только один поток.

1. В файле `SharedObjects.cs` объявите и создайте объектную переменную, которая будет выполнять роль «раковины»:

```
public static object Conch = new(); // Общий объект для блокировки
```

2. В файле `Program.Methods.cs` добавьте конструкцию `for` с кодом блокировки объекта `Conch` в обоих методах — `MethodA` и `MethodB`. Код для метода `MethodB` приведен ниже:

```
lock (SharedObjects.Conch)
{
  for (int i = 0; i < 5; i++)
  {
    Thread.Sleep(Random.Shared.Next(2000));
    SharedObjects.Message += "B";
    Write(".");
  }
}
```

Дельный совет: поскольку проверка объекта Conch выполняется добровольно, блокировка только одного метода не решит проблему. Доступ останется неконтролируемым, если другие методы не будут следовать этому же правилу. Поэтому убедитесь, что все методы, которые обращаются к этому ресурсу, используют блокировку, прежде чем совместно задействовать ресурсы.

3. Выполните код и проанализируйте результат:

```
Please wait for the tasks to complete.
..........
Results: BBBBBAAAAA.
10,345 elapsed milliseconds.
```

Хотя выполнение программы теперь занимает больше времени, мы обеспечили безопасный поочередный доступ к данным. Потоки `MethodA` и `MethodB` могут запускаться в любом порядке, но только один из них будет работать с ресурсом одновременно. Как только один метод завершит выполнение, блокировка снимется и другой поток сможет приступить к работе.

Оператор блокировки

Думаю, вам интересно, как именно оператор `lock` «блокирует» объектную переменную (подсказка: она не блокирует сам объект!):

```
lock (SharedObjects.Conch)
{
  // Работа с общим ресурсом
}
```

Компилятор C# преобразует оператор блокировки в конструкцию `try-finally`, которая с помощью класса `Monitor` получает и освобождает объект `Conch`.

```
try
{
  Monitor.Enter(SharedObjects.Conch);
  // Работа с общим ресурсом
}
finally
{
  Monitor.Exit(SharedObjects.Conch);
}
```

Когда поток вызывает метод `Monitor.Enter` для ссылочного типа, он проверяет, не блокирован ли объект `Conch` другим потоком. Если объект занят, поток ожидает. Если объект свободен, поток получает его и продолжает работать с общим ресурсом. Как только работа с ресурсом завершена, поток вызывает метод `Monitor.Exit`, освобождая объект `Conch`.

Теперь ожидающий поток получает объект `Conch` и выполняет свои задачи. Важно, чтобы все потоки следовали правилам работы с объектом `Conch`, вызывая методы `Monitor.Enter` и `Monitor.Exit` соответствующим образом.

Дельный совет: в качестве объекта Conch нельзя использовать структуры (типы struct). Методу Monitor.Enter нужен ссылочный тип, поскольку блокирует адрес памяти. Это важно, так как внутренние структуры данных объекта Conch не блокируются.

Предотвращение взаимных блокировок

Крайне важно понимать, как оператор блокировки транслируется компилятором в вызовы методов класса `Monitor`, так как неправильные действия могут привести к взаимным блокировкам.

Взаимные блокировки возникают, когда несколько потоков одновременно пытаются получить доступ к двум или более общим ресурсам, каждый из которых контролируется объектом `Conch`. Например:

- поток X «блокирует» объект `Conch` A и начинает работать с ресурсом A;
- поток Y «блокирует» объект `Conch` B и работает с ресурсом B;
- поток X продолжает работать с ресурсом A, но для работы с ресурсом B пытается заблокировать объект `Conch` B, но блокируется, потому что он уже заблокирован потоком Y;
- поток Y продолжает работу с ресурсом B и пытается заблокировать объект `Conch` A, но блокируется, так как он уже заблокирован потоком X.

Чтобы предотвратить такие ситуации, можно использовать тайм-аут при попытке захвата блокировки. Для этого нужно вручную использовать класс `Monitor` вместо оператора `lock`. Рассмотрим, как это сделать.

1. В файле `Program.Methods.cs` замените оператор блокировки на код, который пытается захватить объект `Conch` с тайм-аутом. В случае неудачи генерируется ошибка, а затем поток выходит из блока `Monitor`, позволяя другим потокам продолжить работу. Пример для метода `MethodB` показан ниже:

```
try
{
  if (Monitor.TryEnter(SharedObjects.Conch, TimeSpan.FromSeconds(15)))
  {
    for (int i = 0; i < 5; i++)
    {
      Thread.Sleep(Random.Shared.Next(2000));
      SharedObjects.Message += "B";
      Write(".");
    }
  }
  else
  {
    WriteLine("Method B timed out when entering a monitor on conch.");
  }
}
finally
```

```
  {
    Monitor.Exit(SharedObjects.Conch);
  }
```

2. Выполните код и проверьте результат. Он будет аналогичен предыдущему (хотя первым объект Conch может получить либо ресурс A, либо ресурс B), но возможные взаимные блокировки мы предотвратили.

Дельный совет: используйте оператор lock только во избежание потенциальных взаимных блокировок. В противном случае применяйте вместо него метод Monitor.TryEnter с конструкцией try-finally, чтобы задать тайм-аут и выйти из блокировки, если она возникнет. Подробнее о передовых методах работы с потоками читайте по адресу https://learn.microsoft.com/ru-ru/dotnet/standard/threading/managed-threading-best-practices.

Синхронизация событий

События .NET не безопасны для потоков, поэтому в многопоточных сценариях их следует избегать.

Некоторые разработчики, узнав о том, что события не потокобезопасны, пытаются применить эксклюзивные блокировки при добавлении и удалении обработчиков событий или при их вызове:

```
// поле делегата события
public event EventHandler? Shout;

// объект conch
private object eventConch = new();

// метод
public void Poke()
{
  lock (eventConch) // плохая идея
  {
    // Если какой-то элемент прослушивает...
    if (Shout != null)
    {
      // ...вызовите делегат, чтобы поднять событие
      Shout(this, EventArgs.Empty);
    }
  }
}
```

Дельный совет: так нужно или нет использовать блокировки при обработке событий? Ответ неоднозначный и зависит от множества факторов, поэтому оценить это можно только в контексте конкретного случая. Более подробную информацию о событиях и безопасности потоков можно найти по адресу https://learn.microsoft.com/ru-ru/archive/blogs/cburrows/field-like-events-considered-harmful. Как поясняет Стивен Клири в своей статье по адресу https://blog.stephencleary.com/2009/06/threadsafe-events.html, ситуация действительно сложная.

Атомарность операций ЦП

Слово «атомарный» происходит от греческого *atomos*, что означает «неразделимый». Важно понимать, какие операции атомарны в многозадачных режимах, поскольку, если операция неатомарна, ее выполнение может быть прервано другим потоком, что приведет к ошибке.

Например, атомарен ли оператор инкремента в C#, как показано ниже?

```
int x = 3;
x++; // Это атомарная операция ЦП?
```

Нет! Увеличение целого числа осуществляется посредством следующих трех операций процессора.

1. Загрузка значения из переменной экземпляра в регистр.
2. Увеличение значения.
3. Сохранение значения в переменной экземпляра.

Поток может быть прерван после первых двух операций, а второй поток может выполнить все три операции. Когда первый поток продолжит выполнение, он перезапишет значение переменной и результат инкремента/декремента, выполненного вторым потоком, будет утерян!

Существует тип `Interlocked`, позволяющий выполнять с целочисленными типами атомарные операции, такие как `Add`, `Increment`, `Decrement`, `Exchange`, `CompareExchange`, `And`, `Or` и `Read`:

- `System.Int32 (int)`, `System.UInt32 (uint)`
- `System.Int64 (long)`, `System.UInt64 (ulong)`

Тип `Interlocked` не поддерживает такие типы, как `byte`, `sbyte`, `short`, `ushort` и `decimal`.

Тип `Interlocked` также поддерживает атомарные операции обмена значениями в памяти, такие как `Exchange` и `CompareExchange`, для следующих типов:

- `System.Single (float)`, `System.Double (double)`
- `nint`, `nuint`
- `System.Object(object)`

Посмотрим, как это работает.

1. В классе `SharedObjects` объявите поле для подсчета выполненных операций:

   ```
   public static int Counter; // Другой общий ресур
   ```

2. В файле `Program.Methods.cs`, в методах A и B, внутри цикла `for` и после кода изменения строкового значения добавьте оператор для безопасного увеличения счетчика:

   ```
   Interlocked.Increment(ref SharedObjects.Counter);
   ```

3. В файле `Program.cs` перед выводом прошедшего времени выведите в консоль текущее значение счетчика:

```
WriteLine($"{SharedObjects.Counter} string modifications.");
```

4. Выполните код и проанализируйте результат:

```
Please wait for the tasks to complete.
..........
Results: BBBBBAAAAA.
10 string modifications.
13,531 elapsed milliseconds.
```

Внимательные читатели заметят, что объект `Conch` защищает все общие ресурсы, доступ к которым осуществляется в блоке кода, заблокированном этим объектом. Поэтому в данном примере класс `Interlocked` не требуется. Однако если бы другой общий ресурс, например `Message`, не был защищен, то класс `Interlocked` стал бы необходим.

Другие типы синхронизации

`Monitor` и `Interlocked` — простые и эффективные механизмы взаимоисключающих блокировок, но иногда для синхронизации доступа к общим ресурсам требуются более сложные инструменты. Ниже приведена табл. 5.3 с дополнительными типами синхронизации.

Таблица 5.3. Типы синхронизации

Тип	Описание
`ReaderWriterLock`, `ReaderWriterLockSlim`	Эти типы позволяют нескольким потокам работать в *режиме чтения*, одному потоку — в *режиме записи* с эксклюзивным правом на блокировку записи, а также одному потоку, имеющему доступ на чтение, — находиться в *режиме обновляемого чтения*, из которого поток может перейти в режим записи без потери доступа на чтение к ресурсу
`Mutex`	Подобно *`Monitor`*, предоставляет эксклюзивный доступ к ресурсу, но используется для межпроцессной синхронизации
`Semaphore`, `SemaphoreSlim`	Ограничивают количество потоков, которые могут одновременно работать с ресурсом или пулом ресурсов, задавая количество доступных слотов. Способ известен как *ограничение ресурсов* (throttling), а не их *блокировка*
`AutoResetEvent`, `ManualResetEvent`	Обработчики событий, которые позволяют потокам синхронизировать свои действия, подавая сигналы и ожидая их

Теперь, когда мы рассмотрели основные механизмы синхронизации доступа к общим ресурсам в многопоточном программировании, стоит обратить внимание на новые ключевые слова, появившиеся в C# 5, которые упрощают написание асинхронного кода.

Конструкции async и await

Конструкции `async` и `await` были введены в версии C# 5 для упрощения асинхронного программирования и особенно полезны в следующих ситуациях.

- Реализация многозадачности для *графического интерфейса пользователя* (Graphical User Interface, *GUI*).
- Улучшение масштабируемости веб-приложений и веб-сервисов.

В главе 16 мы подробно рассмотрим, как ключевые слова `async` и `await` применяются для реализации многозадачности в GUI.

Прежде чем перейти к практическим примерам, разберем теоретические причины, по которым эти ключевые слова были добавлены в C#.

Ускорение отклика консольных приложений

Одно из ограничений консольных приложений заключается в том, что ключевое слово `await` можно использовать только в методах, помеченных как `async`, а в версиях C# до 7 включительно метод `Main` нельзя выполнять асинхронно. Такая возможность появилась лишь с выходом C# 7.1. Вот как это можно реализовать.

1. В редакторе кода создайте консольное приложение `AsyncConsole` и добавьте его в решение `Chapter05`.
2. Удалите содержимое файла `Program.cs`, статически импортируйте класс `System.Console` на глобальном уровне, а затем добавьте код для создания экземпляра `HttpClient`, выполнения запроса к главной странице Apple и вывода ее размера в байтах:

```
using static System.Console;

HttpClient client = new();

HttpResponseMessage response =
  await client.GetAsync("http://www.apple.com/");

WriteLine("Apple's home page has {0:N0} bytes.",
  response.Content.Headers.ContentLength);
```

3. Выполните сборку проекта (завершится успешно). В версиях .NET 5 и ниже шаблон проекта создавал класс `Program` с неасинхронным методом `Main`, что приводило к ошибке:

```
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/apps-services-net7/
Chapter04/AsyncConsole/AsyncConsole.csproj]
```

4. В этих версиях необходимо было добавить ключевое слово `async` в метод `Main` и изменить его тип возвращаемого значения на `Task`. В .NET 6 и позднее используется функция программы верхнего уровня, которая автоматически создает класс `Program` с асинхронным методом `Main`.
5. Выполните код и проверьте результат. В вашем случае размер страницы будет отличаться, так как компания Apple часто обновляет свою главную страницу. Мой результат показан ниже:

```
Apple's home page has 40,252 bytes.
```

Управление асинхронными потоками

В версии .NET Core 3 компания Microsoft реализовала поддержку асинхронной обработки потоков.

Руководство по асинхронным потокам вы найдете по адресу https://learn.microsoft.com/ru-ru/dotnet/csharp/asynchronous-programming/generate-consume-asynchronous-stream.

До выхода C# 8 и .NET Core 3 ключевое слово `await` использовалось только для задач, возвращающих скалярные значения. Поддержка асинхронных потоков в .NET Standard 2.1 позволяет асинхронному методу возвращать последовательность значений одно за другим в асинхронном режиме.

Попробуем вернуть три случайных целых числа в виде асинхронного потока.

1. В редакторе кода создайте консольное приложение `AsyncEnumerable` в решении `Chapter05`.
2. Статически импортируйте класс `System.Console` на глобальном уровне и добавьте строгую обработку предупреждений.
3. Создайте файл `Program.Methods.cs`.

4. В файле `Program.Methods.cs` удалите весь код и определите метод, использующий ключевое слово `yield` для асинхронного возврата случайной последовательности из трех чисел:

```
partial class Program
{
  private static async IAsyncEnumerable<int> GetNumbersAsync()
  {
    Random r = Random.Shared;

    // Моделирование работы длительностью от 1,5 до 3 секунд
    await Task.Delay(r.Next(1500, 3000));

    // Возврат случайного числа от 1 до 1000
    yield return r.Next(1, 1001);

    await Task.Delay(r.Next(1500, 3000));
    yield return r.Next(1, 1001);

    await Task.Delay(r.Next(1500, 3000));
    yield return r.Next(1, 1001);
  }
}
```

5. Удалите содержимое файла `Program.cs` и добавьте код для перечисления последовательности чисел:

```
// Использование асинхронных потоков для асинхронного
// итерационного просмотра коллекции
await foreach (int number in GetNumbersAsync())
{
  WriteLine($"Number: {number}");
}
```

6. Выполните код и проанализируйте результат:

```
Number: 509
Number: 813
Number: 307
```

Ускорение отклика GUI-приложений

До сих пор в книге мы работали только с консольными приложениями. Разработка веб-приложений, веб-сервисов и приложений с графическим интерфейсом, таких как классические и мобильные приложения для Windows, значительно усложняет жизнь программиста.

Одной из причин является наличие специального потока для GUI-приложений — потока *пользовательского интерфейса (UI)*. Для работы с графическими интерфейсами следует учитывать два важных правила.

- Не выполняйте длительные задачи в потоке UI.
- Не обращайтесь к элементам UI из других потоков, кроме потока UI.

Чтобы соблюдать эти правила, разработчики должны были писать сложный код для выполнения длительных задач в отдельных потоках, не связанных с пользовательским интерфейсом. После завершения задачи результаты должны были безопасно передаваться в поток UI для отображения пользователю. Проекты быстро становились запутанными.

Для решения проблемы в C# 5 и более поздних версиях появились ключевые слова `async` и `await`, которые позволяют продолжать писать код, выглядящий как синхронный, при этом компилятор C# магически создает сложный конечный автомат, следя за потоками. Комбинация этих двух ключевых слов позволяет асинхронному методу работать в рабочем потоке, а по завершении возвращать результаты в поток UI.

Рассмотрим пример. Мы создадим классическое *WPF*-приложение *(Windows Presentation Foundation)* для Windows, которое будет извлекать данные о сотрудниках из базы данных Northwind в SQL Server, используя такие низкоуровневые типы, как `SqlConnection`, `SqlCommand` и `SqlDataReader`.

База данных Northwind имеет среднюю сложность и достаточное количество данных для выборки. Мы рассмотрели и настроили ее в главе 2.

Внимание! Это задание можно выполнить только в операционной системе Microsoft Windows с базой данных Northwind в Microsoft SQL Server. Это единственный не кросс-платформенный раздел в книге (технологии WPF уже 17 лет!). Для выполнения задания используйте программу Visual Studio 2022 или Visual Studio Code.

На этом этапе сосредоточимся на том, как сделать приложение с GUI более отзывчивым. О языке XAML и создании кросс-платформенных GUI-приложений вы узнаете в главе 16. Поскольку технология WPF в этой книге не рассматривается, вам предстоит прекрасная возможность попрактиковаться с WPF-приложением, хотя мы и не будем разбирать его подробно. Поехали!

1. В Visual Studio 2022 создайте проект `WPF Application` (Приложение ЦЗА) на языке C#, присвойте ему имя `WpfResponsive` и поместите в решение `Chapter05`.

В Visual Studio Code выполните команду `dotnet new wpf`. В JetBrains Rider выберите вариант Desktop Application (Классическое приложение), а затем присвойте параметру Type (Тип) значение WPF Application (Приложение WPF).

2. Добавьте в проект ссылку на пакет `Microsoft.Data.SqlClient`.
3. В файле проекта обратите внимание на тип вывода — Windows EXE, целевой фреймворк — .NET для Windows (он не будет работать на других платформах, таких как macOS и Linux), а в проекте используется WPF:

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

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net8.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Data.SqlClient" Version="5.2.0" />
  </ItemGroup>

</Project>
```

4. Выполните сборку проекта `WpfResponsive` для восстановления зависимостей.
5. В файле `MainWindow.xaml` добавьте в элемент `<Grid>` разметку для создания двух кнопок, текстового поля и поля со списком, расположенных вертикально на панели в виде стека:

```
<StackPanel>
  <Button Name="GetEmployeesSyncButton" Click="GetEmployeesSyncButton_Click">
    Get Employees Synchronously</Button>
  <Button Name="GetEmployeesAsyncButton" Click="GetEmployeesAsyncButton_Click">
    Get Employees Asynchronously</Button>
  <TextBox HorizontalAlignment="Stretch" Text="Type in here" />
  <ListBox Name="EmployeesListBox" Height="400" />
</StackPanel>
```

Visual Studio 2022 плотно поддерживает создание WPF-приложений и предоставляет технологию IntelliSense для редактирования кода и разметки XAML. В Visual Studio Code этой поддержки нет.

6. Добавьте в файл `MainWindow.xaml.cs` операторы импорта пространств имен для работы с ADO.NET и классом `Stopwatch`:

```
using Microsoft.Data.SqlClient; // Для класса SqlConnection и т. д.
using System.Diagnostics;       // Для класса Stopwatch
```

7. В конструкторе класса `MainWindow` определите два строковых поля для строки подключения к базе данных и SQL-запроса:

```
private string connectionString;

private string sql = "WAITFOR DELAY '00:00:05';" +
  "SELECT EmployeeId, FirstName, LastName FROM Employees";
```

SQL-код содержит два оператора. Первый из них имитирует длительное выполнение, задерживая работу на 5 секунд.

8. В конструкторе класса `MainWindow` после вызова метода `InitializeComponent` используйте класс `SqlConnectionStringBuilder` для формирования строки подключения к базе данных:

```
public MainWindow()
{
  InitializeComponent();

  // Измените при необходимости для работы с вашей БД Northwind
  SqlConnectionStringBuilder builder = new();

  builder.DataSource = ".";
  builder.InitialCatalog = "Northwind";
  builder.Encrypt = false;
  builder.MultipleActiveResultSets = true;
  builder.ConnectTimeout = 5;

  // Чтобы использовать интегрированную проверку подлинности Windows
  builder.IntegratedSecurity = true;

  // Чтобы использовать проверку подлинности SQL Server
  // builder.UserID = Environment.GetEnvironmentVariable("MY_SQL_USR");
  // builder.Password = Environment.GetEnvironmentVariable("MY_SQL_PWD");

  connectionString = builder.ConnectionString;
}
```

9. Создайте обработчики событий, инициируемых нажатием кнопок, с учетом строковых констант для подключения к базе данных Northwind и заполнения поля списка идентификаторами и именами сотрудников:

```
private void GetEmployeesSyncButton_Click(object sender, RoutedEventArgse)
{
  Stopwatch timer = Stopwatch.StartNew();
  using (SqlConnection connection = new(connectionString))
  {
```

```
      try
      {
        connection.Open();

        SqlCommand command = new(sql, connection);
        SqlDataReader reader = command.ExecuteReader();

        while (reader.Read())
        {
          string employee = string.Format("{0}: {1} {2}",
            reader.GetInt32(0), reader.GetString(1),
            reader.GetString(2));

          EmployeesListBox.Items.Add(employee);
        }

        reader.Close();
        connection.Close();
      }
      catch (Exception ex)
      {
        MessageBox.Show(ex.Message);
      }
    }
    EmployeesListBox.Items.Add(
      $"Sync: {timer.ElapsedMilliseconds:N0}ms");
  }

  private async void GetEmployeesAsyncButton_Click(
    object sender, RoutedEventArgs e)
  {
    Stopwatch timer = Stopwatch.StartNew();

    using (SqlConnection connection = new(connectionString))
    {
      try
      {
        await connection.OpenAsync();

        SqlCommand command = new(sql, connection);
        SqlDataReader reader = await command.ExecuteReaderAsync();

        while (await reader.ReadAsync())
        {
          string employee = string.Format("{0}: {1} {2}",
            await reader.GetFieldValueAsync<int>(0),
            await reader.GetFieldValueAsync<string>(1),
            await reader.GetFieldValueAsync<string>(2));
```

```
        EmployeesListBox.Items.Add(employee);
      }
      await reader.CloseAsync();
      await connection.CloseAsync();
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
  }
  EmployeesListBox.Items.Add(
    $"Async: {timer.ElapsedMilliseconds:N0}ms");
}
```

Учтите следующие моменты.

- Метод `async void` обычно не рекомендуется, потому что не сообщает о своем завершении и не позволяет отменить выполнение, так как не возвращает объект `Task` или `Task<T>` для управления процессом.
- В SQL-запросе используется команда `WAITFOR DELAY`, имитирующая длительную операцию, которая занимает 5 секунд, после чего выполняется выборка данных из таблицы `Employees`.
- Обработчик события `GetEmployeesSyncButton_Click` использует синхронные методы для подключения к базе данных и извлечения данных о сотрудниках.
- Обработчик события `GetEmployeesAsyncButton_Click` помечен как асинхронный и использует оператор `await` для асинхронного подключения и извлечения данных о сотрудниках.
- Оба обработчика событий используют секундомер для записи времени выполнения операций (в миллисекундах) и добавления его в поле списка.

10. Запустите проект WPF-приложения без отладки.
11. Щелкните кнопкой мыши в текстовом поле, введите текст и обратите внимание на отклик графического интерфейса.
12. Нажмите кнопку Get Employees Synchronously.
13. Щелкните кнопкой мыши в текстовом поле и обратите внимание, что графический интерфейс не реагирует.
14. Подождите не менее 5 секунд, пока в окне не появится список сотрудников.
15. Щелкните кнопкой мыши на текстовом поле, введите текст и обратите внимание, что графический интерфейс снова стал откликаться.
16. Нажмите кнопку Get Employees Asynchronously.

17. Щелкните кнопкой мыши на текстовом поле, введите текст и обратите внимание, что графический интерфейс откликается даже в процессе выполнения операции. Продолжайте вводить текст до тех пор, пока в окне не появится список сотрудников (рис. 5.2).

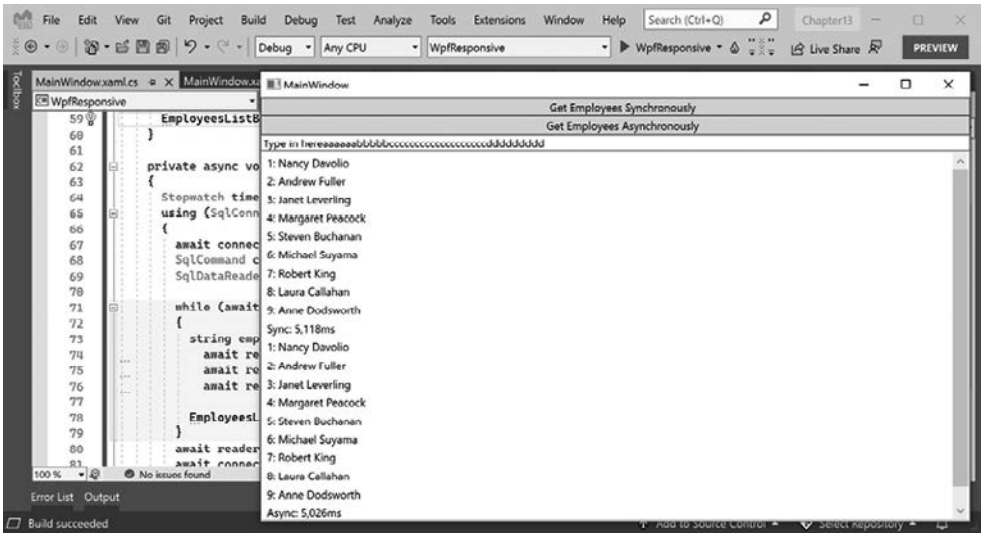


Рис. 5.2. Синхронная и асинхронная загрузка сотрудников в WPF-приложении

18. Обратите внимание на разницу в режиме выполнения этих двух операций. При синхронной выборке данных пользовательский интерфейс блокируется, а при асинхронной — остается отзывчивым.
19. Закройте WPF-приложение.

Оптимизация масштабируемости веб-приложений и веб-сервисов

Ключевые слова `async` и `await` можно применять на стороне сервера при создании сайтов, приложений и сервисов. Для клиентского приложения ничего не изменится (или даже возрастет, пусть и незначительно, время выполнения запроса). Таким образом, подобный подход не только не улучшает клиентский опыт, но даже может его ухудшить.

На сервере создаются дополнительные, менее ресурсоемкие рабочие потоки, ожидающие завершения длительных задач. Таким образом, затратные потоки ввода-вывода могут обрабатывать другие запросы, не блокируя выполнение. В результате увеличивается масштабируемость веб-приложения или сервиса в целом, позволяя обслуживать больше клиентов одномоментно.

Распространенные типы с поддержкой многозадачности

Существует множество распространенных типов с асинхронными методами, к которым применимо ключевое слово `await` (табл. 5.4).

Таблица 5.4. Типы с асинхронными методами

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`

Дельный совет: если имя метода оканчивается словом Async, проверьте, возвращает ли он объект Task или Task<T>. Если возвращает, его можно вызывать с await вместо синхронного метода. В этом случае сам метод нужно объявить с модификатором async.

Оператор await в блоках catch

Когда ключевые слова `async` и `await` были добавлены в версии C# 5, оператор `await` можно было использовать только в блоке `try`, но не в `catch`. В C# 6 и более поздних версиях это ограничение убрали, и теперь `await` поддерживается в обоих блоках.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 5.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. Какие сведения можно получить о процессе?
2. Насколько точен класс `Stopwatch`?
3. Какой суффикс должен иметь метод, возвращающий объект `Task` или `Task<T>`?
4. Какое ключевое слово должно быть указано в объявлении метода, если внутри него используется оператор `await`?

5. Как создать дочернюю задачу?
6. Почему рекомендуется избегать ключевого слова `lock`?
7. В каких случаях следует использовать класс `Interlocked`?
8. Когда класс `Mutex` предпочтительнее класса `Monitor`?
9. Каковы преимущества применения ключевых слов `async` и `await` на стороне сервера в веб-приложениях и веб-сервисах?
10. Можно ли отменить задачу? Если да, то каким образом?

Упражнение 5.2. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-5---multitasking-and-concurrency.

Упражнение 5.3. Узнайте больше о параллельном программировании

В книге Элвина Эшкрофта *Parallel Programming and Concurrency with C# 10 and .NET 6: A modern approach to building faster, more responsive and asynchronous .NET applications using C#* более подробно рассматриваются темы этой главы: https://www.packtpub.com/product/parallel-programming-and-concurrency-with-c-10-andnet-6/9781803243672.

Резюме

Из этой главы вы узнали о том, как:

- определить и запустить задачу;
- дождаться завершения задач;
- контролировать порядок выполнения задач;
- синхронизировать доступ к общим ресурсам;
- использовать ключевые слова `async` и `await` для асинхронного выполнения задач.

В следующей главе вы познакомитесь с популярными библиотеками сторонних разработчиков.

6 Популярные сторонние библиотеки

В этой главе рассматриваются популярные сторонние библиотеки для .NET, расширяющие возможности платформы, дополняя или улучшая встроенный функционал. Вы узнаете, как работать с изображениями с помощью *ImageSharp*, вести журналирование в *Serilog*, преобразовывать объекты с *AutoMapper*, писать модульные тесты с *FluentAssertions*, проверять данные с *FluentValidation* и создавать PDF-документы в *QuestPDF*.

Наиболее популярные сторонние библиотеки

Выбирая, какие сторонние библиотеки включить в эту книгу, я проанализировал самые скачиваемые на сайте https://www.nuget.org/stats/packages (табл. 6.1).

Таблица 6.1. Наиболее популярные (и не только) NuGet-пакеты

Рейтинг	Пакет	Скачиваний
1	`newtonsoft.json`	143 405 809
2	`serilog`	59 377 136
3	`awssdk.core`	39 346 965
4	`polly`	31 018 224
5	`google.protobuf`	30 760 313
6	`swashbuckle.aspnetcore.swaggergen`	27 862 122
7	`swashbuckle.aspnetcore.swagger`	27 489 106
8	`castle.core`	27 312 705
9	`grpc.core.api`	26 391 956
10	`newtonsoft.json.bson`	23 265 531
20	`moq`	20 198 273

Продолжение ⇨

Таблица 6.1 *(продолжение)*

Рейтинг	Пакет	Скачиваний
21	`humanizer.core`	20 001 363
23	`fluentvalidation`	19 328 210
24	`automapper`	19 181 472
39	`fluentassertions`	14 432 213
42	`dapper`	13 577 872
68	`rabbitmq.client`	8 593 335
92	`hangfire.core`	7 302 962
--	`nodatime`	921 719

О чем рассказывается в моих книгах

В моей книге, посвященной C# 12 и .NET 8, рассматриваются обработка JSON-документов с помощью `newtonsoft.json` и документирование веб-сервисов с использованием `swashbuckle`.

На данный момент такие темы, как создание динамических прокси и типизированных словарей с помощью Castle Core, а также развертывание и интеграция с *Amazon Web Services (AWS)*, выходят за рамки книги.

Выбор библиотек для этой главы основан не только на их популярности, но и на вопросах читателей, а также на практической ценности. В список вошли следующие библиотеки:

- для работы с изображениями — *ImageSharp*;
- для работы с текстом — *Humanizer*;
- для журналирования — *Serilog*;
- для сопоставления объектов — *AutoMapper*;
- для модульного тестирования — *FluentAssertions*;
- для проверки данных — *FluentValidation*;
- для создания PDF-файлов — *QuestPDF* (с открытым исходным кодом).

В главе 7 рассматривается самая популярная библиотека для работы с датами и временем — *Noda Time*.

В главе 9 представлено несколько других востребованных решений:

- для обеспечения отказоустойчивости и обработки временных сбоев — *Polly*;
- для планирования заданий и внедрения фоновых сервисов — *Hangfire*;
- для распределенного кэширования — *Redis*;
- для работы с очередями — *RabbitMQ*.

Работа с изображениями

ImageSharp — сторонняя кросс-платформенная библиотека для работы с 2D-графикой. Еще до выпуска .NET Core 1.0 разработчики критиковали отсутствие пространства имен `System.Drawing`, предназначенного для работы с изображениями. Чтобы восполнить этот пробел в современных .NET-приложениях, был создан проект *ImageSharp*.

В официальной документации к `System.Drawing` указано:

> «Некоторые типы в System.Drawing пространстве имен используют GDI+, который не поддерживается в службах Windows, приложениях ASP.NET Core и ASP.NET. Если вы не можете использовать System.Drawing с приложением, рекомендуемые варианты: ImageSharp, SkiaSharp, компоненты образов Windows и Microsoft.Maui.Graphics».

В марте 2023 года компания Six Labors выпустила версию ImageSharp 3.0. Сейчас для работы этого пакета требуется платформа .NET 6 или более новая версия, а будущие основные обновления будут ориентированы на LTS-релизы, такие как .NET 8. Подробнее об этом можно прочитать по адресу https://sixlabors.com/posts/announcing-imagesharp-300/.

Генерация полутоновых изображений

Давайте посмотрим, на что способен пакет ImageSharp.

1. В редакторе кода создайте консольное приложение со следующими настройками:
   - выберите шаблон проекта Console App (Консольное приложение);
   - в поле Solution name (Имя решения) укажите значение `Chapter06`;
   - в поле Project name (Имя проекта) укажите значение `WorkingWithImages`;
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
   - снимите флажок Enable native AOT publishing (Включить публикацию Native AOT).
2. В проекте `WorkingWithImages` создайте папку `images` и скачайте в нее девять изображений со страницы по адресу https://github.com/markjprice/apps-services-net8/tree/master/images/Categories.
3. В Visual Studio 2022 папку `images` с содержимым необходимо перенести в каталог `WorkingWithImages\bin\Debug\net8`, откуда будет запускаться скомпилированное консольное приложение. В Visual Studio доступно и автоматизированное решение этой задачи:
   1) на панели Solution Explorer (Обозреватель решений) выделите все девять изображений;

2) на панели Properties (Свойства) в раскрывающемся списке Copy to Output Directory (Копировать в выходной каталог) выберите пункт Copy always (Всегда копировать) (рис. 6.1);

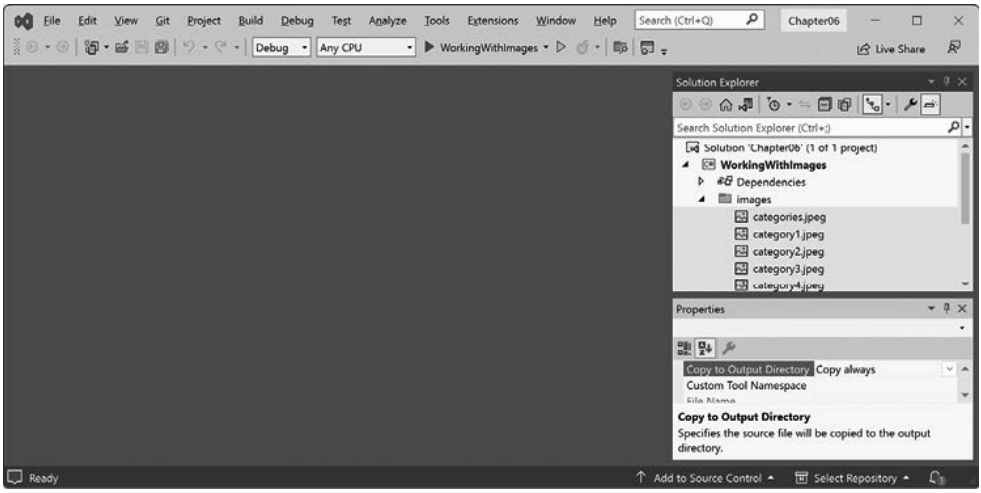


Рис. 6.1. Настройка автоматизированного копирования изображений в выходной каталог

3) откройте файл проекта и обратите внимание на содержимое элемента `<ItemGroup>`, согласно которому девять изображений будут копироваться в нужную папку:

```
<ItemGroup>
  <None Update="images\categories.jpeg">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  <None Update="images\category1.jpeg">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
...
```

4. В файле проекта `WorkingWithImages` настройте строгую обработку предупреждений, статически импортируйте класс `System.Console` на глобальном уровне и подключите пакет `SixLabors.ImageSharp`:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
```

```
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>

  <ItemGroup>
    <Using Include="System.Console" Static="true" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="SixLabors.ImageSharp" Version="3.0.2" />
  </ItemGroup>
...
```

Чтобы не перегружать код, далее я не буду показывать код для обработки предупреждений, а также статический импорт класса System.Console на глобальном уровне. Я сосредоточусь на элементах ItemGroup и PackageReference, относящихся к подключаемым библиотекам.

5. Выполните сборку проекта `WorkingWithImages`.
6. В Visual Studio 2022: на панели Solution Explorer (Обозреватель решений) нажмите кнопку Show All Files (Показать все файлы).
7. Перейдите в папку `obj\Debug\net8.0`, откройте файл `WorkingWithImages.GlobalUsings.g.cs` и обратите внимание, что в ссылке `SixLabors.ImageSharp` в дополнение к стандартным пространствам имен, предоставляемым .NET SDK, указаны три глобальных:

```
// <автоматически сгенерированы/>
global using global::SixLabors.ImageSharp;
global using global::SixLabors.ImageSharp.PixelFormats;
global using global::SixLabors.ImageSharp.Processing;
...
```

Если вы используете более старую версию SixLabors.ImageSharp, например 2.0.0, указанные пространства имен не добавляются автоматически и их придется импортировать вручную в каждом файле. Эта особенность — одна из причин, почему, начиная с версии ImageSharp 3.0, требуется платформа .NET версии не ниже 6.

8. Удалите содержимое файла `Program.cs`, а затем добавьте код, который преобразует все изображения в папке `images` в миниатюры с оттенками серого (полутоновые) размером 1/10 от оригинальных файлов:

```
string imagesFolder = Path.Combine(
  Environment.CurrentDirectory, "images");

WriteLine($"I will look for images in the following folder:\
n{imagesFolder}");
WriteLine();
```

```
if (!Directory.Exists(imagesFolder))
{
  WriteLine();
  WriteLine("Folder does not exist!");
  return;
}

IEnumerable<string> images =
  Directory.EnumerateFiles(imagesFolder);

foreach (string imagePath in images)
{
  if (Path.GetFileNameWithoutExtension(imagePath).EndsWith("-thumbnail"))
  {
    WriteLine($"Skipping:\n {imagePath}");
    WriteLine();
    continue; // Этот файл уже был конвертирован
  }

  string thumbnailPath = Path.Combine(
    Environment.CurrentDirectory, "images",
    Path.GetFileNameWithoutExtension(imagePath)
    + "-thumbnail" + Path.GetExtension(imagePath));

  using (Image image = Image.Load(imagePath))
  {
    WriteLine($"Converting:\n {imagePath}");
    WriteLine($"To:\n {thumbnailPath}");
    image.Mutate(x => x.Resize(image.Width / 10, image.Height / 10));
    image.Mutate(x => x.Grayscale());
    image.Save(thumbnailPath);
    WriteLine();
  }
}

WriteLine("Image processing complete.");

if (OperatingSystem.IsWindows())
{
  Process.Start("explorer.exe", imagesFolder);
}
else
{
  WriteLine("View the images folder.");
}
```

9. Запустите консольное приложение и убедитесь, что изображения преобразованы в полутоновые копии уменьшенного размера (ниже показан фрагмент вывода):

```
I will look for images in the following folder:
C:\apps-services-net8\Chapter06\WorkingWithImages\bin\Debug\net8.0\images

Converting:
  C:\apps-services-net8\Chapter06\WorkingWithImages\bin\Debug\net8.0\images\
categories.jpeg
To:
  C:\apps-services-net8\Chapter06\WorkingWithImages\bin\Debug\net8.0\images\
categories-thumbnail.jpeg

Converting:
  C:\apps-services-net8\Chapter06\WorkingWithImages\bin\Debug\net8.0\images\
category1.jpeg
To:
  C:\apps-services-net8\Chapter06\WorkingWithImages\bin\Debug\net8.0\images\
category1-thumbnail.jpeg
...

Converting:
  C:\apps-services-net8\Chapter06\WorkingWithImages\bin\Debug\net8.0\images\
category8.jpeg
To:
  C:\apps-services-net8\Chapter06\WorkingWithImages\bin\Debug\net8.0\images\
category8-thumbnail.jpeg

Image processing complete.
```

10. Откройте папку с изображениями и обратите внимание, что полутоновые миниатюры значительно меньше по размеру в байтах. Пример показан на рис. 6.2.

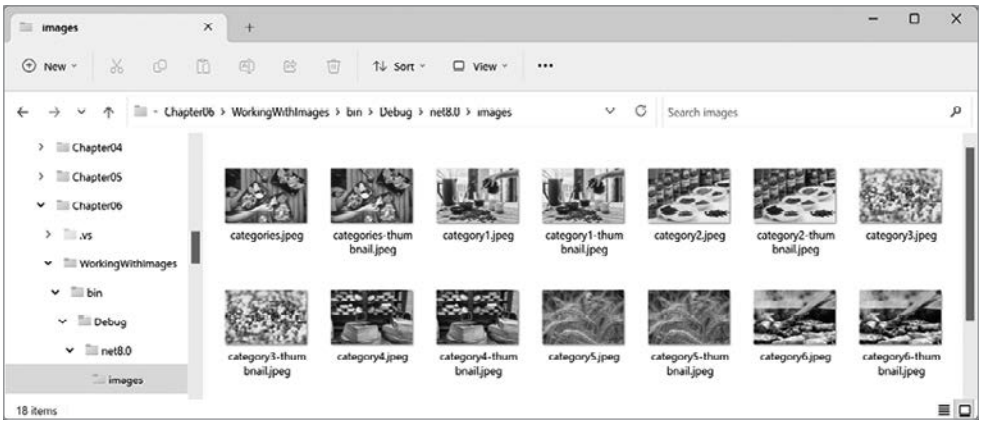


Рис. 6.2. Содержимое каталога после выполнения кода

Функционал ImageSharp для отрисовки и обработки изображений в Интернете

Инструмент ImageSharp сопровождается NuGet-пакетами для программной отрисовки графики и работы с изображениями в Интернете:

- `SixLabors.ImageSharp.Drawing`
- `SixLabors.ImageSharp.Web`

В копилку знаний: более подробную информацию вы найдете по адресу https://docs.sixlabors.com/.

Обработка строк и чисел с помощью Humanizer

С помощью библиотеки Humanizer можно обрабатывать текст типа `string`, имена значений `enum`, даты, время, числа и количественные выражения.

Работа с текстом

Встроенный тип `string` поддерживает методы `Substring` и `Trim` для обработки текста, но во многих случаях их недостаточно. Например:

- если код генерирует нечитабельную строку, ее нужно преобразовать в удобный для пользователя вид. Это особенно актуально для значений `enum`, в которых нельзя использовать пробелы;
- в системе управления веб-контентом может потребоваться превратить заголовок статьи в формат, подходящий для URL-адреса;
- длинный текст иногда нужно сокращать для отображения в ограниченном пространстве, например в мобильном интерфейсе.

Преобразование регистра

Сложные преобразования можно выполнять последовательно, передавая методу `Transform` ряд преобразований из библиотеки Humanizer. Преобразования реализуют интерфейсы `IStringTransformer` и `ICulturedStringTransformer`, так что вы можете настраивать собственные преобразования.

Все встроенные преобразования изменяют регистр строк. Методы, принимающие тип `string`, перечислены в табл. 6.2 вместе с примерами текста.

Дельный совет: важно учитывать регистр исходного текста. Если текст изначально набран прописными буквами, методы To.TitleCase и To.SentenceCase не изменят регистр символов. В этом случае перед преобразованием сначала переведите строку в нижний регистр.

Таблица 6.2. Преобразование регистра символов с помощью библиотеки Humanizer

Метод	Описание	Пример
`To.LowerCase`	Преобразует все символы в строке в нижний регистр	кошка сидела на коврике
`To.UpperCase`	Преобразует все символы в строке в верхний регистр	КОШКА СИДЕЛА НА КОВРИКЕ
`To.TitleCase`	Преобразует первый символ каждого слова в строке в верхний регистр	Кошка Сидела На Коврике
`To.SentenceCase`	Преобразует первый символ в строке в верхний регистр. Однако этот метод не учитывает точки, поэтому предложения он не распознает!	Кошка сидела на коврике.

Помимо метода `Transform`, существуют удобные методы для изменения регистра, например `To.TitleCase` (табл. 6.3).

Таблица 6.3. Методы расширения текста в Humanizer

Метод	Описание
`Titleize`	Аналог `To.TitleCase`
`Pascalize`	Преобразует строки в PascalCase («паскальный» регистр), удаляя подчеркивания
`Camelize`	Делает то же самое, но первая буква остается строчной (camelCase, верблюжий регистр)

Преобразование пробелов

Существуют удобные методы обработки пробелов в тексте, добавления символов подчеркивания и дефисов (табл. 6.4).

Таблица 6.4. Методы преобразования пробелов в Humanizer

Метод	Описание
`Underscore`	Заменяет пробелы между словами в строке символами подчеркивания и проводит текст к нижнему регистру
`Dasherize`, `Hyphenate`	Заменяет символы подчеркивания дефисами
`Kebaberize`	Разделяет слова дефисами

Методы преобразования форм слов

Библиотека Humanizer также предоставляет методы для автоматического преобразования форм слов из единственного во множественное число и наоборот (табл. 6.5).

Таблица 6.5. Методы Singularize и Pluralize библиотеки Humanizer

Метод	Описание
`Singularize`	Преобразует слово во множественном числе в единственное
`Pluralize`	Преобразует слово в единственном числе во множественное

Эти методы используются в Entity Framework Core для автоматического преобразования имен классов сущностей и их членов.

Пример обработки текста в консольном приложении

Рассмотрим несколько примеров работы с текстом с помощью библиотеки Humanizer.

1. В редакторе кода создайте консольное приложение `HumanizingData` и поместите его в решение `Chapter`. В Visual Studio 2022 присвойте параметру Startup Project (Запускаемый проект) значение Current selection (Текущий вариант).
2. В файле проекта `HumanizingData` настройте строгую обработку предупреждений, статически импортируйте класс `System.Console` на глобальном уровне и подключите пакет `Humanizer`:

```
<ItemGroup>
  <PackageReference Include="Humanizer" Version="2.14.1" />
</ItemGroup>
```

В данном проекте мы подключаем пакет Humanizer с поддержкой всех языков. Если нужен только английский, используйте строку Humanizer.Core. Для определенного языка укажите соответствующий пакет, например Humanizer.Core.fr для французского.

3. Выполните сборку проекта `HumanizingData` для восстановления зависимостей.
4. Добавьте в проект файл класса `Program.Functions.cs`.
5. В нем импортируйте пространство имен для работы с глобализацией и определите метод настройки консоли, который упростит смену текущей культуры и обеспечит корректное отображение специальных символов:

```
using System.Globalization; // Для класса CultureInfo

partial class Program
{
  private static void ConfigureConsole(string culture = "en-US")
  {
    // Для использования специальных символов типа ... (многоточие)
    OutputEncoding = System.Text.Encoding.UTF8;

    Thread t = Thread.CurrentThread;
    t.CurrentCulture = CultureInfo.GetCultureInfo(culture);
    t.CurrentUICulture = t.CurrentCulture;

    WriteLine("Current culture: {0}", t.CurrentCulture.DisplayName);
    WriteLine();
  }
}
```

6. Удалите содержимое файла `Program.cs` и вызовите метод `ConfigureConsole`:

```
ConfigureConsole(); // По умолчанию установлена культура en-US
```

7. В файле `Program.Functions.cs` импортируйте пространство имен с методами расширения `Humanizer`:

```
using Humanizer; // Для методов расширения библиотеки Humanizer общего назначения
```

8. В том же файле реализуйте метод, который выводит строку и результаты ее обработки с использованием различных способов преобразования регистра:

```
private static void OutputCasings(string original)
{
  WriteLine("Original casing: {0}", original);
  WriteLine("Lower casing: {0}", original.Transform(To.LowerCase));
  WriteLine("Upper casing: {0}", original.Transform(To.UpperCase));
  WriteLine("Title casing: {0}", original.Transform(To.TitleCase));
  WriteLine("Sentence casing: {0}", original.Transform(To.SentenceCase));
  WriteLine("Lower, then Sentence casing: {0}",
    original.Transform(To.LowerCase, To.SentenceCase));
  WriteLine();
}
```

9. В файле `Program.cs` вызовите метод `OutputCasings` с тремя разными строками:

```
OutputCasings("The cat sat on the mat.");
OutputCasings("THE CAT SAT ON THE MAT.");
OutputCasings("the cat sat on the mat. the frog jumped.");
```

10. Выполните код и проанализируйте результат:

```
Current culture: English (United States)

Original casing: The cat sat on the mat.
Lower casing: the cat sat on the mat.
Upper casing: THE CAT SAT ON THE MAT.
Title casing: The Cat Sat on the Mat.
Sentence casing: The cat sat on the mat.
Lower, then Sentence casing: The cat sat on the mat.

Original casing: THE CAT SAT ON THE MAT.
Lower casing: the cat sat on the mat.
Upper casing: THE CAT SAT ON THE MAT.
Title casing: THE CAT SAT ON THE MAT.
Sentence casing: THE CAT SAT ON THE MAT.
Lower, then Sentence casing: The cat sat on the mat.

Original casing: the cat sat on the mat. the frog jumped.
Lower casing: the cat sat on the mat. the frog jumped.
Upper casing: THE CAT SAT ON THE MAT. THE FROG JUMPED.
Title casing: The Cat Sat on the Mat. the Frog Jumped.
Sentence casing: The cat sat on the mat. the frog jumped.
Lower, then Sentence casing: The cat sat on the mat. the frog jumped.
```

11. В файле `Program.Functions.cs` добавьте метод, форматирующий нечитабельную строку с помощью различных методов расширения `Humanizer`:

```
private static void OutputSpacingAndDashes()
{
  string ugly = "ERROR_MESSAGE_FROM_SERVICE";

  WriteLine("Original string: {0}", ugly);

  WriteLine("Humanized: {0}", ugly.Humanize());

  // Класс LetterCasing устарел и в скором времени будет удален
  WriteLine("Humanized, lower case: {0}",
    ugly.Humanize(LetterCasing.LowerCase));

  // Вместо него используйте класс Transform для изменения регистра
  WriteLine("Transformed (lower case, then sentence case): {0}",
    ugly.Transform(To.LowerCase, To.SentenceCase));

  WriteLine("Humanized, Transformed (lower case, then sentence case): {0}",
    ugly.Humanize().Transform(To.LowerCase, To.SentenceCase));
}
```

12. В файле `Program.cs` закомментируйте предыдущие вызовы и добавьте вызов нового метода:

```
/*
OutputCasings("The cat sat on the mat.");
OutputCasings("THE CAT SAT ON THE MAT.");
OutputCasings("the cat sat on the mat. the frog jumped.");
*/

OutputSpacingAndDashes();
```

13. Выполните код и проанализируйте результат:

```
Original string: ERROR_MESSAGE_FROM_SERVICE
Humanized: ERROR MESSAGE FROM SERVICE
Humanized, lower case: error message from service
Transformed (lower case, then sentence case): Error_message_from_service
Humanized, Transformed (lower case, then sentence case): Error message from
service
```

14. Добавьте в проект файл `WondersOfTheAncientWorld.cs`.
15. Определите в нем перечисление `WondersOfTheAncientWorld`:

```
namespace Packt.Shared;

public enum WondersOfTheAncientWorld : byte
{
  None                      = 0b_0000_0000, // то есть 0
  GreatPyramidOfGiza        = 0b_0000_0001, // то есть 1
  HangingGardensOfBabylon   = 0b_0000_0010, // то есть 2
  StatueOfZeusAtOlympia     = 0b_0000_0100, // то есть 4
  TempleOfArtemisAtEphesus  = 0b_0000_1000, // то есть 8
  MausoleumAtHalicarnassus  = 0b_0001_0000, // то есть 16
  ColossusOfRhodes          = 0b_0010_0000, // то есть 32
  LighthouseOfAlexandria    = 0b_0100_0000  // то есть 64
}
```

16. Импортируйте это перечисление в файл `Program.Functions.cs`:

```
using Packt.Shared; // Для класса WondersOfTheAncientWorld
```

17. В этом же файле напишите метод, создающий переменную `favoriteAncient-Wonder` и выводящий ее имя с применением методов:

```
private static void OutputEnumNames()
{
  var favoriteAncientWonder = WondersOfTheAncientWorld.StatueOfZeusAtOlympia;

  WriteLine("Raw enum value name: {0}", favoriteAncientWonder);

  WriteLine("Humanized: {0}", favoriteAncientWonder.Humanize());
```

```
  WriteLine("Humanized, then Titleized: {0}",
    favoriteAncientWonder.Humanize().Titleize());

  WriteLine("Truncated to 8 characters: {0}",
    favoriteAncientWonder.ToString().Truncate(length: 8));

  WriteLine("Kebaberized: {0}",
    favoriteAncientWonder.ToString().Kebaberize());
}
```

18. В файле `Program.cs` закомментируйте предыдущий вызов метода и добавьте вызов метода `OutputEnumNames`:

```
//OutputSpacingAndDashes();

OutputEnumNames();
```

19. Выполните код и проанализируйте результат:

```
Raw enum value name: StatueOfZeusAtOlympia
Humanized: Statue of zeus at olympia
Humanized, then Titlerized: Statue of Zeus at Olympia
Truncated to 8 characters: StatueO…
Kebaberized: statue-of-zeus-at-olympia
```

Метод Truncate по умолчанию добавляет … (многоточие). Если сократить строку до восьми символов, он оставит семь и добавит многоточие. Можно передать другой символ, воспользовавшись перегруженной версией метода.

Работа с числами

Рассмотрим, как библиотека Humanizer может помочь при работе с числами.

1. В файле `Program.Functions.cs` определите метод, который будет генерировать несколько чисел, а затем выведет их с помощью различных методов расширения Humanizer:

```
private static void NumberFormatting()
{
  int number = 123;

  WriteLine($"Original number: {number}");
  WriteLine($"Roman: {number.ToRoman()}");
  WriteLine($"Words: {number.ToWords()}");
  WriteLine($"Ordinal words: {number.ToOrdinalWords()}");
  WriteLine();

  string[] things = { "fox", "person", "sheep", "apple", "goose",
    "oasis", "potato", "die", "dwarf", "attorney general", "biceps"};
```

```
  for (int i = 1; i <= 3; i++)
  {
    for (int j = 0; j < things.Length; j++)
    {
      Write(things[j].ToQuantity(i, ShowQuantityAs.Words));

      if (j < things.Length - 1) Write(", ");
    }
    WriteLine();
  }
  WriteLine();

  int thousands = 12345;
  int millions = 123456789;

  WriteLine("Original: {0}, Metric: About {1}", thousands,
    thousands.ToMetric(decimals: 0));

  WriteLine("Original: {0}, Metric: {1}", thousands,
    thousands.ToMetric(MetricNumeralFormats.WithSpace
      | MetricNumeralFormats.UseShortScaleWord,
      decimals: 0));

  WriteLine("Original: {0}, Metric: {1}", millions,
    millions.ToMetric(decimals: 1));
}
```

2. В файле `Program.cs` закомментируйте предыдущий вызов метода и добавьте вызов метода `NumberFormatting`:

```
//OutputEnumNames();

NumberFormatting();
```

3. Выполните код и проанализируйте результат:

```
Original number: 123
Roman: CXXIII
Words: one hundred and twenty-three
Ordinal words: hundred and twenty-third

one fox, one person, one sheep, one apple, one goose, one oasis, one potato,
one die, one dwarf, one attorney general, one bicep
two foxes, two people, two sheep, two apples, two geese, two oases, two
potatoes, two dice, two dwarves, two attorney generals, two biceps
three foxes, three people, three sheep, three apples, three geese, three oases,
three potatoes, three dice, three dwarves, three attorney generals, three
biceps

Original: 12345, Metric: About 12k
Original: 12345, Metric: About 12 thousand
Original: 123456789, Metric: 123.5M
```

4. В файле `Program.Functions.cs` импортируйте пространство имен для работы со словарями Humanizer:

```
using Humanizer.Inflections; // Для словарей
```

5. В верхней части метода `NumberFormatting` зарегистрируйте два неправильных слова:

```
Vocabularies.Default.AddIrregular("biceps", "bicepses");
Vocabularies.Default.AddIrregular("attorney general", "attorneys general");
```

6. Выполните код и убедитесь, что два неправильных слова теперь выводятся корректно.

Работа с датами и временем

Теперь рассмотрим, как Humanizer помогает при работе с датами и временем.

1. В файле `Program.Functions.cs` создайте метод для извлечения текущей даты и времени, а также нескольких дней, а затем выведите их с помощью методов расширения библиотеки Humanizer:

```
private static void DateTimeFormatting()
{
  DateTimeOffset now = DateTimeOffset.Now;

  // По умолчанию все сравнения Humanizer относятся к свойству Now (UTC)
  WriteLine($"Now (UTC): {now}");

  WriteLine("Add 3 hours, Humanized: {0}",
    now.AddHours(3).Humanize());

  WriteLine("Add 3 hours and 1 minute, Humanized: {0}",
    now.AddHours(3).AddMinutes(1).Humanize());

  WriteLine("Subtract 3 hours, Humanized: {0}",
    now.AddHours(-3).Humanize());

  WriteLine("Add 24 hours, Humanized: {0}",
    now.AddHours(24).Humanize());

  WriteLine("Add 25 hours, Humanized: {0}",
    now.AddHours(25).Humanize());

  WriteLine("Add 7 days, Humanized: {0}",
    now.AddDays(7).Humanize());

  WriteLine("Add 7 days and 1 minute, Humanized: {0}",
    now.AddDays(7).AddMinutes(1).Humanize());
```

```
  WriteLine("Add 1 month, Humanized: {0}",
    now.AddMonths(1).Humanize());

  WriteLine();

  // Примеры инициализации объекта TimeSpan
  int[] daysArray = { 12, 13, 14, 15, 16 };

  foreach (int days in daysArray)
  {
    WriteLine("{0} days, Humanized: {1}",
      days, TimeSpan.FromDays(days).Humanize());

    WriteLine("{0} days, Humanized with precision 2: {1}",
      days, TimeSpan.FromDays(days).Humanize(precision: 2));

    WriteLine("{0} days, Humanized with max unit days: {1}",
      days, TimeSpan.FromDays(days).Humanize(
        maxUnit: Humanizer.Localisation.TimeUnit.Day));

    WriteLine();
  }

  // Примеры обозначения значения часов
  TimeOnly[] times = { new TimeOnly(9, 0),
    new TimeOnly(9, 15), new TimeOnly(15, 30) };

  foreach (TimeOnly time in times)
  {
    WriteLine("{0}: {1}", time, time.ToClockNotation());
  }
}
```

2. В файле `Program.cs` закомментируйте предыдущий вызов метода и добавьте вызов метода `DateTimeFormatting`:

```
//NumberFormatting();

DateTimeFormatting();
```

3. Выполните код и проанализируйте результат:

```
Current culture: English (United States)

Now (UTC): 5/30/2023 8:12:51 AM +01:00
Add 3 hours, Humanized: 2 hours from now
Add 3 hours and 1 minute, Humanized: 3 hours from now
Subtract 3 hours, Humanized: 3 hours ago
Add 24 hours, Humanized: 23 hours from now
Add 25 hours, Humanized: tomorrow
Add 7 days, Humanized: 6 days from now
```

```
Add 7 days and 1 minute, Humanized: 7 days from now
Add 1 month, Humanized: one month from now

12 days, Humanized: 1 week
12 days, Humanized with precision 2: 1 week, 5 days
12 days, Humanized with max unit days: 12 days

13 days, Humanized: 1 week
13 days, Humanized with precision 2: 1 week, 6 days
13 days, Humanized with max unit days: 13 days

14 days, Humanized: 2 weeks
14 days, Humanized with precision 2: 2 weeks
14 days, Humanized with max unit days: 14 days

15 days, Humanized: 2 weeks
15 days, Humanized with precision 2: 2 weeks, 1 day
15 days, Humanized with max unit days: 15 days

16 days, Humanized: 2 weeks
16 days, Humanized with precision 2: 2 weeks, 2 days
16 days, Humanized with max unit days: 16 days

9:00 AM: nine o'clock
9:15 AM: a quarter past nine
3:30 PM: half past three
```

4. В файле `Program.cs` установите французский язык и регион для настройки консоли:

```
ConfigureConsole("fr-FR"); // По умолчанию используется культура en-US
```

5. Выполните код и убедитесь, что текст локализован на французский язык.

Журналирование с помощью Serilog

Хотя .NET включает встроенные фреймворки журналирования, сторонние инструменты, такие как Serilog (наиболее популярен), предоставляют больше возможностей и гибкости благодаря использованию *структурированных данных*.

Структурированные данные о событиях

В большинстве систем в журналах регистрируются обычные текстовые сообщения. Инструмент Serilog позволяет записывать сериализованные структурированные данные. Символ `@` перед параметром указывает, что нужно сериализовать объект, а не просто зарегистрировать результат вызова метода `ToString`. Эти данные можно затем использовать для эффективного поиска и сортировки в журналах. Например:

```
var lineitem = new { ProductId = 11, UnitPrice = 25.49, Quantity = 3 };

log.Information("Added {@LineItem} to shopping cart.", lineitem);
```

Подробнее об обработке структурированных данных библиотекой Serilog вы можете узнать по адресу https://github.com/serilog/serilog/wiki/Structured-Data.

Приемники журналов Serilog

Все системы журналирования должны куда-то сохранять записи журнала. Это может быть вывод на консоль, в файл или более сложные хранилища данных, такие как реляционные базы данных и облачные репозитории. В библиотеке Serilog такие средства называются *приемниками*.

Библиотека Serilog поддерживает множество пакетов приемников для записи журналов. Для их использования нужно подключить соответствующие пакеты. Вот наиболее популярные из них:

- `serilog.sinks.file`
- `serilog.sinks.console`
- `serilog.sinks.periodicbatching`
- `serilog.sinks.debug`
- `serilog.sinks.rollingfile` (устарел; вместо него используйте пакет `serilog.sinks.file`)
- `serilog.sinks.applicationinsights`
- `serilog.sinks.mssqlserver`

В феврале 2025 года на сайте NuGet числилось 563 пакета приемника: https://www.nuget.org/packages?q=serilog.sinks.

Журналирование в консоль и файл

Приступим.

1. В редакторе кода создайте консольное приложение `Serilogging` и добавьте его в решение `Chapter06`.
2. В файле проекта `Serilogging` добавьте строгую обработку предупреждений, статически импортируйте класс `System.Console` на глобальном уровне и подключите пакет Serilog, в том числе приемники для вывода в консоль и в файл (с планируемым журналированием):

```
<ItemGroup>
  <PackageReference Include="Serilog" Version="3.1.1" />
  <PackageReference Include="Serilog.Sinks.Console" Version="5.0.0" />
  <PackageReference Include="Serilog.Sinks.File" Version="5.0.0" />
</ItemGroup
```

3. Выполните сборку проекта `Serilogging`.
4. В этом проекте создайте папку `Models`.
5. Добавьте в нее файл класса `ProductPageView.cs` и измените его содержимое следующим образом:

```
namespace Serilogging.Models;

public class ProductPageView
{
  public int ProductId { get; set; }
  public string? PageTitle { get; set; }
  public string? SiteSection { get; set; }
}
```

6. Удалите содержимое файла `Program.cs`, а затем напишите операторы импорта пространств имен, необходимые для работы с библиотекой Serilog:

```
using Serilog; // Для элементов Log, LoggerConfiguration, RollingInterval
using Serilog.Core; // Для интерфейса Logger
using Serilogging.Models; // Для класса ProductPageView
```

7. В файле `Program.cs` подготовьте конфигурацию Serilog для записи событий в консоль и файл, а также настройте создание нового файла журнала ежедневно и многоуровневую запись:

```
// Создание нового интерфейса Logger, который будет выполнять запись
// в консоль и в текстовый файл ежедневно по одному файлу с текущей
// датой в качестве имени.
using Logger log = new LoggerConfiguration()
    .WriteTo.Console()
    .WriteTo.File("log.txt", rollingInterval: RollingInterval.Day)
    .CreateLogger();

// Назначение нового интерфейса статической точке входа для журналирования
Log.Logger = log;
Log.Information("The global logger has been configured.");

// Занесите в журнал несколько примеров записей разной степени важности
Log.Warning("Danger, Serilog, danger!");
Log.Error("This is an error!");
Log.Fatal("Fatal problem!");

ProductPageView pageView = new() {
  PageTitle = "Chai",
  SiteSection = "Beverages",
  ProductId = 1 };

Log.Information("{@PageView} occurred at {Viewed}",
  pageView, DateTimeOffset.UtcNow);
```

```
// Для журнала с буфером, например компонента журналирования текстовых
// файлов, необходимо выполнять полную очистку перед завершением приложения
Log.CloseAndFlush();
```

8. Запустите консольное приложение и убедитесь, что сообщения выводятся как в консоль, так и в файл:

```
[07:09:43 INF] The global logger has been configured.
[07:09:43 WRN] Danger, Serilog, danger!
[07:09:43 ERR] This is an error!
[07:09:43 FTL] Fatal problem!
[07:09:43 INF] {"ProductId": 1, "PageTitle": "Chai", "SiteSection":
"Beverages", "$type": "ProductPageView"} occurred at 09/07/2023 15:08:44 +00:00
```

9. Откройте файл вида `log`*`ГГГГММДД`*`.txt`, где *`ГГГГ`* — год, *`ММ`* — месяц, *`ДД`* — день, и обратите внимание, что он содержит те же сообщения.
   - Если вы используете Visual Studio 2022, файл журнала размещается в каталоге `Serilogging\bin\Debug\net8.0`.
   - В случае применения Visual Studio Code и команды `dotnet run` файл журнала размещается в папке `Serilogging`.

В копилку знаний: более подробную информацию вы найдете по адресу https://serilog.net/.

Дельный совет: отключайте ведение журнала, если оно не требуется. Журналирование может сильно увеличить расходы. Например, одна компания тратила незапланированные 10 тысяч долларов в месяц на облачные ресурсы. Оказалось, что каждый SQL-запрос в их рабочей среде фиксировался на сервере. После отключения журналирования экономия составила 8500 долларов в месяц!

Сопоставление объектов

Одна из рутинных задач программистов — это сопоставление объектов. Часто возникает необходимость интегрировать системы или компоненты с концептуально схожими объектами, но разными структурами.

Модели данных различаются в зависимости от уровня приложения. Представляющие данные в хранилище называются *моделями сущностей*. Предназначенные для передачи данных между уровнями — *объектами передачи данных (data transfer object, DTO)*. Предназначенные для отображения данных пользователю — это *модели представлений*. Все они могут иметь схожие элементы, но их структуры различны.

Для сопоставления объектов нужен специальный инструмент. *AutoMapper* — один из них, упрощающий решение данной задачи благодаря встроенной системе соглашений. Например, обнаруженное им свойство `CompanyName` будет автоматически сопоставлено с аналогичным свойством с тем же именем в целевом классе.

Создатель библиотеки AutoMapper Джимми Богард написал статью о философии дизайна, которую можно прочитать на сайте https://jimmybogard.com/automappers-design-philosophy/.

Рассмотрим пример работы AutoMapper. Мы создадим четыре проекта.

- Библиотеку классов для моделей сущностей и представлений.
- Библиотеку классов для конфигурации сопоставления, чтобы использовать ее в тестах и реальных проектах.
- Проект для модульного тестирования сопоставлений.
- Консольное приложение для выполнения сопоставления в реальном времени.

Мы создадим модель, представляющую клиента интернет-магазина и его корзину, а затем сопоставим ее с моделью сводного представления для отображения пользователю.

Определение моделей для конфигурации AutoMapper

Для проверки сопоставления определим несколько типов записей (`record`). Напоминаю, что запись (или класс записи — `record class`) — это ссылочный тип, который, в отличие от класса (`class`), сравнивается по значениям, а не по адресу памяти (кроме строки, которая переопределяет это поведение).

Перед использованием рекомендуется проверить конфигурацию сопоставлений, поэтому начнем с определения моделей и сопоставлений между ними, а затем создадим модульный тест.

1. В редакторе кода создайте проект библиотеки классов `MappingObjects.Models` (Class Library (Библиотека классов)) и добавьте его в решение `Chapter06`.
2. В этом проекте удалите файл `Class1.cs`.
3. В том же проекте, `MappingObjects.Models`, создайте файл класса `Customer.cs` и измените его содержимое, с помощью синтаксиса конструктора определив неизменяемый тип записи `Customer`:

```
namespace Northwind.EntityModels;

// У этой записи будет только конструктор с указанными ниже параметрами.
// Объекты будут неизменяемыми после инстанцирования с помощью конструктора.
// У записи не будет конструктора без параметров по умолчанию.
public record class Customer(
```

```
  string FirstName,
  string LastName
);
```

4. В том же проекте создайте файл класса `LineItem.cs` и измените его содержимое следующим образом:

```
namespace Northwind.EntityModels;

public record class LineItem(
  string ProductName,
  decimal UnitPrice,
  int Quantity
);
```

5. В том же проекте создайте файл класса `Cart.cs` и измените его содержимое следующим образом:

```
namespace Northwind.EntityModels;

public record class Cart(
  Customer Customer,
  List<LineItem> Items
);
```

6. В том же проекте создайте файл класса `Summary.cs` в пространстве имен `Northwind.ViewModels` (не `Northwind.EntityModels`!), удалите весь код, а затем определите тип записи с публичными свойствами, которые можно установить при инициализации:

```
namespace Northwind.ViewModels;

public record class Summary
{
  // Эти свойства могут быть инициализированы один раз,
  // но затем изменяться не будут
  public string? FullName { get; init; }
  public decimal Total { get; init; }

  // Этот класс записи будет иметь конструктор по умолчанию без параметров.
  // Следующее закомментированное выражение генерируется автоматически:
  // public Summary() { }
}
```

Для моделей сущностей мы используем типы record class с синтаксисом конструктора, чтобы сделать их неизменяемыми. Для модели Summary требуется конструктор без параметров по умолчанию, так как экземпляры этой модели будут инициализироваться с помощью библиотеки AutoMapper. Поэтому это должен быть класс записи с публичными свойствами, настраиваемыми во время инициализации, но не после нее. Для этого указывается ключевое слово init.

Определение сопоставлений для конфигурации AutoMapper

Теперь определим сопоставления между моделями.

1. В редакторе кода создайте проект библиотеки классов `MappingObjects.Mappers` и добавьте его в решение `Chapter06`.
2. В созданном проекте добавьте строгую обработку предупреждений, а также ссылки на пакет `AutoMapper` и проект `Models`:

```
<ItemGroup>
  <PackageReference Include="AutoMapper" Version="12.0.1" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include=
    "..\MappingObjects.Models\MappingObjects.Models.csproj" />
</ItemGroup>
```

3. Выполните сборку проекта `MappingObjects.Mappers` для компиляции и восстановления зависимостей.
4. В этом проекте удалите файл `Class1.cs`.
5. Затем создайте в нем файл класса `CartToSummaryMapper.cs` и настройте сопоставление, например, для элемента `FullName` из класса `Summary` с комбинацией объектов `FirstName` и `LastName` из класса `Customer`:

```
using AutoMapper; // Для класса MapperConfiguration
using AutoMapper.Internal; // Для метода расширения Internal()
using Northwind.EntityModels; // Для класса Cart
using Northwind.ViewModels; // Для класса Summary

namespace MappingObjects.Mappers;

public static class CartToSummaryMapper
{
  public static MapperConfiguration GetMapperConfiguration()
  {
    MapperConfiguration config = new(cfg =>
    {
      // Для исправления проблемы с методом MaxInteger:
      // https://github.com/AutoMapper/AutoMapper/issues/3988
      cfg.Internal().MethodMappingEnabled = false;
      // Настройка сопоставления с помощью проекций
      cfg.CreateMap<Cart, Summary>()
        // Сопоставление имени и фамилии, отформатированных в полное имя
        .ForMember(dest => dest.FullName, opt => opt.MapFrom(src =>
          string.Format("{0} {1}",
```

```
            src.Customer.FirstName,
            src.Customer.LastName)
        ));
    });
    return config;
  }
}
```

Тестирование конфигурации AutoMapper

После определения сопоставлений конфигурацию можно протестировать.

1. В редакторе кода создайте проект xUnit Test Project (Тестовый проект xUnit), присвойте ему имя `MappingObjects.Tests` и добавьте в решение `Chapter06`.
2. В созданном проекте добавьте ссылку на пакет `AutoMapper` и проект `Models`:

```
<ItemGroup>
  <PackageReference Include="AutoMapper" Version="12.0.1" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.8.0" />
```

3. Также добавьте ссылки на проекты `MappingObjects.Models` и `MappingObjects.Mappers`:

```
<ItemGroup>
  <ProjectReference Include=
    "..\MappingObjects.Mappers\MappingObjects.Mappers.csproj" />
  <ProjectReference Include=
    "..\MappingObjects.Models\MappingObjects.Models.csproj" />
</ItemGroup>
```

4. Выполните сборку проекта `MappingObjects.Tests` для компиляции и восстановления зависимостей.
5. Переименуйте файл `UnitTest1.cs` в `TestAutoMapperConfig.cs`.
6. В файле `TestAutoMapperConfig.cs` добавьте проверку конфигурации сопоставлений и утверждение о ее успешном выполнении:

```
using AutoMapper; // Для класса MapperConfiguration
using MappingObjects.Mappers; // Для объекта CartToSummaryMapper

namespace MappingObjects.Tests;

public class TestAutoMapperConfig
{
  [Fact]
  public void TestSummaryMapping()
  {
    MapperConfiguration config = CartToSummaryMapper.GetMapperConfiguration();
```

```
        config.AssertConfigurationIsValid();
    }
}
```

7. Запустите тесты.
   - В программе Visual Studio 2022 выполните команду меню Test ▸ Run All Tests (Тесты ▸ Запуск всех тестов).
   - В программе Visual Studio Code на панели Terminal (Терминал) выполните команду `dotnet test`.
8. Тест будет провален, так как член `Total` модели представления `Summary` не сопоставлен с соответствующим полем данных (рис. 6.3).

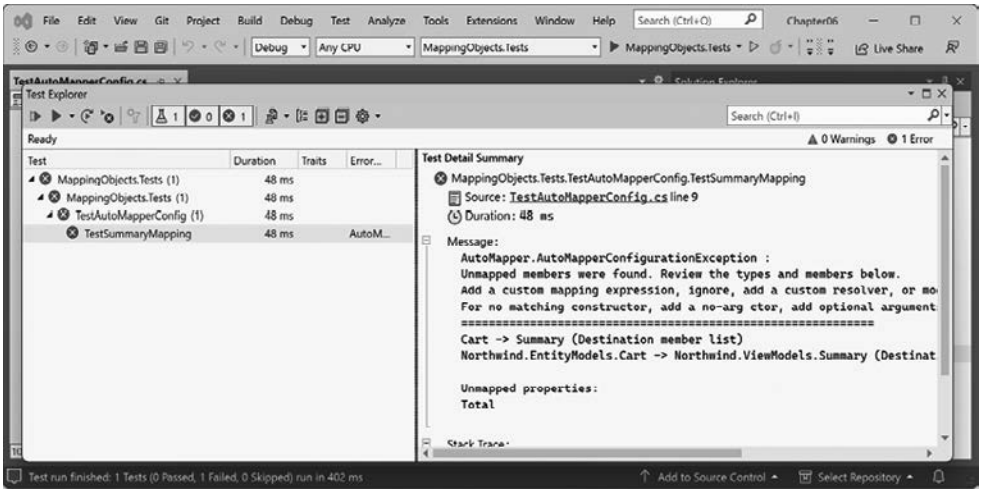


Рис. 6.3. Тест завершился неудачей, потому что член Total не сопоставлен

9. В файле проекта `MappingObjects.Mappers` в конфигурации сопоставления после настройки `FullName` добавьте сопоставление для `Total`:

```
cfg.CreateMap<Cart, Summary>()
  // Сопоставление имени и фамилии, отформатированных в полное имя
  .ForMember(dest => dest.FullName, opt => opt.MapFrom(src =>
    string.Format("{0} {1}",
      src.Customer.FirstName, src.Customer.LastName)
  )) // Мы убрали отсюда точку с запятой
  // Сопоставление суммы элементов с членом Total
  .ForMember(dest => dest.Total, opt => opt.MapFrom(
    src => src.Items.Sum(item => item.UnitPrice * item.Quantity)));
});
```

10. Запустите тест и обратите внимание, что на этот раз он выполнен успешно.

Выполнение сопоставления в консольном приложении

Теперь, когда конфигурация проверена, можно выполнить сопоставление в реальном времени.

1. В редакторе кода создайте консольное приложение `MappingObjects.Console` и добавьте его в решение `Chapter06`.
2. В файле созданного проекта добавьте строгую обработку предупреждений, статически импортируйте класс `System.Console` на глобальном уровне, добавьте ссылку на проекты `MappingObjects.Models` и `MappingObjects.Mappers` и подключите пакет `AutoMapper`:

```
<ItemGroup>
  <ProjectReference Include=
    "..\MappingObjects.Mappers\MappingObjects.Mappers.csproj" />
  <ProjectReference Include=
    "..\MappingObjects.Models\MappingObjects.Models.csproj" />
</ItemGroup>

<ItemGroup>
  <PackageReference Include="AutoMapper" Version="12.0.1" />
</ItemGroup>
```

3. Выполните сборку проекта `MappingObjects.Console`.
4. Удалите содержимое файла `Program.cs` и напишите код, создающий объектную модель, представляющую покупателя и его корзину с несколькими товарами. Добавьте код для сопоставления данных с моделью представления сводных данных для отображения пользователю

```
using AutoMapper; // Для класса MapperConfiguration, интерфейса IMapper
using MappingObjects.Mappers; // Для объекта CartToSummaryMapper
using Northwind.EntityModels; // Для сущностей Customer, Cart, LineItem
using Northwind.ViewModels; // Для класса Summary
using System.Text; // Для класса Encoding

// Установка кодировки вывода консоли на UTF-8 для поддержки
// символов Юникода, таких как знак евро
OutputEncoding = Encoding.UTF8;

// Создание объектной модели из типов моделей "сущность",
// поступающих из хранилища данных, например SQL Server

Cart cart = new(
  Customer: new(
    FirstName: "John",
    LastName: "Smith"
  ),
```

```
    Items: new()
    {
      new(ProductName: "Apples", UnitPrice: 0.49M, Quantity: 10),
      new(ProductName: "Bananas", UnitPrice: 0.99M, Quantity: 4)
    }
  );

  WriteLine("*** Original data before mapping.");
  WriteLine($"{cart.Customer}");
  foreach (LineItem item in cart.Items)
  {
    WriteLine($" {item}");
  }

  // Извлечение конфигурации сопоставления
  // для преобразования объекта Cart в Summary
  MapperConfiguration config = CartToSummaryMapper.GetMapperConfiguration();

  // Создание интерфейса сопоставления с помощью конфигурации
  IMapper mapper = config.CreateMapper();

  // Выполнение сопоставления
  Summary summary = mapper.Map<Cart, Summary>(cart);
  // Вывод результата
  WriteLine();
  WriteLine("*** After mapping.");
  WriteLine($"Summary: {summary.FullName} spent {summary.Total:C}.");
```

5. Запустите консольное приложение и обратите внимание на успешный результат:

```
*** Original data before mapping.
Customer { FirstName = John, LastName = Smith }
  LineItem { ProductName = Apples, UnitPrice = 0.49, Quantity = 10 }
  LineItem { ProductName = Bananas, UnitPrice = 0.99, Quantity = 4 }

*** After mapping.
Summary: John Smith spent £8.86.
```

6. По аналогии с предыдущим кодом можно написать модульный тест, проверяющий корректность имен и фамилий клиентов в объекте `Summary`.

Дельный совет: вокруг использования AutoMapper идут споры. Подробнее можно прочитать в статье по адресу https://www.anthonysteele.co.uk/AgainstAutoMapper.html, где также приведены дополнительные ссылки по теме.

В копилку знаний: подробная информация доступна по адресу https://automapper.org/.

Формулировка утверждений в модульном тестировании

Библиотека FluentAssertions предоставляет методы расширения, упрощающие написание и чтение кода в модульных тестах. Благодаря им утверждения становятся более понятными человеку и приближенными к естественному языку. Эта библиотека совместима с большинством фреймворков тестирования, включая xUnit.

При добавлении пакета `FluentAssertions` в проект автоматически определяется используемый фреймворк и на его основе настраивается генерация исключений.

Чтобы использовать `FluentAssertions`, импортируйте соответствующее пространство имен, вызовите метод `Should()` для переменной, а затем добавьте подходящие (из сотен доступных) методы расширения. Утверждения можно применять по отдельности или цепочками, объединяя их с помощью метода `And()`.

Формулировка утверждений для строковых значений

Начнем с формулировки утверждений для одного строкового значения.

1. В редакторе кода создайте проект xUnit Test Project (Тестовый проект xUnit), присвойте ему имя `FluentTests` и добавьте в решение `Chapter06`.
2. В файле созданного проекта подключите пакет `FluentAssertions`:

```
<ItemGroup>
  <PackageReference Include="FluentAssertions" Version="6.12.0" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.8.0" />
```

Актуальную версию пакета FluentAssertions можно узнать по адресу https://www.nuget.org/packages/FluentAssertions/.

3. Выполните сборку проекта `FluentTests`.
4. Переименуйте файл `UnitTest1.cs` в `FluentExamples.cs`.
5. В файле `FluentExamples.cs` импортируйте пространство имен для поддержки методов расширения библиотеки FluentAssertions, и напишите метод для проверки строкового значения:

```
using FluentAssertions; // Для распространенных методов Fluent Assertions

namespace FluentTests;

public class FluentExamples
{
  [Fact]
  public void TestString()
  {
```

```
    string city = "London";
    string expectedCity = "London";

    city.Should().StartWith("Lo")
      .And.EndWith("on")
      .And.Contain("do")
      .And.HaveLength(6);

    city.Should().NotBeNull()
      .And.Be("London")
      .And.BeSameAs(expectedCity)
      .And.BeOfType<string>();

    city.Length.Should().Be(6);
  }
}
```

6. Запустите тест.
   - В Visual Studio 2022 выполните команду меню Test ▸ Run All Tests (Тесты ▸ Запуск всех тестов).
   - В Visual Studio Code на панели Terminal (Терминал) выполните команду `dotnet test`.
7. Обратите внимание, что тест пройден успешно.
8. В методе `TestString` в значении переменной `city` удалите последнюю букву `n` в слове `London`.
9. Запустите тест и обратите внимание, что он завершился неудачей:

```
Expected city "Londo" to end with "on".
```

10. Добавьте букву `n` обратно.
11. Запустите тест еще раз. На этот раз он завершится успешно.

Формулировка утверждений для коллекций и массивов

Теперь перейдем к утверждениям для коллекций и массивов.

1. В файле `FluentExamples.cs` добавьте метод для тестирования утверждений коллекции:

```
[Fact]
public void TestCollections()
{
  string[] names = { "Alice", "Bob", "Charlie" };
```

```
  names.Should().HaveCountLessThan(4,
    "because the maximum items should be 3 or fewer");
  names.Should().OnlyContain(name => name.Length <= 6);
}
```

2. Запустите тесты и обратите внимание на неудачное завершение процесса тестирования коллекций:

```
Expected names to contain only items matching (name.Length <= 6), but
{"Charlie"} do(es) not match.
```

3. Замените слово `Charlie` на `Charly`.
4. Запустите тесты и убедитесь, что они завершились успешно.

Формулировка утверждений для дат и времени

Теперь займемся утверждениями для значений даты и времени.

1. В файле `FluentExamples.cs` подключите пространство имен, добавляющее методы для работы с именованными месяцами и другими полезными функциями, связанными с датой и временем:

```
using FluentAssertions.Extensions; // Для методов расширения February и March
```

2. Добавьте метод для тестирования утверждений даты/времени:

```
[Fact]
public void TestDateTimes()
{
  DateTime when = new(
    hour: 9, minute: 30, second: 0,
    day: 25, month: 3, year: 2024);

  when.Should().Be(25.March(2024).At(9, 30));

  when.Should().BeOnOrAfter(23.March(2024));

  when.Should().NotBeSameDateAs(12.February(2024));

when.Should().HaveYear(2024);

  DateTime due = new(
    hour: 11, minute: 0, second: 0,
    day: 25, month: 3, year: 2024);

  when.Should().BeAtLeast(2.Hours()).Before(due);
}
```

3. Запустите тесты. Они должны завершиться неудачно:

```
Expected when <2024-03-25 09:30:00> to be at least 2h before <2024-03-25
11:00:00>, but it is behind by 1h and 30m.
```

4. В переменной `due` измените значение параметра `hour` с `11` на `13`.
5. Запустите тесты и убедитесь, что проверка даты и времени прошла успешно.

В копилку знаний: подробную информацию о библиотеке FluentAssertions вы найдете по адресу https://fluentassertions.com/.

Проверка достоверности данных

Инструмент FluentValidation позволяет конструировать типизированные правила валидации в удобной и понятной форме. В процессе создания валидатора для определенного типа нужно унаследовать класс от `AbstractValidator<T>`, где `T` — это проверяемый тип. В конструкторе класса вызывается метод `RuleFor`, определяющий правила. Если правило нужно применять только в определенных случаях, используется метод `When`.

Встроенные валидаторы

В библиотеку FluentValidation встроено множество полезных методов валидации данных с возможностью настройки правил. Вот неполный список таких методов, с которыми вы познакомитесь в этом разделе:

- `Null`, `NotNull`, `Empty`, `NotEmpty`;
- `Equal`, `NotEqual`;
- `Length`, `MaxLength`, `MinLength`;
- `LessThan`, `LessThanOrEqualTo`, `GreaterThan`, `GreaterThanOrEqualTo`;
- `InclusiveBetween`, `ExclusiveBetween`;
- `ScalePrecision`;
- `Must` (делегат);
- `Matches` (регулярное выражение), `EmailAddress`, `CreditCard`;
- `IsInEnum`, `IsEnumName`.

Настройка валидации

Наипростейший способ создания пользовательских правил — тип `Predicate`, с помощью которого пишутся функции валидации. Для большего контроля можно воспользоваться методом `Custom`.

Уведомления валидации

Есть несколько методов расширения, позволяющих настроить вывод сообщений, когда данные не соответствуют правилам:

- `WithName` — изменение имени свойства в сообщении;
- `WithSeverity` — изменение уровня серьезности (по умолчанию `Error`) на `Warning` и т. д.;
- `WithErrorCode` — добавление в сообщение кода ошибки;
- `WithState` — добавление состояния, которое может быть использовано в сообщении;
- `WithMessage` — настройка формата сообщения о нарушении правил.

Создание модели и валидатора

Давайте попрактикуемся с библиотекой `FluentValidation` и создадим три проекта.

- Библиотеку классов для валидации модели, представляющей заказ клиента.
- Библиотеку классов для валидатора модели.
- Консольное приложение для тестирования валидации в реальном времени.

Выполните следующее.

1. В редакторе кода создайте проект библиотеки классов `FluentValidation.Models` и добавьте его в решение `Chapter06`.
2. В созданном проекте удалите файл `Class1.cs`.
3. Затем создайте файл класса `CustomerLevel.cs` и добавьте в него перечисление с уровнями клиентов: `Bronze`, `Silver`, `Gold`:

```
namespace FluentValidation.Models;

public enum CustomerLevel
{
  Bronze,
  Silver,
  Gold
}
```

4. Еще создайте файл класса `Order.cs` и добавьте в него свойства для модели заказа:

```
namespace FluentValidation.Models;

public class Order
{
  public long OrderId { get; set; }
```

```
  public string? CustomerName { get; set; }
  public string? CustomerEmail { get; set; }
  public CustomerLevel CustomerLevel { get; set; }
  public decimal Total { get; set; }
  public DateTime OrderDate { get; set; }
  public DateTime ShipDate { get; set; }
}
```

5. В редакторе кода создайте проект библиотеки классов `FluentValidation.Validators` и добавьте его в решение `Chapter06`.
6. Добавьте в него ссылки на проект `FluentValidation.Models` и на пакет `FluentValidation`:

```
<ItemGroup>
  <PackageReference Include="FluentValidation" Version="11.8.1" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include=
    "..\FluentValidation.Models\FluentValidation.Models.csproj" />
</ItemGroup>
```

7. Выполните сборку проекта `FluentValidation.Validators`.
8. Удалите в нем файл `Class1.cs`.
9. Затем создайте файл класса `OrderValidator.cs` со следующим кодом:

```
using FluentValidation.Models;

namespace FluentValidation.Validators;

public class OrderValidator : AbstractValidator<Order>
{
  public OrderValidator()
  {
    RuleFor(order => order.OrderId)
      .NotEmpty(); // Не default(long), что равно 0
    RuleFor(order => order.CustomerName)
      .NotNull()
      .WithName("Name"); // Использование в сообщениях Name
                         // вместо CustomerName

    RuleFor(order => order.CustomerName)
      .MinimumLength(5)
      .WithSeverity(Severity.Warning);

    RuleFor(order => order.CustomerEmail)
      .NotEmpty()
      .EmailAddress();
```

```
    RuleFor(order => order.CustomerLevel)
      .IsInEnum();

    RuleFor(order => order.Total)
      .GreaterThan(0);

    RuleFor(order => order.ShipDate)
      .GreaterThan(order => order.OrderDate);

    When(order => order.CustomerLevel == CustomerLevel.Gold, () =>
    {
      RuleFor(order => order.Total).LessThan(50M);
      RuleFor(order => order.Total).GreaterThanOrEqualTo(20M);
    }).Otherwise(() =>
    {
      RuleFor(order => order.Total).LessThan(20M);
    });
  }
}
```

Тестирование валидатора

Теперь, когда валидатор готов, можно создать консольное приложение для его тестирования.

1. В редакторе кода создайте консольное приложение `FluentValidation.Console` и добавьте его в решение `Chapter06`.
2. В файле созданного проекта добавьте строгую обработку предупреждений, статически импортируйте класс `System.Console` на глобальном уровне и добавьте ссылки на проекты `FluentValidation.Validators` и `FluentValidation.Models`:

```
<ItemGroup>
  <ProjectReference Include=
    "..\FluentValidation.Models\FluentValidation.Models.csproj" />
  <ProjectReference Include=
    "..\FluentValidation.Validators\FluentValidation.Validators.csproj" />
</ItemGroup>
```

3. Выполните сборку проекта `FluentValidation.Console` для восстановления зависимостей.
4. Удалите содержимое файла `Program.cs`, а затем добавьте в него код для создания заказа и его валидации:

```
using FluentValidation.Models; // Для класса Order
using FluentValidation.Results; // Для класса ValidationResult
using FluentValidation.Validators; // Для класса OrderValidator
```

```
using System.Globalization; // Для класса CultureInfo
using System.Text; // Для класса Encoding

OutputEncoding = Encoding.UTF8; // Добавление символа евро

// Управление культурой, используемой для форматирования дат
// и денежных единиц и для перевода сообщений об ошибках
// на местный язык
Thread t = Thread.CurrentThread;
t.CurrentCulture = CultureInfo.GetCultureInfo("en-US");
t.CurrentUICulture = t.CurrentCulture;
WriteLine($"Current culture: {t.CurrentCulture.DisplayName}");
WriteLine();

Order order = new()
{
  // Попытка начать с заведомо недействительного заказа
};

OrderValidator validator = new();

ValidationResult result = validator.Validate(order);

// Вывод данных о заказе
WriteLine($"CustomerName:  {order.CustomerName}");
WriteLine($"CustomerEmail: {order.CustomerEmail}");
WriteLine($"CustomerLevel: {order.CustomerLevel}");
WriteLine($"OrderId:       {order.OrderId}");
WriteLine($"OrderDate:     {order.OrderDate}");
WriteLine($"ShipDate:      {order.ShipDate}");
WriteLine($"Total:         {order.Total:C}");
WriteLine();

// Оповещение о допустимости заказа и нарушенных правилах
WriteLine($"IsValid: {result.IsValid}");
foreach (var item in result.Errors)
{
  WriteLine($" {item.Severity}: {item.ErrorMessage}");
}
```

5. Запустите консольное приложение и убедитесь, что нарушенные правила отображаются корректно:

```
Current culture: English (United States)

CustomerName:
CustomerEmail:
CustomerLevel: Bronze
OrderId:       0
```

```
OrderDate:     01/01/0001 12:00:00 AM
ShipDate:      01/01/0001 12:00:00 AM
Total:         $0.00

IsValid: False
  Error: 'Order Id' must not be empty.
  Error: 'Name' must not be empty.
  Error: 'Customer Email' must not be empty.
  Error: 'Total' must be greater than '0'.
  Error: 'Ship Date' must be greater than '01/01/0001 12:00:00 AM'.
```

6. Закомментируйте код культуры, чтобы вывести сообщения на своем языке и в своем регионе (например, на французском, если вы находитесь во Франции, укажите значение `fr-FR`):

```
Current culture: français (France)

CustomerName:
CustomerEmail:
CustomerLevel: Bronze
OrderId:       0
OrderDate:     01/01/0001 00:00:00
ShipDate:      01/01/0001 00:00:00
Total:         0,00 €

IsValid: False
  Error: 'Order Id' ne doit pas être vide.
  Error: 'Name' ne doit pas avoir la valeur null.
  Error: 'Customer Email' ne doit pas être vide.
  Error: 'Total' doit être plus grand que '0'.
  Error: 'Ship Date' doit être plus grand que '01/01/0001 00:00:00'.
```

7. Настройте свойства заказа:

```
Order order = new()
{
  OrderId = 10001,
  CustomerName = "Abc",
  CustomerEmail = "abc&example.com",
  CustomerLevel = (CustomerLevel)4,
  OrderDate = new(2022, month: 12, day: 1),
  ShipDate = new(2022, month: 11, day: 5),
  Total = 49.99M
};
```

8. Измените текущую культуру на английскую (США), чтобы убедиться, что вы видите тот же результат, который показан в книге. В дальнейшем вы сможете поэкспериментировать с культурами самостоятельно.

9. Запустите приложение и убедитесь, что вывод соответствует ожидаемым ошибкам:

```
Current culture: English (United States)

CustomerName:  Abc
CustomerEmail: abc&example.com
CustomerLevel: 4
OrderId:       10001
OrderDate:     12/1/2022 12:00:00 AM
ShipDate:      11/5/2022 12:00:00 AM
Total:         $49.99

IsValid: False
  Warning: The length of 'Customer Name' must be at least 5 characters.
You entered 3 characters.
  Error: 'Customer Email' is not a valid email address.
  Error: 'Customer Level' has a range of values which does not include '4'.
  Error: 'Ship Date' must be greater than '12/1/2022 12:00:00 AM'.
  Error: 'Total' must be less than '20'.
```

10. Внесите изменения в значения некоторых свойств заказа:

```
Order order = new()
{
  OrderId = 10001,
  CustomerName = "Abcdef",
  CustomerEmail = "abc@example.com",
  CustomerLevel = CustomerLevel.Gold,
  OrderDate = new(2022, month: 12, day: 1),
  ShipDate = new(2022, month: 12, day: 5),
  // Значение параметра CustomerLevel равно Gold, следовательно
  // значение Total может быть >20
  Total = 49.99M
};
```

11. Запустите консольное приложение и обратите внимание, что заказ теперь допустим:

```
IsValid: True
```

Проверка достоверности данных в ASP.NET Core

Библиотека FluentValidation поддерживает автоматическую валидацию данных в ASP.NET Core, начиная с версии .NET Core 3.1 и выше.

В копилку знаний: более подробную информацию вы найдете по адресу https://cecilphillip.com/fluent-validation-rules-with-asp-net-core/.

Создание PDF-файлов

На курсах по C# и .NET меня очень часто спрашивают, какую библиотеку выбрать для создания PDF-файлов. В свое время я долго искал ответ на этот вопрос. Среди множества платных библиотек найти кросс-платформенное решение с открытым исходным кодом оказалось довольно сложно.

В лицензионной информации к QuestPDF указано:

> «Многопользовательскую лицензию Professional или Enterprise License необходимо приобретать только при использовании QuestPDF в коммерческом ПО или при разработке программных продуктов частными лицами или компаниями с годовым доходом свыше 1 млн долларов США. Для более подробной информации следует ознакомиться с лицензионным соглашением и ценами на сайте программы QuestPDF по адресу https://www.questpdf.com/pricing.html».

Для разработки некоммерческого ПО лицензия не требуется. Кроме того, старая версия библиотеки (2022.12.X) также доступна по лицензии MIT и может использоваться бесплатно в коммерческих проектах. Если же вы хотите поддержать разработчиков библиотеки, рекомендую приобрести профессиональную лицензию к версии 2023.1.X или более поздней.

QuestPDF на устройствах Mac с Apple Silicon

В библиотеке QuestPDF используется API SkiaSharp с реализациями для операционных систем Windows, macOS и Linux. Таким образом, консольное приложение, создаваемое в этом разделе, будет кросс-платформенным. Однако на компьютере Mac с архитектурой Apple Silicon (например, на Mac mini M1) потребуется установить версию .NET SDK для архитектуры x64 и запускать проект командой `dotnet new -a x64`. Это необходимо во избежание ошибок, связанных с отсутствием сборок библиотеки SkiaSharp для архитектуры ARM64.

Создание библиотек классов для генерации PDF-документов

Теперь рассмотрим пример использования библиотеки QuestPDF. Мы создадим три проекта.

- *Библиотека классов для модели*, которая будет представлять каталог категорий товаров с именами и изображениями.
- *Библиотека классов для шаблона документа*, который будет использоваться для генерации PDF-документа.
- *Консольное приложение*, которое будет создавать PDF-файл в реальном времени.

Приступим.

1. В редакторе кода создайте проект библиотеки классов `GeneratingPdf.Models` и добавьте его в решение `Chapter06`.
2. В папке созданного проекта удалите файл `Class1.cs`.
3. В файле проекта `GeneratingPdf.Models` создайте класс `Category` и определите в нем два свойства — имя категории и идентификатор:

```
namespace GeneratingPdf.Models;

public class Category
{
  public int CategoryId { get; set; }
  public string CategoryName { get; set; } = null!;
}
```

Позже вам нужно будет создать папку images, в которой будут находиться изображения с именами category*N*.jpeg, где *N* — это число от 1 до 8, соответствующее значению параметра CategoryId.

4. В папке проекта `GeneratingPdf.Models` создайте класс `Catalog` с восемью категориями:

```
namespace GeneratingPdf.Models;

public class Catalog
{
  public List<Category> Categories { get; set; } = null!;
}
```

5. В редакторе кода создайте проект библиотеки классов `GeneratingPdf.Document` и добавьте его в решение `Chapter06`.
6. В файле созданного проекта подключите пакет `QuestPDF` и добавьте ссылку на проект `GeneratingPdf.Models`:

```
<ItemGroup>
  <!-- Самая новая версия с лицензией MIT -->
  <PackageReference Include="QuestPDF" Version="2022.12.6" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include=
    "..\GeneratingPdf.Models\GeneratingPdf.Models.csproj" />
</ItemGroup>
```

7. Выполните сборку проекта `GeneratingPdf.Document`.
8. Удалите файл `Class1.cs` из этого проекта.
9. В этом же проекте создайте файл класса `CatalogDocument.cs`.

10. Добавьте в этот файл класс, реализующий интерфейс `IDocument`. В нем определите шаблон с верхним и нижним колонтитулами, а также вывод восьми категорий с соответствующими названиями и изображениями:

```
using GeneratingPdf.Models; // Класс Catalog
using QuestPDF.Drawing; // Библиотека DocumentMetadata
using QuestPDF.Fluent; // Класс Page
using QuestPDF.Helpers; // Класс Colors
using QuestPDF.Infrastructure; // Интерфейсы IDocument, IDocumentContainer

namespace GeneratingPdf.Document;

public class CatalogDocument : IDocument
{
  public Catalog Model { get; }

  public CatalogDocument(Catalog model)
  {
    Model = model;
  }

  public void Compose(IDocumentContainer container)
  {
    container
      .Page(page =>
      {
        page.Margin(50 /* точек */);

        page.Header()
          .Height(100).Background(Colors.Grey.Lighten1)
          .AlignCenter().Text("Catalogue")
          .Style(TextStyle.Default.FontSize(20));

        page.Content()
          .Background(Colors.Grey.Lighten3)
          .Table(table =>
          {
            table.ColumnsDefinition(columns =>
            {
              columns.ConstantColumn(100);
              columns.RelativeColumn();
            });

            foreach (var item in Model.Categories)
            {
              table.Cell().Text(item.CategoryName);

              string imagePath = Path.Combine(
                Environment.CurrentDirectory, "images",
                $"category{item.CategoryId}.jpeg");

              table.Cell().Image(imagePath);
            }
          });
```

```
            page.Footer()
              .Height(50).Background(Colors.Grey.Lighten1)
              .AlignCenter().Text(x =>
              {
                x.CurrentPageNumber();
                x.Span(" of ");
                x.TotalPages();
              });
          });
      }

      public DocumentMetadata GetMetadata() => DocumentMetadata.Default;
    }
```

Создание консольного приложения для формирования PDF-документов

Теперь создадим консольное приложение, которое с помощью вышеописанных библиотек будет генерировать PDF-документы.

1. В редакторе кода создайте консольное приложение `GeneratingPdf.Console` и поместите его в решение `Chapter06`.
2. В проекте `GeneratingPdf.Console` создайте папку `images` и загрузите в нее восемь изображений категорий с 1-го по 8-й из GitHub-репозитория по адресу https://github.com/markjprice/apps-services-net8/tree/master/images/Categories.
3. В Visual Studio 2022 или JetBrains Rider вам нужно скопировать эти изображения в папку `GeneratingPdf.Console\bin\Debug\net8`:
   1) выделите все изображения на панели Solution Explorer (Обозреватель решений);
   2) на панели Properties (Свойства) в раскрывающемся списке Copy to Output Directory (Копировать в выходной каталог) выберите пункт Copy always (Всегда копировать);
   3) откройте файл проекта и убедитесь, что в элементе `<ItemGroup>` прописаны правила копирования изображений:

```
<ItemGroup>
  <None Update="images\category1.jpeg">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
...
```

4. В проекте `GeneratingPdf.Console` добавьте строгую обработку предупреждений, статически импортируйте класс `System.Console` на глобальном уровне и добавьте ссылку на проект для библиотеки классов шаблона `Document`:

```
<ItemGroup>
  <ProjectReference Include=
    "..\GeneratingPdf.Document\GeneratingPdf.Document.csproj" />
</ItemGroup>
```

5. Выполните сборку проекта `GeneratingPdf.Console`.
6. Удалите содержимое файла `Program.cs`, а затем добавьте код для создания модели каталога, передачи ее в шаблон документа, создания PDF-документа и открытия готового файла с помощью соответствующей команды операционной системы:

```
using GeneratingPdf.Document; // Для класса CatalogDocument
using GeneratingPdf.Models; // Для классов Catalog, Category
using QuestPDF.Fluent; // Для метода расширения GeneratePdf
using QuestPDF.Infrastructure; // Для перечисления LicenseType

// В ознакомительных целях вы можете использовать инструмент QuestPDF
// с лицензией Community в тестовой среде
QuestPDF.Settings.License = LicenseType.Community;

string filename = "catalog.pdf";

Catalog model = new()
{
  Categories = new()
  {
    new() { CategoryId = 1, CategoryName = "Beverages"},
    new() { CategoryId = 2, CategoryName = "Condiments"},
    new() { CategoryId = 3, CategoryName = "Confections"},
    new() { CategoryId = 4, CategoryName = "Dairy Products"},
    new() { CategoryId = 5, CategoryName = "Grains/Cereals"},
    new() { CategoryId = 6, CategoryName = "Meat/Poultry"},
    new() { CategoryId = 7, CategoryName = "Produce"},
    new() { CategoryId = 8, CategoryName = "Seafood"}
  }
};

CatalogDocument document = new(model);
document.GeneratePdf(filename);

WriteLine("PDF catalog has been created: {0}",
  Path.Combine(Environment.CurrentDirectory, filename));

try
{
  if (OperatingSystem.IsWindows())
  {
    System.Diagnostics.Process.Start("explorer.exe", filename);
  }
  else
  {
    WriteLine("Open the file manually.");
  }
}
catch (Exception ex)
{
  WriteLine($"{ex.GetType()} says {ex.Message}");
}
```

Класс Process и его метод Start должны также работать в macOS и Linux, однако в этих операционных системах может потребоваться дополнительно настроить пути. Оставим это на десерт. Дополнительная информация о классе Process и методе Start опубликована по адресу https://learn.microsoft.com/ru-ru/dotnet/api/system.diagnostics.process.start.

7. Запустите консольное приложение и убедитесь, что созданный PDF-файл выглядит так же, как на рис. 6.4.

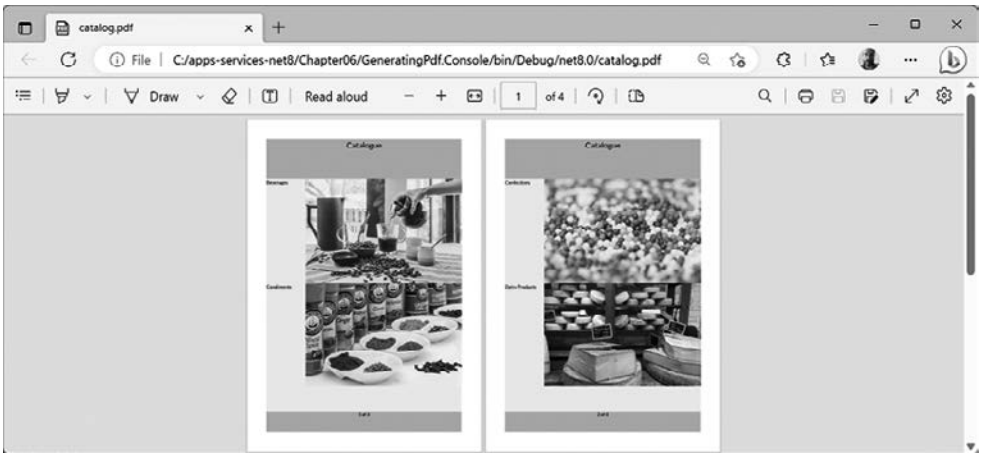


Рис. 6.4. PDF-файл, созданный C#-приложением

В копилку знаний: более подробную информацию вы найдете по адресу https://www.questpdf.com/.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 6.1. Проверьте свои знания

Ответьте на следующие вопросы (при необходимости пользуйтесь источниками информации в Интернете).

1. Какой пакет сторонних разработчиков скачивали в NuGet чаще всего (за все время)?
2. Какой метод в классе `Image` из библиотеки ImageSharp позволяет, помимо прочего, изменить размер изображения и преобразовать его в черно-белое?

3. В чем главное преимущество библиотеки Serilog в рамках журналирования?
4. Что означает термин *«приемник»* в Serilog?
5. Всегда ли следует использовать пакет AutoMapper для сопоставления объектов?
6. Какой метод библиотеки FluentAssertions следует вызвать для простого утверждения значения?
7. Какой метод библиотеки FluentAssertions проверяет, все ли элементы последовательности соответствуют условию, например тому, что строки должны содержать не более шести символов?
8. От какого класса библиотеки FluentValidation нужно наследовать пользовательский валидатор?
9. Можно ли в FluentValidation задать правило, которое будет применяться только при определенных условиях?
10. Какой интерфейс и с какими методами в QuestPDF нужно реализовать для создания PDF-документа?

Упражнение 6.2. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-6---implementing-popular-third-party-libraries.

Резюме

В этой главе вы познакомились с рядом сторонних библиотек, которые широко используются разработчиками .NET для решения различных задач.

- Обработки изображений с помощью ImageSharp — библиотеки, рекомендуемой компанией Microsoft.
- Упрощения работы с текстом, числами, датами и временем благодаря Humanizer.
- Журналирования структурированных данных с использованием Serilog.
- Сопоставления объектов, например моделей сущностей и представлений.
- Создания простых утверждений в модульных тестах.
- Проверки данных с учетом региональных особенностей.
- Формирования PDF-документов.

В следующей главе мы разберем интернационализацию дат и времени, локализацию, а также новый тип в .NET 8, который упрощает тестирование компонентов, зависящих от текущего времени.

7 Дата, время и интернационализация

В этой главе рассматриваются ключевые типы, входящие в состав .NET и предназначенные для работы с датами, временем и интернационализацией, включая глобализацию и локализацию.

При работе с временем особенно важно учитывать часовые пояса. Часто ошибки возникают из-за сравнения двух значений, представленных в разных часовых поясах, без учета этого фактора. Важно понимать концепцию *Всемирного координированного времени (Coordinated Universal Time, UTC)* и приводить все значения к стандарту UTC перед выполнением операций с датой и временем.

Следует также учитывать корректировку времени при *переходе на летнее время (Daylight Saving Time, DST)*, так как это может повлиять на точность вычислений.

Управление датами и временем

После чисел и текста даты и время являются одними из самых распространенных типов данных. В .NET для работы с ними предусмотрены два основных типа:

- `DateTime` — хранит дату и время, соответствующие определенному моменту времени;
- `TimeSpan` — представляет собой продолжительность времени.

Эти типы часто используются вместе. Например:

- разность двух значений `DateTime` дает `TimeSpan`;
- прибавление `TimeSpan` к `DateTime` приводит к новому значению `DateTime`.

Создание значений даты и времени

Один из распространенных способов создания значений `DateTime` — передача отдельных параметров, таких как день и час (табл. 7.1).

Таблица 7.1. Параметры для форматирования значений даты и времени

Параметр даты/времени	Диапазон значений
`year` (год)	От 1 до 9999
`month` (месяц)	От 1 до 12
`day` (день)	От 1 до количества дней в текущем месяце
`hour` (час)	От 0 до 23
`minute` (минута)	От 0 до 59
`second` (секунда)	От 0 до 59
`millisecond` (миллисекунда)	От 0 до 999
`microsecond` (микросекунда)	От 0 до 999

Пример генерации значения `DateTime`, соответствующего дате выхода .NET 9 в рамках релиза *General Availability (GA)*:

```
DateTime dotnet9GA = new(year: 2024, month: 11, day: 12, hour: 11, minute: 0,
second: 0);
```

Дельный совет: важно понимать, в каком часовом поясе представлено значение DateTime. Это одна из основных проблем при использовании этого типа. Вместо него рекомендуется использовать тип DateTimeOffset, который дополнительно хранит информацию о часовом поясе. Подробнее этот вопрос рассмотрен далее в этой главе.

Другой способ задания даты — передача строки с датой и выполнением синтаксического анализа. Однако такой метод может привести к некорректному результату из-за различий в форматах даты. Например, в Великобритании используется формат *«день/месяц/год»*, а в США — *«месяц/день/год»*.

Давайте поэкспериментируем с датами и временем.

1. В редакторе кода создайте консольное приложение со следующими настройками:
   - выберите шаблон проекта Console App (Консольное приложение);
   - в поле Project name (Имя проекта) укажите значение `WorkingWithTime`;
   - в поле Solution name (Имя решения) укажите значение `Chapter07`;
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
   - снимите флажок Enable native AOT publishing (Включить публикацию Native AOT).

2. В файле проекта добавьте строгую обработку предупреждений и статически импортируйте класс `System.Console` на глобальном уровне.
3. Создайте файл класса `Program.Helpers.cs` и напишите в нем следующий код:

```
using System.Globalization; // Для класса CultureInfo

partial class Program
{
  private static void ConfigureConsole(string culture = "en-US",
    bool overrideComputerCulture = true)
  {
    // Для добавления специальных символов, например евро
    OutputEncoding = System.Text.Encoding.UTF8;

    Thread t = Thread.CurrentThread;

    if (overrideComputerCulture)
    {
      t.CurrentCulture = CultureInfo.GetCultureInfo(culture);
      t.CurrentUICulture = t.CurrentCulture;
    }

    CultureInfo ci = t.CurrentCulture;
    WriteLine($"Current culture: {ci.DisplayName}");
    WriteLine($"Short date pattern: {
      ci.DateTimeFormat.ShortDatePattern}");
    WriteLine($"Long date pattern: {
      ci.DateTimeFormat.LongDatePattern}");
    WriteLine();
  }

  private static void SectionTitle(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkYellow;
    WriteLine($"*** {title}");
    ForegroundColor = previousColor;
  }
}
```

4. Удалите содержимое файла `Program.cs` и добавьте код для инициализации специальных значений даты/времени:

```
ConfigureConsole(); // По умолчанию установлена культура en-US

SectionTitle("Specifying date and time values");

WriteLine($"DateTime.MinValue:  {DateTime.MinValue}");
WriteLine($"DateTime.MaxValue:  {DateTime.MaxValue}");
WriteLine($"DateTime.UnixEpoch: {DateTime.UnixEpoch}");
```

```
WriteLine($"DateTime.Now:       {DateTime.Now}");
WriteLine($"DateTime.Today:     {DateTime.Today}");
WriteLine($"DateTime.Today:     {DateTime.Today:d}");
WriteLine($"DateTime.Today:     {DateTime.Today:D}");
```

5. Выполните код и обратите внимание на результаты:

```
Current culture: English (United States)
Short date pattern: M/d/yyyy
Long date pattern: dddd, MMMM d, yyyy

*** Specifying date and time values
DateTime.MinValue:  1/1/0001 12:00:00 AM
DateTime.MaxValue:  12/31/9999 11:59:59 PM
DateTime.UnixEpoch: 1/1/1970 12:00:00 AM
DateTime.Now:       5/30/2023 9:18:05 AM
DateTime.Today:     5/30/2023 12:00:00 AM
DateTime.Today:     5/30/2023
DateTime.Today:     Tuesday, May 30, 2023
```

Вывод значений даты и времени в консольном приложении зависит от настроек культуры. Чтобы убедиться, что форматирование соответствует ожидаемому стандарту, вызывается метод ConfigureConsole, который задает культуру по умолчанию.

6. В начале файла `Program.cs`, где вызывается `ConfigureConsole`, добавьте оператор для запрета переопределения локальных настроек культуры в вашей системе:

```
ConfigureConsole(overrideComputerCulture: false);
```

7. Выполните код и посмотрите, как форматируется дата в зависимости от текущей культуры компьютера.
8. В файле `Program.cs` задайте альтернативную культуру, например:
   - французский (Канада) — `fr-CA`;
   - английский (Великобритания) — `en-GB`.

```
ConfigureConsole("fr-CA");
```

В копилку знаний: список кодов распространенных культур опубликован на сайте по адресу https://en.wikipedia.org/wiki/Language_localisation#Language_tags_and_codes.

9. Выполните код и обратите внимание, что вывод локализован в соответствии с указанной культурой.
10. Сбросьте конфигурацию консоли на значения по умолчанию, чтобы в ней использовалась культура `en-US`:

```
ConfigureConsole(); // По умолчанию установлена культура en-US
```

Форматирование значений даты и времени

Даты и время по умолчанию форматируются в зависимости от текущей культуры. Однако можно задать формат вручную, используя коды пользовательских форматов (табл. 7.2).

Таблица 7.2. Коды пользовательских форматов для значений даты и времени

Код	Описание
`/`	Разделитель частей даты. Зависит от культуры: например, в `en-US` используется символ `/`, а в `fr-FR` применяется `-` (дефис)
`\`	Escape-символ. Необходим для предотвращения интерпретации специального символа в качестве описателя формата; например, шаблон `h \h m \m` отформатирует время 9:30 как `9 h 30 m`
`:`	Разделитель частей времени. Зависит от культуры: например, в `en-US` используется `:`, а в `fr-FR` — `.` (точка)
`d`	День месяца, от `1` до `31`
`dd`	День месяца с нулем в начале, от `01` до `31`
`ddd`	Сокращенное название дня недели, например `Mon`, локализованное для текущей культуры
`dddd`	Полное название дня недели, например `Monday`, локализованное для текущей культуры
`f`	Десятые доли секунды
`ff`	Сотые доли секунды
`fff`	Тысячные доли секунды (миллисекунды)
`g`	Период или эра, например `A.D.`
`h`	Час в 12-часовом формате от `1` до `12`
`hh`	Час в 12-часовом формате от `01` до `12`
`H`	Час в 24-часовом формате от `0` до `23`
`HH`	Час в 24-часовом формате от `00` до `23`

Код	Описание
K	Данные о часовом поясе. null — неопределенный часовой пояс, Z — дата в формате UTC, значение вида -8:00 — местное время с часовым сдвигом от времени в формате UTC (универсального времени)
m	Минута, от 0 до 59
mm	Минута с нулем в начале, от 00 до 59
M	Месяц, от 1 до 12
MM	Месяц с нулем в начале, от 01 до 12
MMM	Сокращенное название месяца, например Jan, локализованное для текущей культуры
MMMM	Полное название месяца, например January, локализованное для текущей культуры
s	Секунда, от 0 до 59
ss	Секунда с нулем в начале, от 00 до 59
t	Первый символ указателя AM/PM (до полудня/после полудня)
tt	Указатель AM/PM (до полудня/после полудня)
y	Год текущего столетия, от 0 до 99
yy	Год текущего столетия с нулем в начале, от 00 до 99
yyy	Год в виде как минимум трех цифр. Если в году больше трех значащих цифр, то они включаются в результат. Например, 1 год н. э. — это 001. Первое разграбление Рима произошло в 410 году. Год издания этой книги — 2023
yyyy	Год в виде четырехзначного числа
yyyyy	Год в виде пятизначного числа
z	Часовой сдвиг от времени в формате UTC (универсального времени) без нулей в начале
zz	Часовой сдвиг от времени в формате UTC (универсального времени) с нулями в начале для значений из одной цифры
zzz	Сдвиг в часах и минутах от времени в формате UTC (универсального времени), например +04:30

В копилку знаний: полный список кодов пользовательских форматов доступен по адресу https://learn.microsoft.com/ru-ru/dotnet/standard/base-types/custom-date-and-time-format-strings.

Вместо кодов пользовательских форматов можно применять коды стандартных форматов, например d и D, которые мы использовали в примере (табл. 7.3).

Таблица 7.3. Коды стандартных форматов для значений даты и времени

Код формата	Описание
d	Короткий шаблон даты. Зависит от культуры: например, в en-US используется формат M/d/yyyy, а в fr-FR — dd/MM/yyyy
D	Длинный шаблон даты. Зависит от культуры: например, в en-US используется формат mmmm, MMMM d, yyyy. а для fr-FR — mmmm, dd MMMM yyyy
f	Полный шаблон даты и времени (короткий шаблон времени — часы и минуты). Зависит от культуры
F	Полный шаблон даты и времени (полный шаблон времени — часы, минуты, секунды и AM/PM). Зависит от культуры
o O	Шаблон обратного преобразования даты и времени, стандартизированный формат ISO 8601, например 2023-05-30T13:45:30.0000000-08:00
r R	Шаблон RFC1123
t	Короткий шаблон времени. Зависит от культуры: например, в en-US используется формат h:mm tt, а в fr-FR — HH:mm
T	Длинный шаблон времени. Зависит от культуры: например, в en-US используется формат h:mm:ss tt, а в fr-FR — HH:mm:ss
u	Универсальный сортируемый шаблон времени и даты, например 2009-06-15 13:45:30Z
U	Универсальный полный шаблон даты и времени. Зависит от культуры: например, для en-US это может иметь вид Monday, June 15, 2009 8:45:30 PM

Рассмотрим несколько примеров.

1. В файле Program.cs добавьте код, который определяет дату католического Рождества 2024 года и выводит ее разными способами:

```
DateTime xmas = new(year: 2024, month: 12, day: 25);
WriteLine($"Christmas (default format): {xmas}");
WriteLine($"Christmas (custom short format): {xmas:ddd d/M/yy}");
WriteLine($"Christmas (custom long format): {
  xmas:dddd, dd MMMM yyyy}");
WriteLine($"Christmas (standard long format): {xmas:D}");
WriteLine($"Christmas (sortable): {xmas:u}");
WriteLine($"Christmas is in month {xmas.Month} of the year.");
WriteLine($"Christmas is day {xmas.DayOfYear} of {xmas.Year}.");
WriteLine($"Christmas {xmas.Year} is on a {xmas.DayOfWeek}.");
```

2. Выполните код и обратите внимание на результаты:

```
Christmas (default format): 12/25/2024 12:00:00 AM
Christmas (custom short format): Wed, 25/12/24
```

```
Christmas (custom long format): Wednesday, 25 December 2024
Christmas (standard long format): Wednesday, December 25, 2024
Christmas (sortable): 2024-12-25 00:00:00Z
Christmas is in month 12 of the year.
Christmas is day 360 of 2024.
Christmas 2024 is on a Wednesday.
```

3. Отмените переопределение культурного контекста или задайте конкретную культуру, например `fr-FR` (Франция):

```
ConfigureConsole("fr-FR"); // По умолчанию используется культура en-US
```

4. Выполните код и посмотрите, как изменится вывод. Результаты должны быть локализованы для заданной культуры.
5. Сбросьте конфигурацию консоли на значения по умолчанию, чтобы в ней использовалась культура `en-US`.

Вычисления даты и времени

Теперь поэкспериментируем с простыми вычислениями дат и времени.

1. В файл `Program.cs` добавьте операторы сложения и вычитания вместе с датой католического Рождества в 2024 году:

```
SectionTitle("Date and time calculations");

DateTime beforeXmas = xmas.Subtract(TimeSpan.FromDays(12));
DateTime afterXmas = xmas.AddDays(12);

WriteLine($"12 days before Christmas: {beforeXmas:d}");
WriteLine($"12 days after Christmas: {afterXmas:d}");

TimeSpan untilXmas = xmas - DateTime.Now;

WriteLine($"Now: {DateTime.Now}");
WriteLine($"There are {untilXmas.Days} days and {untilXmas.Hours
  } hours until Christmas {xmas.Year.");

WriteLine("There are {untilXmas.TotalHours:N0} hours " +
  $"until Christmas {xmas.Year}.");
```

2. Выполните код и обратите внимание на результаты:

```
*** Date and time calculations
12 days before Christmas: 12/13/2024
12 days after Christmas: 1/6/2025
Now: 5/30/2023 1:57:01 PM
There are 574 days and 10 hours until Christmas 2024.
There are 13,786 hours until Christmas 2024.
```

3. Определите, во сколько ваши дети (или питомцы) могут проснуться в этот день и открыть подарки, и выведите время разными способами:

```
DateTime kidsWakeUp = new(
  year: 2024, month: 12, day: 25,
  hour: 6, minute: 30, second: 0);

WriteLine($"Kids wake up: {kidsWakeUp}");

WriteLine($"The kids woke me up at {
  kidsWakeUp.ToShortTimeString()}");
```

4. Выполните код и обратите внимание на результаты:

```
Kids wake up: 25/12/2024 06:30:00 AM
The kids woke me up at 06:30 AM
```

Микросекунды и наносекунды

Ранее минимальной единицей измерения времени в .NET был тик (tick), равный 100 наносекундам. Разработчикам приходилось самостоятельно вычислять высокоточные значения в наносекундах. В версии .NET 7 конструкторы обзавелись поддержкой параметров милли- и микросекунд, а также были добавлены соответствующие свойства в типы `DateTime`, `DateTimeOffset`, `TimeSpan` и `TimeOnly`.

Рассмотрим пару примеров.

1. В файле `Program.cs` добавьте код для вычисления высокоточных значений даты и времени и выведите результат:

```
SectionTitle("Milli-, micro-, and nanoseconds");

DateTime preciseTime = new(
  year: 2022, month: 11, day: 8,
  hour: 12, minute: 0, second: 0,
  millisecond: 6, microsecond: 999);

WriteLine($"Millisecond: {preciseTime.Millisecond}, Microsecond: {
  preciseTime.Microsecond}, Nanosecond: {preciseTime.Nanosecond}");
preciseTime = DateTime.UtcNow;

// Значение наносекунды будет от 0 до 900 с шагом 100 наносекунд
WriteLine($"Millisecond: {preciseTime.Millisecond}, Microsecond: {
  preciseTime.Microsecond}, Nanosecond: {preciseTime.Nanosecond}");
```

2. Выполните код и обратите внимание на результаты:

```
*** Milli-, micro- and nanoseconds
Millisecond: 6, Microsecond: 999, Nanosecond: 0
Millisecond: 243, Microsecond: 958, Nanosecond: 400
```

Глобализация дат и времени

Форматирование и синтаксический анализ дат зависят от текущей культуры системы.

1. В начале файла `Program.cs` импортируйте пространство имен для поддержки глобализации:

```
using System.Globalization; // Для класса CultureInfo
```

2. Выведите текущую культуру, согласно которой форматируются даты и время. Затем выполните синтаксический анализ даты Дня независимости США и выведите ее в разных форматах:

```
SectionTitle("Globalization with dates and times");
// То же самое, что свойство Thread.CurrentThread.CurrentCulture
WriteLine($"Current culture: {CultureInfo.CurrentCulture.Name}");

string textDate = "4 July 2024";
DateTime independenceDay = DateTime.Parse(textDate);

WriteLine($"Text: {textDate}, DateTime: {independenceDay:d MMMM}");

textDate = "7/4/2024";
independenceDay = DateTime.Parse(textDate);

WriteLine($"Text: {textDate}, DateTime: {independenceDay:d MMMM}");

// Переопределение текущей культуры путем установки провайдера
independenceDay = DateTime.Parse(textDate,
  provider: CultureInfo.GetCultureInfo("en-US"));

WriteLine($"Text: {textDate}, DateTime: {independenceDay:d MMMM}");
```

Дельный совет: если вам не нужно изменять объект CultureInfo, используйте метод GetCultureInfo, который возвращает общий экземпляр, доступный только для чтения.

3. В начале файла `Program.cs` задайте культуру `en-GB`:

```
ConfigureConsole("en-GB");
```

4. Выполните код и обратите внимание на результаты:

```
*** Globalization with dates and times
Current culture is: en-GB
Text: 4 July 2024, DateTime: 4 July
Text: 7/4/2024, DateTime: 7 April
Text: 7/4/2024, DateTime: 4 July
```

Если используется английская (Великобритания) культура, дата 4 июля 2024 года будет распознана правильно вне зависимости от формата (британского или американского). Однако запись 7/4/2024 будет интерпретирована как 7 апреля, а не 4 июля. Чтобы избежать ошибок, явно указывайте нужную культуру при анализе даты.

5. Напишите цикл, перебирающий годы с 2023-го по 2028-й, который:
   - определяет, является ли год високосным;
   - выводит количество дней в феврале;
   - проверяет, попадают ли Рождество и День независимости США на период летнего времени (DST):

```
for (int year = 2023; year <= 2028; year++)
{
  Write($"{year} is a leap year: {DateTime.IsLeapYear(year)}. ");
  WriteLine($"There are {DateTime.DaysInMonth(year: year, month: 2)
    } days in February {year}.");
}

WriteLine($"Is Christmas daylight saving time? {
  xmas.IsDaylightSavingTime()}");

WriteLine($"Is July 4th daylight saving time? {
  independenceDay.IsDaylightSavingTime()}");
```

6. Выполните код и обратите внимание на результаты:

```
2023 is a leap year: False. There are 28 days in February 2023.
2024 is a leap year: True. There are 29 days in February 2024.
2025 is a leap year: False. There are 28 days in February 2025.
2026 is a leap year: False. There are 28 days in February 2026.
2027 is a leap year: False. There are 28 days in February 2027.
2028 is a leap year: True. There are 29 days in February 2028.
Is Christmas daylight saving time? False
Is July 4th daylight saving time? True
```

Сложности перехода на летнее время (DST)

Летнее время (DST) используется не во всех странах, а его переход зависит от полушария и даже политических решений. Например, в США обсуждается вопрос о том, стоит ли переводить часы во всей стране одновременно или оставить это решение на усмотрение губернаторов отдельных штатов. В ближайшие годы американцам, возможно, придется разбираться с разными правилами перехода на летнее время.

В каждой стране соблюдаются собственные правила перехода между зимним и летним временем. В .NET эти правила закодированы, позволяя системе при необходимости автоматически корректировать время.

В США перевод часов на час вперед происходит весной в 02:00 ночи, а осенью — назад в 02:00 ночи. Подробнее можно узнать на вики-сайте https://en.wikipedia.org/wiki/Daylight_saving_time_in_the_United_States.

В Великобритании весенний переход происходит в 01:00 ночи, а осенью — в 02:00 ночи. Официальная информация доступна на сайте правительства Великобритании по адресу https://www.gov.uk/when-do-the-clocks-change.

Представьте, что вам нужно установить будильник на 01:30, чтобы успеть на самолет, который вылетает из аэропорта Хитроу в день смены времени.

- Весной в этот день часы перескакивают с 00:59 сразу на 02:00 и время 01:30 не наступает. Будильник не сработает, а если попытаться задать такое время в .NET, будет выброшено исключение.
- Осенью стрелки переводят в 01:59 на час назад и время 01:30 наступает дважды.

Локализация перечисления DayOfWeek

Перечисление `DayOfWeek` жестко закодировано на английском языке и не может быть локализовано автоматически:

```
namespace System
{
  public enum DayOfWeek
  {
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6
  }
}
```

Есть два способа обойти это ограничение.

Во-первых, можно использовать формат даты `dddd`, позволяющий вывести название дня недели в соответствии с текущей культурой:

```
WriteLine($"The day of the week is {DateTime.Now:dddd}.");
```

Во-вторых, вы можете воспользоваться вспомогательным методом класса `DateTimeFormatInfo`, который преобразует `DayOfWeek` в локализованный текст.

Попробуем применить это на практике.

1. В файле `Program.cs` явно задайте датскую культуру (`da-DK`) и выведите текущий день недели в соответствующем формате:

```
SectionTitle("Localizing the DayOfWeek enum");

CultureInfo previousCulture = Thread.CurrentThread.CurrentCulture;

// Явно установите культуру на датскую (Дания)
Thread.CurrentThread.CurrentCulture =
  CultureInfo.GetCultureInfo("da-DK");

// Перечисление DayOfWeek не локализовано для Дании
WriteLine("Culture: {Thread.CurrentThread.CurrentCulture
  .NativeName}, DayOfWeek: {DateTime.Now.DayOfWeek}");

// Применение кода формата dddd для вывода локализованного дня недели
WriteLine($"Culture: {Thread.CurrentThread.CurrentCulture
  .NativeName}, DayOfWeek: {DateTime.Now:dddd}");

// Применение метода GetDayName для вывода локализованного дня недели
WriteLine("Culture: {Thread.CurrentThread.CurrentCulture
  .NativeName}, DayOfWeek: {DateTimeFormatInfo.CurrentInfo
  .GetDayName(DateTime.Now.DayOfWeek)}");

Thread.CurrentThread.CurrentCulture = previousCulture;
```

2. Выполните код и обратите внимание на результаты:

```
*** Localizing the DayOfWeek enum
Culture: dansk (Danmark), DayOfWeek: Thursday
Culture: dansk (Danmark), DayOfWeek: torsdag
Culture: dansk (Danmark), DayOfWeek: torsdag
```

Работа с датами и временем по отдельности

В версии .NET 6 были добавлены типы `DateOnly` и `TimeOnly` для работы с датами и временем по отдельности.

- `DateOnly` удобен для хранения дат без учета времени, что исключает ошибки при обработке данных. Он хорошо сопоставляется с типами БД, например со столбцом `DATE` в SQL Server.
- `TimeOnly` идеально подходит для работы с расписаниями, будильниками и часами работы. В БД SQL Server он соответствует типу `TIME`.

Попробуем применить их на практике в рамках планирования вечеринки по случаю выхода .NET 9. Это событие произошло во вторник, 12 ноября 2024 года, через неделю после президентских выборов в США.

1. В файле `Program.cs` добавьте код для определения даты вечеринки по случаю выхода .NET 9. Определите время начала вечеринки и объедините дату с временем, чтобы добавить событие в календарь:

```
SectionTitle("Working with only a date or a time");

DateOnly party = new(year: 2024, month: 11, day: 12);
WriteLine($"The .NET 9 release party is on {party.ToLongDateString()}.");

TimeOnly starts = new(hour: 11, minute: 30);
WriteLine($"The party starts at {starts}.");

DateTime calendarEntry = party.ToDateTime(starts);
WriteLine($"Add to your calendar: {calendarEntry}.");
```

2. Выполните код и запишите результаты:

```
*** Working with only a date or a time
The .NET 9 release party is on Tuesday, November 12, 2024.
The party starts at 11:30 AM.
Add to your calendar: 11/12/2024 11:30:00 AM.
```

Получение информации о форматировании даты/времени

У каждой культуры свои правила форматирования даты и времени. В .NET они определяются свойством `DateTimeFormat` объекта `CultureInfo`.

Рассмотрим этот аспект.

1. В файле `Program.cs` добавьте код для извлечения сведений о формате даты/времени для текущей культуры и выведите основные параметры:

```
SectionTitle("Working with date/time formats");

DateTimeFormatInfo dtfi = DateTimeFormatInfo.CurrentInfo;
// Или используйте выражение Thread.CurrentThread.CurrentCulture.DateTimeFormat

WriteLine($"Date separator: {dtfi.DateSeparator}");
WriteLine($"Time separator: {dtfi.TimeSeparator}");

WriteLine($"Long date pattern: {dtfi.LongDatePattern}");
WriteLine($"Short date pattern: {dtfi.ShortDatePattern}");

WriteLine($"Long time pattern: {dtfi.LongTimePattern}");
WriteLine($"Short time pattern: {dtfi.ShortTimePattern}");
```

```
Write("Day names:");
for (int i = 0; i < dtfi.DayNames.Length - 1; i++)
{
  Write($" {dtfi.GetDayName((DayOfWeek)i)}");
}
WriteLine();

Write("Month names:");
for (int i = 1; i < dtfi.MonthNames.Length; i++)
{
  Write($" {dtfi.GetMonthName(i)}");
}
WriteLine();
```

2. Выполните код и обратите внимание на результаты:

```
*** Working with date/time formats
Date separator: /
Time separator: :
Long date pattern: dddd, MMMM d, yyyy
Short date pattern: M/d/yyyy
Long time pattern: h:mm:ss tt
Short time pattern: h:mm tt
Day names:  Sunday  Monday  Tuesday  Wednesday  Thursday  Friday
Month names:  January  February  March  April  May  June  July  August
September   October  November  December
```

3. Смените культуру, вновь выполните код и сравните результаты.

Модульное тестирование с учетом времени

Тестирование компонентов, которым требуется текущее время, может быть непростой задачей, так как время постоянно меняется.

Предположим, что в интернет-магазине покупатели получают скидку 20 % на заказы, оформленные в выходные дни, а в будни платят полную стоимость. Как проверить корректность такой логики в модульных тестах?

В .NET 8 появился класс `TimeProvider`, позволяющий контролировать время в тестах. Рассмотрим процесс его использования.

1. В решении `Chapter07` создайте проект библиотеки классов `TimeFunctionsLib`.
2. В созданном проекте переименуйте файл `Class1.cs` в `DiscountService.cs`.
3. В файле `DiscountService.cs` реализуйте метод для расчета скидки:

```
namespace Northwind.Services;

public class DiscountService
{
  public decimal GetDiscount()
```

```
  {
    // Это значение зависит от текущего времени,
    // предоставляемого системой
    var now = DateTime.UtcNow;

    return now.DayOfWeek switch
    {
      DayOfWeek.Saturday or DayOfWeek.Sunday => 0.2M,
      _ => 0M
    };
  }
}
```

4. В решении `Chapter07` создайте проект xUnit Test Project (Тестовый проект xUnit) и присвойте ему имя `TestingWithTimeProvider`.
5. В файле созданного проекта добавьте ссылку на проект `TimeFunctionsLib`:

```
<ItemGroup>
  <ProjectReference Include="..\TimeFunctionsLib\TimeFunctionsLib.csproj" />
</ItemGroup>
```

6. Выполните сборку проекта `TestingWithTimeProvider`.
7. В папке проекта `TestingWithTimeProvider` переименуйте файл `Test1.cs` в `TimeTests.cs`.
8. В переименованном файле `TimeTests.cs` добавьте код для импортирования пространства имен `TestingWithTimeProvider` и определите два теста:
   - один для проверки расчета скидки в рабочие дни;
   - второй — для выходных:

```
using Northwind.Services; // Для класса DiscountService

namespace TestingWithTimeProvider;

public class TimeTests
{
  [Fact]
  public void TestDiscountDuringWorkdays()
  {
    // Подготовка
    DiscountService service = new();

    // Действие
    decimal discount = service.GetDiscount();

    // Проверка
    Assert.Equal(0M, discount);
  }

  [Fact]
  public void TestDiscountDuringWeekends()
```

```
        {
          DiscountService service = new();

          decimal discount = service.GetDiscount();

          Assert.Equal(0.2M, discount);
        }
      }
```

9. Запустите тесты и обратите внимание, что в зависимости от текущего дня недели один из них будет завершаться провалом:
   - если тесты выполняются в будни, будет провалена проверка скидки на выходных;
   - если тесты выполняются в выходные, то, наоборот, будет провалена проверка скидки в будни.

Итак, проблема понятна. Как ее решить?

Чтобы устранить зависимость от текущей даты, в Microsoft используют интерфейс `ISystemClock`, определяющий свойство `UtcNow`, а иногда и другие члены класса, а также реализации, обычно учитывающие встроенные системные часы. Однако каждая команда внутри .NET реализует его по-своему. Типичный пример показан ниже:

```
using System;

namespace Microsoft.Extensions.Internal
{
  public interface ISystemClock
  {
    DateTimeOffset UtcNow { get; }
  }

  public class SystemClock
  {
    public DateTimeOffset UtcNow
    {
      return DateTimeOffset.UtcNow;
    }
  }
}
```

В .NET 8 появился более универсальный метод — абстрактный класс `TimeProvider`, который заменяет `ISystemClock` и предоставляет стандартную реализацию системных часов.

Реализуем этот подход в нашем коде.

1. В файле `DiscountService.cs` проекта `TimeFunctionsLib` закомментируйте код, относящийся к свойству `UtcNow`, и замените его внедряемым через конструктор экземпляром `TimeProvider`:

```
namespace Northwind.Services;

public class DiscountService
{
  private TimeProvider _timeProvider;

  public DiscountService(TimeProvider timeProvider)
  {
    _timeProvider = timeProvider;
  }

  public decimal GetDiscount()
  {
    // Это значение зависит от текущего времени,
    // предоставляемого системой
    // var now = DateTime.UtcNow;

    var now = _timeProvider.GetUtcNow();

    // Это значение зависит от текущего времени,
    // предоставляемого системой
    return now.DayOfWeek switch
    {
      DayOfWeek.Saturday or DayOfWeek.Sunday => 0.2M,
      _ => 0M
    };
  }
}
```

2. В файле `TimeTests.cs` проекта `TestingWithTimeProvider` измените оба теста, добавив свойство `TimeProvider.System`. Оно по-прежнему зависит от системных часов! См. код ниже:

```
// Для этого используется механизм внедрения зависимостей .NET 8
// или более поздней версии, но ее реализация
// по-прежнему основана на системных часах

DiscountService service = new(TimeProvider.System);
```

3. В файле проекта `TestingWithTimeProvider` добавьте ссылку на библиотеку `Moq` для имитации зависимостей:

```
<!-- Последняя версия до скандала -->
<PackageReference Include="Moq" Version="4.18.4" />
```

В середине 2023 года вокруг Moq разразился скандал, связанный с добавлением в пакет закрытого (неявного) кода, который выполнялся во время сборки. Последняя версия без этого спорного обновления — Moq 4.18.4. Подробнее об этом можно прочитать по адресу https://github.com/devlooped/moq/issues/1370. Следите за обновлениями, прежде чем использовать этот пакет в будущем.

4. В файле `TimeTests.cs` импортируйте пространство имен для поддержки метода расширения `Mock.Of<T>`:

```
using Moq; // Для метода Mock.Of<T>
```

5. В методе `TestDiscountDuringWorkdays` закомментируйте код, касающийся свойства `TimeProvider.System`, и замените его фиктивным провайдером, который всегда возвращает дату/время в рабочие дни:

```
TimeProvider timeProvider = Mock.Of<TimeProvider>();

// Имитация провайдера времени, чтобы он всегда возвращал дату
// 2023-11-07 09:30:00 UTC - вторник
Mock.Get(timeProvider).Setup(s => s.GetUtcNow()).Returns(
  new DateTimeOffset(year: 2023, month: 11, day: 7,
  hour: 9, minute: 30, second: 0, offset: TimeSpan.Zero));

DiscountService service = new(timeProvider);
```

6. Аналогично в методе `TestDiscountDuringWeekends` подмените провайдер, чтобы он всегда возвращал дату/время в выходные дни:

```
TimeProvider timeProvider = Mock.Of<TimeProvider>();

// Имитация провайдера времени, чтобы он всегда возвращал дату
// 2023-11-04 09:30:00 UTC - суббота
Mock.Get(timeProvider).Setup(s => s.GetUtcNow()).Returns(
  new DateTimeOffset(year: 2023, month: 11, day: 4,
  hour: 9, minute: 30, second: 0, offset: TimeSpan.Zero));

DiscountService service = new(timeProvider);
```

7. Запустите модульные тесты. Теперь оба теста должны успешно выполняться независимо от текущей даты.

Таким образом, `TimeProvider` позволяет гибко управлять временем в тестах, устраняя зависимость от реальных часов и обеспечивая надежную проверку логики кода.

Учет часовых поясов .NET

В примере с вечеринкой, посвященной выпуску .NET, выбор типа `TimeOnly` оказался неудачным, поскольку не учитывались часовые пояса. Поэтому он пригодится только в приложениях, работающих в одном часовом поясе, и не подойдет для событий, требующих учета разницы во времени.

Классы DateTime и TimeZoneInfo

В .NET для работы с датами и временем применяются классы `DateTime` и `TimeZoneInfo`, обладающие рядом полезных свойств и методов.

Класс `DateTime` предоставляет возможности для работы с текущим временем, учетом часовых поясов и преобразования значений между ними (табл. 7.4).

Таблица 7.4. Члены класса DateTime, связанные с часовыми поясами

Член	Описание
Свойство `Now`	Возвращает объект `DateTime`, которому присвоены текущие дата и время данного компьютера, выраженные как местное время
Свойство `UtcNow`	Возвращает объект `DateTime`, которому присвоены текущие дата и время данного компьютера, выраженные в формате UTC
Свойство `Kind`	Возвращает значение, указывающее, на каком времени основано время, представленное этим экземпляром: местном, UTC или ни на том ни на другом
Метод `IsDaylightSavingTime`	Указывает (логическое значение), попадает ли данный экземпляр объекта `DateTime` в диапазон летнего времени для текущего часового пояса
Метод `ToLocalTime`	Преобразует значение текущего объекта `DateTime` в местное время
Метод `ToUniversalTime`	Преобразует значение текущего объекта `DateTime` во время UTC

Для более гибкого управления часовыми поясами в .NET используется класс `TimeZoneInfo`, позволяющий конвертировать время между зонами, получать список доступных зон и учитывать переходы на летнее время (табл. 7.5).

Таблица 7.5. Полезные члены класса TimeZoneInfo

Член	Описание
Свойство `Id`	Возвращает строковый идентификатор часового пояса
Свойство `Local`	Возвращает объект `TimeZoneInfo`, представляющий местный часовой пояс. Зависит от локации выполнения кода
Свойство `Utc`	Возвращает объект `TimeZoneInfo`, представляющий часовой пояс UTC

Продолжение ⇨

Таблица 7.5 *(продолжение)*

Член	Описание
Свойство `StandardName`	Возвращает строковое имя для зимнего времени часового пояса
Свойство `DaylightName`	Возвращает строковое имя для летнего времени текущего часового пояса
Свойство `DisplayName`	Возвращает универсальное строковое имя, представляющее часовой пояс
Свойство `BaseUtcOffset`	Возвращает разницу между зимним временем в текущем часовом поясе и временем в формате UTC
Свойство `SupportsDaylightSavingTime`	Возвращает логическое значение, позволяющее определить, заданы ли для часового пояса какие-либо правила перехода на летнее время
Метод `ConvertTime`	Преобразует время во время в заданном часовом поясе. Можно указать исходный и целевой часовые пояса
Метод `ConvertTimeFromUtc`	Преобразует время в формате UTC во время в указанном часовом поясе
Метод `ConvertTimeToUtc`	Преобразует время в указанном часовом поясе в формат UTC
Метод `IsDaylightSavingTime`	Указывает, попадают ли заданные дата и время в диапазон летнего времени для часового пояса текущего объекта `TimeZoneInfo`
Метод `GetSystemTimeZones`	Возвращает отсортированную коллекцию всех часовых поясов, сведения о которых доступны в локальной системе

Некоторые провайдеры БД в EF Core допускают только значения DateTime в UTC (Kind == Utc). Чтобы корректно работать с такими значениями, их часто преобразуют в DateTimeOffset и наоборот.

Пример работы с классами DateTime и TimeZoneInfo

Создадим консольное приложение для работы с часовыми поясами и классом `TimeZoneInfo`.

1. В редакторе кода создайте консольное приложение `WorkingWithTimeZones` и добавьте его в решение `Chapter07`.
   - В программе Visual Studio 2022 присвойте параметру Startup Project (Запускаемый проект) значение Current selection (Текущий вариант).

- Добавьте строгую обработку предупреждений, а также статически импортируйте класс `System.Console` на глобальном уровне.

2. Создайте файл класса `Program.Helpers.cs`.
3. Удалите его содержимое и напишите вспомогательные методы для:
   - форматирования заголовка раздела;
   - вывода списка часовых поясов в системе;
   - отображения подробной информации о `DateTime` и `TimeZoneInfo`:

```
using System.Collections.ObjectModel; // Для класса ReadOnlyCollection<T>

partial class Program
{
  private static void SectionTitle(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkYellow;
    WriteLine($"*** {title}");
    ForegroundColor = previousColor;
  }

  private static void OutputTimeZones()
  {
    // Извлечение часовых поясов, зарегистрированных в ОС
    ReadOnlyCollection<TimeZoneInfo> zones =
      TimeZoneInfo.GetSystemTimeZones();

    WriteLine($"*** {zones.Count} time zones:");

    // Упорядочение часовых поясов по свойству Id, а не по DisplayName
    foreach (TimeZoneInfo zone in zones.OrderBy(z => z.Id))
    {
      WriteLine($"{zone.Id}");
    }
  }

  private static void OutputDateTime(DateTime dateTime, string title)
  {
    SectionTitle(title);
    WriteLine($"Value: {dateTime}");
    WriteLine($"Kind: {dateTime.Kind}");
    WriteLine($"IsDaylightSavingTime: {dateTime.IsDaylightSavingTime()}");
    WriteLine($"ToLocalTime(): {dateTime.ToLocalTime()}");
    WriteLine($"ToUniversalTime(): {dateTime.ToUniversalTime()}");
  }

  private static void OutputTimeZone(TimeZoneInfo zone, string title)
  {
    SectionTitle(title);
    WriteLine($"Id: {zone.Id}");
```

```
      WriteLine($"IsDaylightSavingTime(DateTime.Now): {
        zone.IsDaylightSavingTime(DateTime.Now)}");
      WriteLine($"StandardName: {zone.StandardName}");
      WriteLine($"DaylightName: {zone.DaylightName}");
      WriteLine($"BaseUtcOffset: {zone.BaseUtcOffset}");
    }

    private static string GetCurrentZoneName(TimeZoneInfo zone, DateTime when)
    {
      // Названия часовых поясов меняются, если действует
      // переход на летнее время.
      // Например, стандартное время GMT становится летним временем GMT
      return zone.IsDaylightSavingTime(when) ?
        zone.DaylightName : zone.StandardName;
    }
  }
```

4. Удалите содержимое файла `Program.cs`. Добавьте в него код для вывода текущих даты и времени в UTC и локальном формате. Выведите информацию о местном часовом поясе и UTC.:

```
OutputTimeZones();

OutputDateTime(DateTime.Now, "DateTime.Now");
OutputDateTime(DateTime.UtcNow, "DateTime.UtcNow");

OutputTimeZone(TimeZoneInfo.Local, "TimeZoneInfo.Local");
OutputTimeZone(TimeZoneInfo.Utc, "TimeZoneInfo.Utc");
```

5. Запустите консольное приложение и проверьте список часовых поясов в вашей операционной системе. На моем ноутбуке под управлением операционной системы Windows 11 доступен 141 часовой пояс, а также были выведены дата/время: 16:17 31 мая 2022 года. Я нахожусь в Англии, поэтому используется стандартное время GMT. Данное время отстает от UTC на 1 час из-за перехода на летнее время (`GMT Summer Time`):

```
*** 141 time zones:
Afghanistan Standard Time
Alaskan Standard Time
...
West Pacific Standard Time
Yakutsk Standard Time
Yukon Standard Time
*** DateTime.Now
Value: 31/05/2022 16:17:03
Kind: Local
IsDaylightSavingTime: True
ToLocalTime(): 31/05/2022 16:17:03
ToUniversalTime(): 31/05/2022 15:17:03
```

```
*** DateTime.UtcNow
Value: 31/05/2022 15:17:03
Kind: Utc
IsDaylightSavingTime: False
ToLocalTime(): 31/05/2022 16:17:03
ToUniversalTime(): 31/05/2022 15:17:03
*** TimeZoneInfo.Local
Id: GMT Standard Time
IsDaylightSavingTime(DateTime.Now): True
StandardName: GMT Standard Time
DaylightName: GMT Summer Time
BaseUtcOffset: 00:00:00
*** TimeZoneInfo.Utc
Id: UTC
IsDaylightSavingTime(DateTime.Now): False
StandardName: Coordinated Universal Time
DaylightName: Coordinated Universal Time
BaseUtcOffset: 00:00:00
```

Свойство BaseUtcOffset для стандартного времени GMT равно 0, так как стандартное время не включает поправки на летнее время. Именно поэтому оно предваряется префиксом Base.

6. В файле `Program.cs` добавьте код, который:
   - предложит пользователю ввести интересующий его часовой пояс (по умолчанию — североамериканское восточное время);
   - выведет информацию об указанном часовом поясе;
   - сравнит указанное время с эквивалентным временем в другом часовом поясе;
   - обработает возможные исключения:

```
Write("Enter a time zone or press Enter for US East Coast: ");
string zoneId = ReadLine()!;

if (string.IsNullOrEmpty(zoneId))
{
  zoneId = "Eastern Standard Time";
}

try
{
  TimeZoneInfo otherZone = TimeZoneInfo.FindSystemTimeZoneById(zoneId);
  OutputTimeZone(otherZone,
    $"TimeZoneInfo.FindSystemTimeZoneById(\"{zoneId}\")");

  SectionTitle($"What's the time in {zoneId}?");
```

```
    Write("Enter a local time or press Enter for now: ");
    string? timeText = ReadLine();
    DateTime localTime;
    if (string.IsNullOrEmpty(timeText) ||
      !DateTime.TryParse(timeText, out localTime))
    {
      localTime = DateTime.Now;
    }

    DateTime otherZoneTime = TimeZoneInfo.ConvertTime(
      dateTime: localTime, sourceTimeZone: TimeZoneInfo.Local,
      destinationTimeZone: otherZone);

    WriteLine($"{localTime} {GetCurrentZoneName(TimeZoneInfo.Local, localTime)}
      is {otherZoneTime} {GetCurrentZoneName(otherZone, otherZoneTime)}.");
  }
  catch (TimeZoneNotFoundException)
  {
    WriteLine($"The {zoneId} zone cannot be found on the local system.");
  }
  catch (InvalidTimeZoneException)
  {
    WriteLine($"The {zoneId} zone contains invalid or missing data.");
  }
  catch (System.Security.SecurityException)
  {
    WriteLine("The application does not have permission to read time zone
  ➥ information.");
  }
  catch (OutOfMemoryException)
  {
    WriteLine($"Not enough memory is available to load information
  ➥ on the {zoneId} zone.");
  }
```

7. Запустите приложение и попробуйте:
   - нажать клавишу Enter, чтобы выбрать североамериканское восточное время;
   - ввести `12:30 PM` и посмотрите, как оно соотносится с UTC и вашим локальным временем:

```
Enter a time zone or press Enter for US East Coast:
*** TimeZoneInfo.FindSystemTimeZoneById("Eastern Standard Time")
Id: Eastern Standard Time
IsDaylightSavingTime(DateTime.Now): True
StandardName: Eastern Standard Time
DaylightName: Eastern Summer Time
BaseUtcOffset: -05:00:00
*** What's the time in Eastern Standard Time?
Enter a local time or press Enter for now: 12:30pm
31/05/2023 12:30:00 GMT Summer Time is 31/05/2023 07:30:00 Eastern Summer
Time.
```

Если ваш текущий пояс — GMT, то разница во времени с Восточным побережьем США составит 5 часов.

8. Можно также скопировать любой часовой пояс из списка, вставить его в окно ввода и нажать клавишу Enter, чтобы проверить разницу во времени:

```
Enter a time zone or press Enter for US East Coast: AUS Eastern Standard
Time
*** TimeZoneInfo.FindSystemTimeZoneById("AUS Eastern Standard Time")
Id: AUS Eastern Standard Time
IsDaylightSavingTime(DateTime.Now): False
StandardName: AUS Eastern Standard Time
DaylightName: AUS Eastern Summer Time
BaseUtcOffset: 10:00:00
*** What's the time in AUS Eastern Standard Time?
Enter a local time or press Enter for now:
31/05/2023 17:00:04 GMT Summer Time is 01/06/2023 02:00:04 AUS Eastern
Standard Time.
```

Например, разница во времени между вашим часовым поясом и Сиднеем (Австралия) составляет 9 часов. Когда у вас 17:00, в Сиднее уже 2 часа ночи следующего дня.

Часовые пояса — лишь часть работы с датами и временем. Следующий важный аспект — культуры (`CultureInfo`), которые определяют форматирование дат, чисел и других данных в зависимости от языка и региона.

Применение культур

Интернационализация — процесс, позволяющий программе корректно работать в разных странах. Он включает в себя два этапа: *глобализацию* и *локализацию*, оба из которых связаны с культурами.

Глобализация — это создание кода, поддерживающего работу с несколькими языками и регионами. Сочетание языка и региона называется *культурой*. Это важно учитывать, поскольку, например, форматы даты и денежных единиц различаются в Квебеке и Париже, несмотря на то что жители обоих регионов говорят на французском языке.

Для всех культур созданы коды *Международной организации по стандартизации (ISO)*. Например, в коде `da-DK` значение `da` обозначает язык (датский), а символы `DK` — регион (Дания). В коде `fr-CA` значение `fr` означает язык (французский), а `CA` — регион (Канада).

ISO — не просто аббревиатура. Это отсылка к греческому слову isos («равный»). Список ISO-кодов культур можно посмотреть по адресу https://lonewolfonline.net/list-net-culture-country-codes/.

Локализация — настройка пользовательского интерфейса для поддержки конкретного языка, например изменение надписи на кнопке с Close (`en`) на Закрыть (`ru`). Поскольку локализация касается прежде всего языка, ей не всегда нужно учитывать регион. Однако различия в написании слов, например *standardization* (`en-US`) и *standardisation* (`en-GB`), показывают, что региональный фактор все же имеет значение.

Дельный совет: меня нельзя назвать профессиональным переводчиком пользовательских интерфейсов, поэтому рассматривайте примеры в этой главе как общее руководство. Если вам нужна точность, лучше привлечь специалиста. В качестве примера полезные ссылки для перевода с английского языка на французский: https://french.stackexchange.com/questions/12969/translation-of-it-terms-like-close-next-search-etc и https://www.linguee.com/english-french/translation/close+button.html.

Определение и изменение текущей культуры

Интернационализация — обширная тема, о которой написано множество книг. В этом разделе приведены основы с использованием типов `CultureInfo` и `RegionInfo` из пространства имен `System.Globalization`.

Займемся кодом.

1. В редакторе кода создайте консольное приложение `WorkingWithCultures` и добавьте его в решение `Chapter07`.
   - В файле проекта добавьте строгую обработку предупреждений и статически импортируйте класс `System.Console` на глобальном уровне. Кроме того, на глобальном уровне импортируйте пространство имен `System.Globalization`, чтобы использовать класс `CultureInfo`:

```
<ItemGroup>
  <Using Include="System.Console" Static="true" />
  <Using Include="System.Globalization" />
</ItemGroup>
```

2. Создайте файл класса `Program.Helpers.cs` и добавьте в класс `Program` метод для вывода информации о культурах, используемых для глобализации и локализации:

```
partial class Program
{
```

```
  private static void OutputCultures(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkYellow;

    WriteLine($"*** {title}");

    // Извлечение культуры из текущего потока
    CultureInfo globalization = CultureInfo.CurrentCulture;
    CultureInfo localization = CultureInfo.CurrentUICulture;

    WriteLine($"The current globalization culture is {
      globalization.Name}: {globalization.DisplayName}");

    WriteLine($"The current localization culture is {
      localization.Name}: {localization.DisplayName}");

    WriteLine($"Days of the week: {string.Join(", ",
      globalization.DateTimeFormat.DayNames)}");

    WriteLine($"Months of the year: {string.Join(", ",
      globalization.DateTimeFormat.MonthNames
      // У некоторых 13 месяцев; у большинства — 12, а последний пуст
      .TakeWhile(month => !string.IsNullOrEmpty(month)))}");

    WriteLine($"1st day of this year: {new DateTime(
      year: DateTime.Today.Year, month: 1, day: 1)
      .ToString("D", globalization)}");

    WriteLine($"Number group separator: {globalization
      .NumberFormat.NumberGroupSeparator}");

    WriteLine($"Number decimal separator: {globalization
      .NumberFormat.NumberDecimalSeparator}");

    RegionInfo region = new(globalization.LCID);

    WriteLine($"Currency symbol: {region.CurrencySymbol}");

    WriteLine($"Currency name: {region.CurrencyNativeName} ({
      region.CurrencyEnglishName})");

    WriteLine($"IsMetric: {region.IsMetric}");

    WriteLine();

    ForegroundColor = previousColor;
  }
}
```

3. Удалите содержимое файла `Program.cs` и настройте Unicode-кодировку вывода консоли. Добавьте код, выводящий текущую культуру. Затем предложите пользователю ввести новый код культуры и продемонстрируйте влияние смены настроек на форматирование дат и денежных единиц:

```
// Для добавления специальных символов, например €
OutputEncoding = System.Text.Encoding.UTF8;

OutputCultures("Current culture");

WriteLine("Example ISO culture codes:");

string[] cultureCodes = {
  "da-DK", "en-GB", "en-US", "fa-IR",
  "fr-CA", "fr-FR", "he-IL", "pl-PL", "sl-SI" };

foreach (string code in cultureCodes)
{
  CultureInfo culture = CultureInfo.GetCultureInfo(code);
  WriteLine($" {culture.Name}: {culture.EnglishName} / {culture.NativeName}");
}

WriteLine();

Write("Enter an ISO culture code: ");
string? cultureCode = ReadLine();

if (string.IsNullOrWhiteSpace(cultureCode))
{
  cultureCode = "en-US";
}

CultureInfo ci;

try
{
  ci = CultureInfo.GetCultureInfo(cultureCode);
}
catch (CultureNotFoundException)
{
  WriteLine($"Culture code not found: {cultureCode}");
  WriteLine("Exiting the app.");
  return;
}

// изменение текущей культуры в потоке
CultureInfo.CurrentCulture = ci;
CultureInfo.CurrentUICulture = ci;

OutputCultures("After changing the current culture");
```

```
Write("Enter your name: ");
string? name = ReadLine();
if (string.IsNullOrWhiteSpace(name))
{
  name = "Bob";
}

Write("Enter your date of birth: ");
string? dobText = ReadLine();

if (string.IsNullOrWhiteSpace(dobText))
{
  // Если пользователь не ввел дату рождения (DOB), используйте значения
  // по умолчанию, соответствующие культуре, заданной в системе пользователя
  dobText = ci.Name switch
  {
    "en-US" or "fr-CA" => "1/27/1990",
    "da-DK" or "fr-FR" or "pl-PL" => "27/1/1990",
    "fa-IR" => "1990/1/27",
    _ => "1/27/1990"
  };
}

Write("Enter your salary: ");
string? salaryText = ReadLine();

if (string.IsNullOrWhiteSpace(salaryText))
{
  salaryText = "34500";
}

DateTime dob = DateTime.Parse(dobText);
int minutes = (int)DateTime.Today.Subtract(dob).TotalMinutes;
decimal salary = decimal.Parse(salaryText);

WriteLine($"{name} was born on a {dob:dddd}. {name} is {
  minutes:N0} minutes old. {name} earns {salary:C}.");
```

При запуске приложения культура устанавливается по умолчанию в соответствии с настройками операционной системы. Например, если код выполняется в Лондоне, поток получит настройку «английский (Великобритания)».

Приложение позволяет пользователю ввести альтернативный ISO-код, чтобы менять культуру на этапе выполнения. В коде используются стандартные форматы:

- `dddd` — день недели;
- `N0` — число с разделителями тысяч;
- символ денежных единиц.

Эти параметры автоматически адаптируются к выбранной культуре.

4. Выполните код и введите ISO-код `en-US` (или нажмите клавишу Enter). Введите примеры данных, например дату в американском формате:

```
*** Current culture
The current globalization culture is en-GB: English (United Kingdom)
The current localization culture is en-GB: English (United Kingdom)
Days of the week: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
Months of the year: January, February, March, April, May, June, July, August,
September, October, November, December
1st day of this year: 01 January 2023
Number group separator: ,
Number decimal separator: .
Currency symbol: £
Currency name: British Pound (British Pound)
IsMetric: True

Example ISO culture codes:
  da-DK: Danish (Denmark) / dansk (Danmark)
  en-GB: English (United Kingdom) / English (United Kingdom)
  en-US: English (United States) / English (United States)
  fa-IR: Persian (Iran)  / فارسی (ایران)
  fr-CA: French (Canada) / français (Canada)
  fr-FR: French (France) / français (France)
  he-IL: Hebrew (Israel) / עברית (ישראל)
  pl-PL: Polish (Poland) / polski (Polska)
  sl-SI: Slovenian (Slovenia) / slovenščina (Slovenija)

Enter an ISO culture code: en-US
*** After changing the current culture
The current globalization culture is en-US: English (United States)
The current localization culture is en-US: English (United States)
Days of the week: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
Months of the year: January, February, March, April, May, June, July, August,
September, October, November, December
1st day of this year: Sunday, January 1, 2023
Number group separator: ,
Number decimal separator: .
Currency symbol: $
Currency name: US Dollar (US Dollar)
IsMetric: False

Enter your name: Alice
Enter your date of birth: 3/30/1967
Enter your salary: 34500
Alice was born on a Thursday. Alice is 29,541,600 minutes old. Alice earns
$34,500.00
```

5. Выполните код снова и попробуйте использовать датский язык (`da-DK`):

```
Enter an ISO culture code: da-DK
*** After changing the current culture
The current globalization culture is da-DK: dansk (Danmark)
```

```
The current localization culture is da-DK: dansk (Danmark)
Days of the week: søndag, mandag, tirsdag, onsdag, torsdag, fredag, lørdag
Months of the year: januar, februar, marts, april, maj, juni, juli, august,
september, oktober, november, december
1st day of this year: søndag den 1. januar 2023
Number group separator: .
Number decimal separator: ,
Currency symbol: kr.
Currency name: dansk krone (Danish Krone)
IsMetric: True

Enter your name: Mikkel
Enter your date of birth: 16/3/1980
Enter your salary: 65000
Mikkel was born on a søndag. Mikkel is 22.723.200 minutes old. Mikkel earns
65.000,00 kr..
```

Вы увидите изменения в формате даты и денежной единицы, но текст интерфейса остается на английском языке. Позже мы переведем интерфейс на другие языки. А пока давайте посмотрим на другие различия между культурами.

6. Выполните код снова и попробуйте использовать польский язык (`pl-PL`). Обратите внимание, что по правилам грамматики в польском языке название месяца указывается в родительном падеже после порядкового числительного даты, поэтому месяц `styczeń` превращается в `stycznia`:

```
The current globalization culture is pl-PL: polski (Polska)
...
Months of the year: styczeń, luty, marzec, kwiecień, maj, czerwiec, lipiec,
sierpień, wrzesień, październik, listopad, grudzień
1st day of this year: niedziela, 1 stycznia 2023
...
Enter your name: Bob
Enter your date of birth: 1972/4/16
Enter your salary: 50000
Bob was born on a niedziela. Bob is 26 886 240 minutes old.
Bob earns 50 000,00 zł.
```

7. Выполните код снова и введите персидский язык (фарси) в Иране (`fa-IR`). Как видно из вывода, даты в Иране указываются в виде «год/месяц/день» и используется летоисчисление по персидскому календарю — 2023 год представлен значением `1401`:

```
The current globalization culture is fa-IR: فارسی (ایران)
The current localization culture is fa-IR: فارسی (ایران)
Days of the week: یکشنبه, دوشنبه, سه شنبه, چهارشنبه, پنجشنبه, جمعه,
شنبه
Months of the year: فروردین, اردیبهشت, خرداد, تیر, مرداد, شهریور,
```

```
د ف ن ا ,ه م ب ,ی د ,ر ذ آ ,ن ا ب آ ,ر ه م
1st day of this year: ه ب ش , 11 ی 1401
Number group separator: ,
Number decimal separator: ,
Currency symbol: ل ا ی ر
Currency name: ل ا ی ر ا ی ر ا ن (Iranian Rial)
IsMetric: True

Enter your name: Cyrus
Enter your date of birth: 1372/4/16
Enter your salary: 50000
Cyrus was born on a ه ب ن ش ا ر ه چ . Cyrus is 15,723,360 minutes old. Cyrus earns
50,000 ل ا ی ر .
```

Работа с языками с письмом справа налево может быть сложной в консольных приложениях. Я постарался проверить корректность примеров у читателя-перса, но прошу прощения у носителей персидского языка за возможные неточности!

Временное использование инвариантной культуры

Иногда может возникнуть необходимость временно использовать другую культуру без изменения текущего потока. Например, при автоматической генерации документов, запросов и команд, которые содержат данные, важно игнорировать текущую культуру и применить более стандартизированную. Для этого можно использовать инвариантную культуру, основанную на американском английском языке.

Пример: вам нужно сгенерировать JSON-документ с десятичным числом, отформатированным с двумя знаками после запятой.

```
decimal price = 54321.99M;
string document = $$"""
  {
    "price": "{{price:N2}}"
  }
  """;
```

Если бы вы выполнили этот код на компьютере с установленной словенской культурой, результат был бы следующим:

```
{
  "price": " 54.321,99"
}
```

Однако если попытаться добавить этот JSON-документ в облачную БД, это не удастся, так как не будет распознан формат чисел, где запятая используется для десятичных дробей, а точка — для разделения групп знаков.

Чтобы решить эту проблему, можно временно переопределить текущую культуру и указать инвариантную культуру только при выводе числа в виде строки:

```
decimal price = 54321.99M;
string document = $$"""
  {
    "price": "{{price.ToString("N2", CultureInfo.InvariantCulture)}}"
  }
  """;
```

В этом случае, даже если код выполняется на компьютере с другой культурой, например словенской, результат будет успешно распознан облачной БД и не вызовет исключений:

```
{
  "price": " 54,321.99"
}
```

Теперь посмотрим, как перевести текст с одного языка на другой, чтобы всплывающие подсказки отображались на правильном языке в зависимости от текущей культуры.

Локализация пользовательского интерфейса

Локализованное приложение состоит из двух частей:

- основная сборка, содержащая общий код для всех локалей и ресурсы для случаев, когда другие файлы ресурсов не найдены;
- одна или несколько *сборок-сателлитов*, содержащих ресурсы для различных локалей, которые отличаются в зависимости от языка и региона.

Такая модель позволяет развернуть приложение с неизменными ресурсами по умолчанию, а по мере перевода можно добавлять дополнительные сборки-сателлиты. В рамках задания по программированию вам предстоит создать консольное приложение с инвариантной культурой и сборками-сателлитами для датской, французской, франко-канадской, польской и иранской (персидской) культур. Для добавления новых культур в будущем нужно будет выполнить те же шаги.

Ресурсы пользовательского интерфейса включают текстовые данные для сообщений, журналов, диалоговых окон, кнопок, ярлыков, а также имена файлов изображений и видео. Файлы ресурсов — это XML-файлы с расширением `.resx`. Имя файла обычно включает код культуры, например `PacktResources.en-GB.resx` или `PacktResources.da-DK.resx`.

Если файл ресурса или конкретная запись отсутствуют, система будет искать ресурсы в следующем порядке: сначала по коду культуры (язык и регион), затем по нейтральной культуре (только язык) и, наконец, по инвариантной культуре (предположительно, американский английский). Например, если текущая культура потока — `en-AU` (австралийский английский), поиск файла ресурса будет осуществляться по следующему порядку:

1) `PacktResources.en-AU.resx` — австралийский английский;
2) `PacktResources.en.resx` — нейтральный английский;
3) `PacktResources.resx` — инвариантная культура.

Определение и загрузка ресурсов

Для загрузки ресурсов из сборок-сателлитов в .NET используются стандартные интерфейсы `IStringLocalizer<T>` и `IStringLocalizerFactory`. Эти интерфейсы загружаются из универсального узла .NET через зависимости.

1. В проект `WorkingWithCultures` добавьте ссылки на пакеты расширений Microsoft для работы с универсальным узлом и локализацией:

```
<ItemGroup>
  <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0" />
  <PackageReference Include="Microsoft.Extensions.Localization" Version="8.0.0" />
</ItemGroup>
```

2. Соберите проект для восстановления зависимостей.
3. В папке проекта создайте каталог `Resources`.
4. В папке `Resources` создайте XML-файл `PacktResources.resx`, который будет содержать ресурсы инвариантной культуры (обычно американский английский):

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Enter your date of birth: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>Enter your name: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Enter your salary: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} was born on a {1:dddd}. {0} is {2:N0} minutes old.
➥ {0} earns {3:C}.</value>
  </data>
</root>
```

5. В проекте `WorkingWithCultures` создайте файл класса `PacktResources.cs`, который будет загружать текстовые ресурсы для пользовательского интерфейса:

```
using Microsoft.Extensions.Localization; // Для типа IStringLocalizer и т. д.

public class PacktResources
{
  private readonly IStringLocalizer<PacktResources> localizer = null!;

  public PacktResources(IStringLocalizer<PacktResources> localizer)
  {
    this.localizer = localizer;
  }

  public string? GetEnterYourNamePrompt()
  {
    string resourceStringName = "EnterYourName";

    // 1. Получение объекта LocalizedString
    LocalizedString localizedString = localizer[resourceStringName];

    // 2. Проверка, найдена ли строка ресурса
    if (localizedString.ResourceNotFound)
    {
      ConsoleColor previousColor = ForegroundColor;
      ForegroundColor = ConsoleColor.Red;
      WriteLine($"Error: resource string \"{resourceStringName}\" not found."
        + Environment.NewLine
        + $"Search path: {localizedString.SearchedLocation}");
      ForegroundColor = previousColor;

      return $"{localizedString}: ";
    }

    // 3. Возврат строки найденного ресурса
    return localizedString;
  }

  public string? GetEnterYourDobPrompt()
  {
    // Объект LocalizedString имеет неявное приведение к строке, и его
    // значение возвращается к ключу, если строка ресурса не найдена
    return localizer["EnterYourDob"];
  }

  public string? GetEnterYourSalaryPrompt()
  {
    return localizer["EnterYourSalary"];
  }
```

```
  public string? GetPersonDetails(
    string name, DateTime dob, int minutes, decimal salary)
  {
    return localizer["PersonDetails", name, dob, minutes, salary];
  }
}
```

Для метода GetEnterYourNamePrompt я разбил реализацию на шаги для проверки наличия строки ресурса и отображения пути поиска, если ресурс не найден. В других вариантах метода используется упрощенный механизм возврата имени ключа, если ресурс не найден.

6. В верхней части файла `Program.cs` импортируйте пространство имен для работы с зависимостями и настройте хост, который будет поддерживать локализацию и сервис `PacktResources`:

```
using Microsoft.Extensions.Hosting; // Для элементов IHost, Host

// Для методов AddLocalization, AddTransient<T>
using Microsoft.Extensions.DependencyInjection;
using IHost host = Host.CreateDefaultBuilder(args)
  .ConfigureServices(services =>
  {
    services.AddLocalization(options =>
    {
      options.ResourcesPath = "Resources";
    });

    services.AddTransient<PacktResources>();
  })
  .Build();
```

Дельный совет: по умолчанию ResourcesPath — это пустая строка, то есть файлы .resx будут запрашиваться в текущем каталоге. Для структуризации проекта поместите ресурсы в подкаталог.

7. Измените текущую культуру, добавьте код для запроса и применения сервиса `PacktResources` для отображения локализованных окон ввода имени, даты рождения и зарплаты, а затем выведите данные:

```
OutputCultures("After changing the current culture");

PacktResources resources =
    host.Services.GetRequiredService<PacktResources>();

Write(resources.GetEnterYourNamePrompt());
string? name = ReadLine();
```

```
if (string.IsNullOrWhiteSpace(name))
{
  name = "Bob";
}

Write(resources.GetEnterYourDobPrompt());
string? dobText = ReadLine();

if (string.IsNullOrWhiteSpace(dobText))
{
  // Если пользователь не ввел дату рождения (DOB), используйте значения
  // по умолчанию, соответствующие культуре, заданной в системе пользователя
  dobText = ci.Name switch
  {
    "en-US" or "fr-CA" => "1/27/1990",
    "da-DK" or "fr-FR" or "pl-PL" => "27/1/1990",
    "fa-IR" => "1990/1/27",
    _ => "1/27/1990"
  };
}

Write(resources.GetEnterYourSalaryPrompt());
string? salaryText = ReadLine();

if (string.IsNullOrWhiteSpace(salaryText))
{
  salaryText = "34500";
}

DateTime dob = DateTime.Parse(dobText);
int minutes = (int)DateTime.Today.Subtract(dob).TotalMinutes;
decimal salary = decimal.Parse(salaryText);

WriteLine(resources.GetPersonDetails(name, dob, minutes, salary));
```

Тестирование глобализации и локализации

Теперь мы можем запустить консольное приложение и проверить загрузку ресурсов.

1. Запустите приложение и введите ISO-код `da-DK`. Обратите внимание, что подсказки отображаются на английском языке, так как в данный момент доступны только ресурсы инвариантной культуры.

Чтобы ускорить процесс, вы можете скопировать и переименовать существующие файлы .resx вместо создания новых с нуля или скачать эти файлы из GitHub-репозитория книги.

2. В папке `Resources` создайте XML-файл `PacktResources.da.resx` с ресурсами датского языка без привязки к региону:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Indtast din fødselsdato: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>Indtast dit navn: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Indtast din løn: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} blev født på en {1:dddd}. {0} er {2:N0} minutter gammel.
➥ {0} tjener {3:C}.</value>
  </data>
</root>
```

3. В той же папке создайте XML-файл `PacktResources.fr.resx` с ресурсами французского языка без привязки к региону:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Entrez votre date de naissance: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>Entrez votre nom: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Entrez votre salaire: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} est né un {1:dddd}. {0} a {2:N0} minutes.
➥ {0} gagne {3:C}.</value>
  </data>
</root>
```

4. В той же папке создайте XML- `PacktResources.fr-CA.resx` с ресурсами французского языка для Канады:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Entrez votre date de naissance / Enter your date of birth: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
```

```
    <value>Entrez votre nom / Enter your name: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Entrez votre salaire / Enter your salary: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} est né un {1:dddd}. {0} a {2:N0} minutes.
➥ {0} gagne {3:C}.</value>
  </data>
</root>
```

5. В той же папке создайте `XML`-файл `PacktResources.pl-PL.resx` с ресурсами польского языка для Польши:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Wpisz swoją datę urodzenia: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value>Wpisz swoje imię i nazwisko: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Wpisz swoje wynagrodzenie: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} urodził się na {1:dddd}. {0} ma {2:N0} minut.
➥ {0} zarabia {3:C}.</value>
  </data>
</root>
```

6. В той же папке создайте XML-файл `PacktResources.fa-IR.resx` с ресурсами персидского языка для Ирана:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value> دیک دراو ار دوخ دلوت خیرات / Enter your date of birth: </value>
  </data>
  <data name="EnterYourName" xml:space="preserve">
    <value> نک دراو ار تمسا / Enter your name: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value> دیک دراو ار دوخ قوقح / Enter your salary: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} رد {1:dddd}ب هیدآ ایند. {0} {2:N 0} دقیقه است.
➥ { 0 } {3:C}.</value>
  </data>
</root>
```

7. Выполните код и введите ISO-код `da-DK`. Обратите внимание, что подсказки отображаются на датском языке:

```
The current localization culture is da-DK: dansk (Danmark)
...
Indtast dit navn: Bob
Indtast din fødselsdato: 3/4/1987
Indtast din løn: 45449
Bob blev født på en fredag. Bob er 19.016.640 minutter gammel. Bob tjener
45.449,00 kr.
```

8. Выполните код и введите ISO-код `fr-FR`. Обратите внимание, что подсказки отображаются на французском языке:

```
The current localization culture is fr-FR: français (France)
...
Entrez votre nom: Monique
Entrez votre date de naissance: 2/12/1990
Entrez votre salaire: 45000
Monique est né un dimanche. Monique a 17 088 480 minutes. Monique gagne
45 000,00 €.
```

9. Выполните код и введите ISO-код `fr-CA`. Обратите внимание, что подсказки отображаются на французском и английском языках, потому что в Канаде два официальных языка:

```
The current localization culture is fr-CA: français (Canada)
...
Entrez votre nom / Enter your name: Sophie
Entrez votre date de naissance / Enter your date of birth: 4/5/2001
Entrez votre salaire / Enter your salary: 65000
Sophie est né un jeudi. Sophie a 11 649 600 minutes. Sophie gagne 65
000,00 $ CA.
```

10. Выполните код и введите ISO-код `fa-IR`. Подсказки отображаются на персидском/фарси и английском языках и частично написаны справа налево:

```
The current localization culture is fa-IR: فارسی (ایران)
...
نام را وارد کنید / Enter your name: Hoshyar
تاریخ تولد خود را وارد کنید / Enter your date of birth: 1370/3/6
حقوق خود را وارد کنید / Enter your salary: 90000
Hoshyar در چه ارشاره به دنیا آمده است . Hoshyar دقیقه 11,190,240 . Hoshyar
90,000 ریال .
```

Для работы с персидскими датами в репозитории GitHub есть соответствующие NuGet-пакеты с открытым исходным кодом, такие как https://github.com/VahidN/DNTPersianUtils.Core и https://github.com/imanabidi/PersianDate.NET.

11. В папке `Resources` откройте файл `PacktResources.da.resx` и внесите в ключ для запроса ввода имени ошибку (добавив слово `Wrong`, как выделено в листинге ниже):

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="EnterYourDob" xml:space="preserve">
    <value>Indtast din fødselsdato: </value>
  </data>
  <data name="EnterYourNameWrong" xml:space="preserve">
    <value>Indtast dit navn: </value>
  </data>
  <data name="EnterYourSalary" xml:space="preserve">
    <value>Indtast din løn: </value>
  </data>
  <data name="PersonDetails" xml:space="preserve">
    <value>{0} blev født på en {1:dddd}. {0} er {2:N0} minutter gammel.
➥ {0} tjener {3:C}.</value>
  </data>
</root>
```

12. Запустите приложение с ISO-кодом `da-DK` и обратите внимание, что подсказки отображаются на датском, но запрос на ввод имени остается на английском, так как используется файл ресурса по умолчанию:

```
The current localization culture is da-DK: dansk (Danmark)
...
Enter your name: Bob
Indtast din fødselsdato: 3/4/1987
Indtast din løn: 45449
Bob blev født på en fredag. Bob er 18.413.280 minutter gammel. Bob tjener
45.449,00 kr.
```

13. В папке `Resources` откройте файл `PacktResources.resx` и внесите в ключ для запроса ввода имени ошибку (добавив слово `Wrong`).
14. Запустите приложение с ISO-кодом `da-DK` и обратите внимание, что подсказки отображаются на датском, кроме запроса на ввод имени. Подсказка для имени не будет найдена, и приложение отобразит ключ вместо строки:

```
The current localization culture is da-DK: dansk (Danmark)
...
Error: resource string "EnterYourName" not found.
```

```
Search path: WorkingWithCultures.Resources.PacktResources
EnterYourName: Bob
Indtast din fødselsdato: 3/4/1987
Indtast din løn: 45449
Bob blev født på en fredag. Bob er 18.413.280 minutter gammel. Bob tjener
45.449,00 kr.
```

15. Удалите суффикс `Wrong` в обоих файлах ресурсов.
16. На панели Solution Explorer (Обозреватель решений) нажмите кнопку Show All Files (Показать все файлы) и раскройте папку `bin/Debug/net8.0/da` (рис. 7.1).

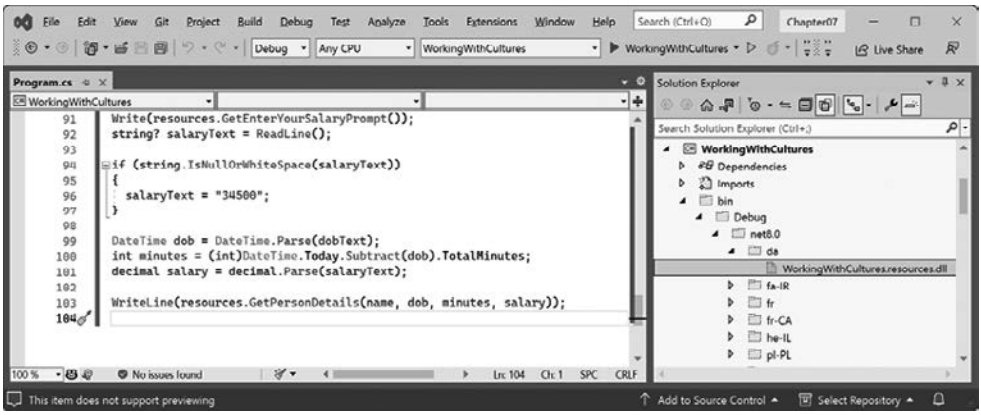


Рис. 7.1. Папки сборки-сателлита для ресурсов культур

17. В ней вы увидите сборку-сателлит `WorkingWithCultures.resources.dll` с датскими ресурсами без привязки к региону.

Другие сборки ресурсов именуются так же, только располагаются в папках, соответствующих коду культуры. Для удобной работы с массивами файлов `.resx`, их компиляции в сборки-сателлиты и последующего развертывания без необходимости повторной компиляции основного консольного приложения можно воспользоваться инструментами наподобие *ResX Resource Manager* (доступен по ссылке https://dotnetfoundation.org/projects/resx-resource-manager).

Дельный совет: перед началом разработки продумайте необходимость интернационализации, форматы данных, чисел, сортировку текста и глобализируемые элементы. Запишите все текстовые элементы пользовательского интерфейса, которые необходимо локализовать.

Для локализации приложений рекомендуется использовать *набор средств многоязычных приложений (Multilingual App Toolkit, MAT)*, который интегрируется в Microsoft Visual Studio и предоставляет поддержку перевода, управления файла-

ми перевода и редакторских инструментов. MAT позволяет эффективно добавлять новые языки и управлять процессом локализации в ваших проектах (рис. 7.2).

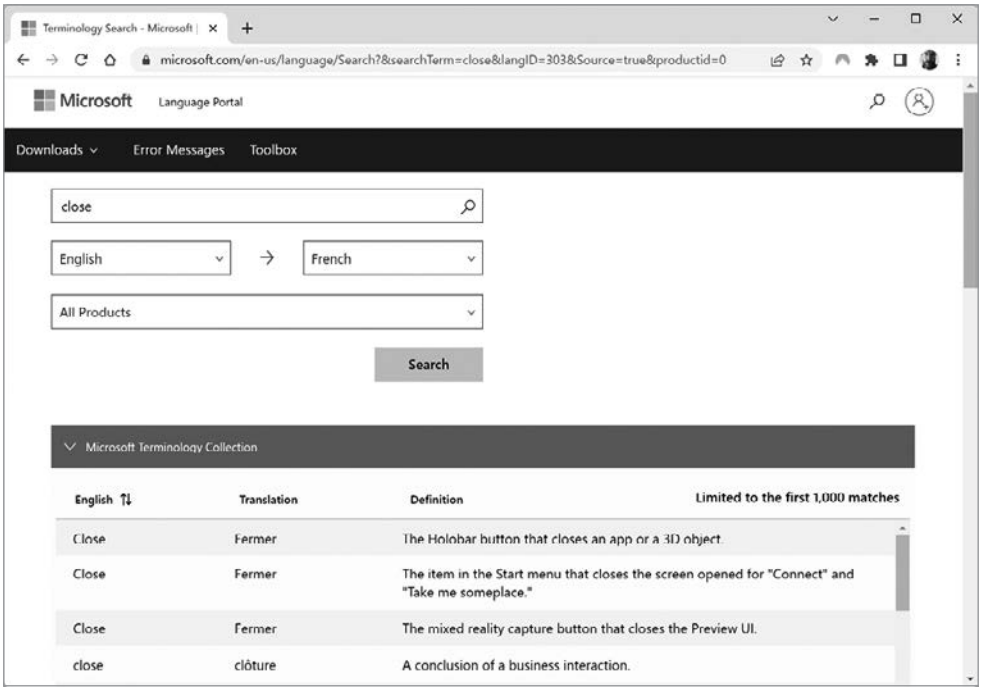


Рис. 7.2. Пользовательский интерфейс онлайн-инструмента Microsoft для локализации текста

Вы познакомились со множеством функций библиотеки базовых классов .NET, предназначенных для работы с датами и временем. Однако для полноценной интернационализации их может быть недостаточно. В таких случаях стоит обратить внимание на библиотеку Noda Time.

Библиотека Noda Time

Библиотека Noda Time предназначена для разработчиков, которым встроенные механизмы работы с датами и временем в .NET кажутся недостаточно гибкими. Ее разработчики вдохновились проектом Joda Time — альтернативной реализацией API для Java.

Библиотека Noda Time версии 3.0 и выше поддерживает .NET Standard 2.0 и .NET 6 или новее, поэтому она совместима как со старыми платформами (например, .NET Framework и Xamarin), так и с современными версиями .NET.

Чтобы понять один из основных недостатков встроенных в .NET типов для работы с датами и временем, проведем аналогию. Представьте, что вместо отдельных числовых типов, таких как `int` (`System.Int32`), `double` (`System.Double`) и `decimal` (`System.Decimal`), используется только один тип `System.Number` со свойством `Kind`. Это свойство определяет тип числа, а также способы хранения в памяти и обработки.

Именно это произошло с типом `System.DateTime`. У него есть свойство `Kind`, указывающее, является ли время локальным, UTC или неопределенным. Поведение объекта зависит от способа использования, поэтому разобраться с `DateTime` — задача нетривиальная.

В копилку знаний: создатель библиотеки Noda Time Джон Скит подробно разобрал ограничения встроенных в .NET типов для работы с датами и временем и DateTime, в частности, в своей статье по адресу https://blog.nodatime.org/2011/08/what-wrong-with-datetime-anyway.html.

Важные понятия и значения по умолчанию в Noda Time

Встроенный в .NET тип `DateTime` может хранить как глобальные, так и локальные значения, а также значения без привязки к конкретному времени. Если работать с ним без должной осторожности, можно столкнуться с ошибками и недоразумениями.

Одно из главных отличий Noda Time заключается в предварительном определении типа данных. В результате в Noda Time используется больше типов, что на первых порах может слегка запутать. Типы в Noda Time бывают глобальными и локальными. Глобальные значения одинаковы для всех пользователей по всему миру, в то время как локальные зависят от таких факторов, как часовой пояс.

Точность стандартных типов даты/времени в .NET ограничена единицей «тик», которая равна примерно 100 наносекундам. В Noda Time точность достигает 1 наносекунды, что в 100 раз точнее.

Нулевая точка отсчета в Noda Time — это полночь 1 января 1970 года по времени UTC. Тип `Instant` в Noda Time отображает количество наносекунд до или после этой точки на глобальной временной шкале.

По умолчанию в Noda Time используется календарь ISO-8601, принятый как международный стандарт. Также библиотека поддерживает автоматическое преобразование в другие календари, например юлианский, коптский и буддийский. Подробнее об этом можно узнать по адресу https://en.wikipedia.org/wiki/ISO_8601.

В Noda Time есть несколько распространенных типов, похожих на аналоги в .NET. Я перечислил их в табл. 7.6.

Таблица 7.6. Распространенные типы библиотеки Noda Time

Тип в Noda Time	Описание
Структура `Instant`	Точка времени на глобальной шкале с точностью до наносекунды
Структура `Interval`	Два значения типа `Instant`: начало (включая) и конец (исключая)
Структура `LocalDateTimet`	Дата и время в определенной системе летоисчисления, но не привязанная к точке времени на глобальной шкале. Например, полночь 1 января 2023 года для разных пользователей наступит по-разному, в зависимости от их часового пояса. Если вы не знаете часовой пояс пользователя, вам, скорее всего, понадобится именно этот тип
Структуры `LocalDate` и `LocalTime`	Похожие на `LocalDateTime`, но только для даты или времени. Обычно с ними работают по отдельности, а затем объединяют в `LocalDateTime`
Класс `DateTimeZone`	Часовой пояс. Легко преобразуется из встроенного в .NET типа `TimeZoneInfo` с помощью `BclDateTimeZone`. Для получения часового пояса используйте провайдер `DateTimeZoneProviders.Tzdb`. Список стандартных названий часовых поясов доступен по адресу https://en.wikipedia.org/wiki/List_of_tz_database_time_zones
Структура `ZonedDateTime`	Дата и время в конкретных системе летоисчисления и часовом поясе, поэтому представляет собой фиксированную точку на глобальной шкале времени
Структура `Offset`	Смещение местного времени относительно UTC. Положительное смещение означает, что местное время опережает UTC, а отрицательное — отстает
Структура `OffsetDateTime`	Если нужно учитывать смещение относительно UTC, но оно не соответствует конкретному часовому поясу, используйте этот тип вместо `ZonedDateTime`
Структура `Duration`	Фиксированное количество наносекунд. Поддерживает свойства для преобразования в привычные единицы времени, такие как дни, часы, минуты и секунды, с округлением до целых чисел (`int`), а также неокругленные значения, такие как `TotalDays` и `TotalMinutes`, которые возвращают `double`. Используется для вычислений значений `Instant` и `ZonedDateTime`. Применяйте вместо встроенного в .NET типа `TimeSpan`
Класс `Period`	Переменная продолжительность, которая зависит от длины периода в разные годы. Например, два месяца могут длиться по-разному в високосный и невисокосный год

Дельный совет: используйте тип Instant для точных временных меток. Он идеально подходит для записи моментов времени. После этого его можно отобразить пользователю в соответствующем часовом поясе. Если нужно записать дату и время, введенные пользователем, лучше использовать типы ZonedDateTime, OffsetDateTime, LocalDateTime, LocalDate или LocalTime.

Преобразование типов даты/времени Noda Time

В целях обобщения распространенных способов преобразования между типами в Noda Time я подготовил следующую диаграмму (неполную) (рис. 7.3).

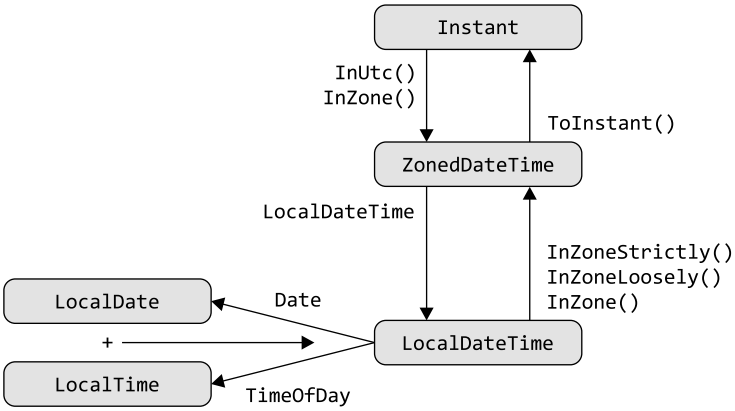


Рис. 7.3. Распространенные способы преобразования типов даты/времени Noda Time

Пример работы с Noda Time в консольном приложении

Напишем немного кода.

1. В редакторе кода создайте консольное приложение `WorkingWithNodaTime` и добавьте его в решение `Chapter07`.

   В файле проекта добавьте строгую обработку предупреждений, статически импортируйте класс `System.Console` на глобальном уровне и подключите пакет `NodaTime`:

```
<ItemGroup>
  <PackageReference Include="NodaTime" Version="3.1.9" />
</ItemGroup>
```

2. Создайте файл класса `Program.Helpers.cs` и замените его содержимое следующим кодом:

```
partial class Program
{
  private static void SectionTitle(string title)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.DarkYellow;
    WriteLine($"*** {title}");
    ForegroundColor = previousColor;
  }
}
```

3. Удалите содержимое файла `Program.cs`, добавьте код для извлечения текущего времени и преобразования его в различные типы Noda Time. Включите примеры для UTC, пары часовых поясов и местного времени:

```
using NodaTime; // Для класса SystemClock, Instant и т. д.

SectionTitle("Converting Noda Time types");

// Извлечение текущего времени
Instant now = SystemClock.Instance.GetCurrentInstant();

WriteLine($"Now (Instant): {now}");
WriteLine();

ZonedDateTime nowInUtc = now.InUtc();

WriteLine($"Now (DateTimeZone): {nowInUtc.Zone}");
WriteLine($"Now (ZonedDateTime): {nowInUtc}");
WriteLine($"Now (DST): {nowInUtc.IsDaylightSavingTime()}");
WriteLine();

// Использование провайдера Tzdb для получения часового пояса
// тихоокеанского региона.
// Чтобы использовать часовые пояса, совместимые с .NET,
// используйте провайдера Bcl.

DateTimeZone zonePT = DateTimeZoneProviders.Tzdb["US/Pacific"];
ZonedDateTime nowInPT = now.InZone(zonePT);

WriteLine($"Now (DateTimeZone): {nowInPT.Zone}");
WriteLine($"Now (ZonedDateTime): {nowInPT}");
WriteLine($"Now (DST): {nowInPT.IsDaylightSavingTime()}");
WriteLine();

DateTimeZone zoneUK = DateTimeZoneProviders.Tzdb["Europe/London"];
ZonedDateTime nowInUK = now.InZone(zoneUK);

WriteLine($"Now (DateTimeZone): {nowInUK.Zone}");
WriteLine($"Now (ZonedDateTime): {nowInUK}");
WriteLine($"Now (DST): {nowInUK.IsDaylightSavingTime()}");
WriteLine();
```

```
LocalDateTime nowInLocal = nowInUtc.LocalDateTime;

WriteLine($"Now (LocalDateTime): {nowInLocal}");
WriteLine($"Now (LocalDate): {nowInLocal.Date}");
WriteLine($"Now (LocalTime): {nowInLocal.TimeOfDay}");
WriteLine();
```

4. Запустите приложение и посмотрите на результаты. Обратите внимание, что «местное» время может не учитывать переход на летнее время (DST). Например, если вы живете в Великобритании, используйте часовой пояс Лондона, чтобы учесть британское летнее время (например, 10:21 утра), а не «местное» (09:21 утра):

```
*** Converting Noda Time types
Now (Instant): 2023-06-01T09:21:05Z

Now (DateTimeZone): UTC
Now (ZonedDateTime): 2023-06-01T09:21:05 UTC (+00)
Now (DST): False

Now (DateTimeZone): US/Pacific
Now (ZonedDateTime): 2023-06-01T02:21:05 US/Pacific (-07)
Now (DST): True

Now (DateTimeZone): Europe/London
Now (ZonedDateTime): 2023-06-01T10:21:05 Europe/London (+01)
Now (DST): True

Now (LocalDateTime): 01/06/2023 09:21:05
Now (LocalDate): 01 June 2023
Now (LocalTime): 09:21:05
```

5. В файле `Program.cs` добавьте код для изучения работы с периодами времени и посмотрите, как с ними можно работать в Noda Time:

```
SectionTitle("Working with periods");

// Современная эра .NET началась с выходом .NET Core 1.0
// 27 июня 2016 года в 10 утра по тихоокеанскому времени
// или в 17 часов по Гринвичу
LocalDateTime start = new(year: 2016, month: 6, day: 27, hour: 17,
➥ minute: 0, second: 0);
LocalDateTime end = LocalDateTime.FromDateTime(DateTime.UtcNow);

WriteLine("Modern .NET era");
WriteLine($"Start: {start}");
WriteLine($"End: {end}");
WriteLine();
```

```
Period period = Period.Between(start, end);

WriteLine($"Period: {period}");
WriteLine($"Years: {period.Years}");
WriteLine($"Months: {period.Months}");
WriteLine($"Weeks: {period.Weeks}");
WriteLine($"Days: {period.Days}");
WriteLine($"Hours: {period.Hours}");
WriteLine();

Period p1 = Period.FromWeeks(2);
Period p2 = Period.FromDays(14);

WriteLine($"p1 (period of two weeks): {p1}");
WriteLine($"p2 (period of 14 days): {p2}");
WriteLine($"p1 == p2: {p1 == p2}");
WriteLine($"p1.Normalize() == p2: {p1.Normalize() == p2}");
```

6. Запустите консольное приложение и обратите внимание на результаты. Так, вы увидите, что на момент запуска приложения (у меня 1 июня 2023 года) «возраст» .NET составил 6 лет 11 месяцев 4 дня. Кроме того, можно исследовать формат сериализации типа `Period` и способы сравнения двух периодов, а также посмотреть, как их нормализовать перед сравнением:

```
*** Working with periods
Modern .NET era
Start: 27/06/2016 17:00:00
End: 01/06/2023 09:21:05

Period: P6Y11M4DT16H21M5S889s9240t
Years: 6
Months: 11
Weeks: 0
Days: 4
Hours: 16

p1 (period of two weeks): P2W
p2 (period of 14 days): P14D
p1 == p2: False
p1.Normalize() == p2: True
```

В копилку знаний: метод Normalize позволяет нормализовать период, умножив все недели на 7 и прибавив их к количеству дней. После этого значение Weeks обнуляется и выполняются другие вычисления. Подробнее об этом можно прочитать по адресу https://nodatime.org/3.1.x/api/NodaTime.Period.html#NodaTime_Period_Normalize.

Модульное тестирование и сериализация JSON с помощью Noda Time

Для модульного тестирования в Noda Time есть специальный пакет `NodaTime.Testing`, а для работы с JSON.NET — пакет `NodaTime.Serialization.JsonNet`.

Документация по Noda Time доступна на сайте по адресу https://nodatime.org/.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 7.1. Проверьте свои знания

Ответьте на следующие вопросы (при необходимости пользуйтесь источниками информации в Интернете).

1. В чем разница между локализацией, глобализацией и интернационализацией?
2. Какое наименьшее измерение времени доступно в .NET?
3. Сколько времени длится один тик в .NET?
4. В каких случаях следует использовать значение `DateOnly` вместо `DateTime`?
5. Что обозначает свойство `BaseUtcOffset` для часового пояса?
6. Как получить информацию о локальном часовом поясе, в котором выполняется код?
7. Что означает свойство `Kind` у значения `DateTime`?
8. Как изменить текущую культуру для исполняемого кода?
9. Каков ISO-код валлийского языка?
10. Как используются резервные файлы ресурсов локализации?

Упражнение 7.2. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-7---handling-dates-times-and-internationalization.

Упражнение 7.3. Уроки эксперта Джона Скита

Джон Скит — всемирно известный эксперт по интернационализации. Посмотрите его презентацию Working with Time is Easy по адресу https://www.youtube.com/watch?v=saeKBuPewcU.

Резюме

В этой главе вы:

- изучили работу с датами и временем, включая абстракцию `TimeProvider` из .NET 8, предназначенную для улучшения модульных тестов;
- освоили работу с часовыми поясами и научились правильно учитывать их в приложениях;
- узнали, как интернационализировать код путем глобализации и локализации;
- ознакомились с возможностями библиотеки Noda Time, которая предоставляет удобные инструменты для работы с временем.

В следующей главе вы научитесь создавать веб-сервисы на минимальных API ASP.NET Core, а также обеспечивать их безопасность и защиту.

8 Создание и защита веб-сервисов на основе минимальных API

В этой главе рассмотрена работа с веб-сервисами, построенными на минимальных API ASP.NET Core. Мы разберем методы предотвращения атак, механизмы проверки подлинности и авторизации, а также оптимизацию работы сервисов.

Создание веб-сервисов на основе минимальных API в ASP.NET Core

В ранних версиях фреймворка ASP.NET Core веб-сервисы создавались с помощью контроллеров, где для каждой конечной точки использовался отдельный метод действия. Такой подход был схож с разработкой веб-приложений на ASP.NET Core MVC, но без представлений.

Начиная с версии .NET 6, у вас есть альтернативный, зачастую более предпочтительный вариант: *минимальные API ASP.NET Core*.

Преимущества минимальных API

В ранних версиях ASP.NET Core реализация даже простого веб-сервиса требовала большого количества кода по сравнению с альтернативными платформами разработки веб-приложений. Например, минимальная реализация веб-сервиса Hello World! с единственной конечной точкой, возвращающей обычный текст, может быть реализована с помощью фреймворка *Express.js* всего в девяти строках кода:

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

В ASP.NET Core 5 и более ранних версиях аналогичная реализация занимала более 50 строк кода.

С выходом ASP.NET Core 6 и поддержкой минимальных API этот же веб-сервис можно реализовать всего в пяти строках кода и шести строках конфигурации:

```
int port = 3000;
var app = WebApplication.Create();
app.MapGet("/", () => "Hello World!");
Console.WriteLine($"Example app listening on port {port}");
await app.RunAsync($"https://localhost:{port}/");
```

Это стало возможным благодаря неявным директивам `using`, которые по умолчанию включены в проект:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
</Project>
```

Дельный совет: минимальные API не используют динамически генерируемый код, в отличие от контроллеров. Благодаря этому вы можете с помощью технологии Native AOT создавать компактные и производительные сервисы, отлично подходящие для микросервисной архитектуры и контейнеров. По возможности реализуйте свои веб-сервисы, используя минимальные API вместо контроллеров.

Если у вас есть выбор, рекомендуется использовать минимальные API вместо контроллеров.

В дальнейшем в этой главе мы подробнее рассмотрим технологию Native AOT и ее применение в минимальных API.

Маршрутизация в минимальных API

Класс `WebApplication` содержит методы, позволяющие сопоставлять маршруты с обработчиками запросов, реализованными с помощью лямбда-выражений или отдельных методов:

- `MapGet` — сопоставляет маршрут с запросом `GET` для получения сущности;
- `MapPost` — сопоставляет маршрут с запросом `POST` для создания сущности;
- `MapPut` — сопоставляет маршрут с запросом `PUT` для полного обновления сущности;
- `MapPatch` — сопоставляет маршрут с запросом `PATCH` для частичного обновления сущности;
- `MapDelete` — сопоставляет маршрут с запросом `DELETE` для удаления сущности;

- `MapMethods` — позволяет сопоставлять маршрут с любым другим HTTP-методом, например `HEAD` или `CONNECT`.

Так, можно сопоставить HTTP-запрос `GET` по относительному пути `/api/customers` с обработчиком, реализованным через лямбда-выражение или функцию, который возвращает список клиентов в формате JSON:

```
app.MapGet("api/customers", GetCustomers);
app.MapPost("api/customers", InsertCustomer);
app.MapDelete("api/customers/{id}", DeleteCustomer);
```

Иногда может потребоваться сопоставить маршрут с HTTP-запросом `CONNECT` для определенного пути, например `/api/customers`. Это можно сделать следующим образом:

```
app.MapMethods("api/customers", new[] { "CONNECT" }, () =>
  {
    // Некие действия
  });
```

Если у приложения есть несколько конечных точек, относящихся к одной группе, их можно объединить с помощью метода `MapGroup`, который появился в .NET 7. Так вы упростите код и сократите дублирование:

```
RouteGroupBuilder group = app.MapGroup("api/customers")

group.MapGet("/", GetCustomers)
  .MapGet("/{id}", GetCustomerById)
  .MapPost("/", InsertCustomer)
  .MapDelete("/{id}", DeleteCustomer);
```

В копилку знаний: подробную информацию об обработчиках маршрутов вы найдете по адресу https://learn.microsoft.com/ru-ru/aspnet/core/fundamentals/minimal-apis/route-handlers.

Настройка параметров в минимальных API

Методы обработки запросов могут принимать параметры, источники которых определяются автоматически. Хотя большинство параметров можно получить без явного указания на их источник, иногда требуется точно указать, откуда их следует извлекать. Для этого используются специальные атрибуты:

- `[FromServices]`: получает параметр из зарегистрированных в DI-контейнере сервисов;
- `[FromRoute]`: получает параметр из *сегмента маршрута*;
- `[FromQuery]`: извлекает параметр из *строки запроса*;
- `[FromBody]`: получает параметр из тела HTTP-запроса.

Так, обновление сущности в БД требует передачи контекста БД (из DI-контейнера), идентификатора (из маршрута или строки запроса) и данных сущности (из тела запроса):

```
app.MapPut("api/customers/{id}", async (
  [FromServices] NorthwindContext db,
  [FromRoute] string id, // или [FromQuery] string id,
  [FromBody] Customer customer) =>
{
  Customer? existingCustomer = await db.Customers.FindAsync(id);
  ...
});
```

Возвращаемые значения

Минимальные API могут возвращать данные в нескольких форматах, в том числе (табл. 8.1).

Таблица 8.1. Примеры возвращаемых значений минимальных API

Тип	Лямбда-выражение
Обычный текст	`() => "Hello World!"` `() => Results.Text("Hello World!")`
JSON-документ	`() => new { FirstName = "Bob", LastName = "Jones" }` `() => Results.Json(new { FirstName = "Bob", LastName = "Jones" })`

Интерфейс `IResult` с кодами ответа	`() => Results.Ok(new { FirstName = "Bob", LastName = "Jones" })`
	`() => Results.NoContent()`
	`() => Results.Redirect("new/path")` `() => Results.NotFound()`
	`() => Results.BadRequest()` `() => Results.Problem()`
	`() => Results.StatusCode(405)`

Файл	`() => Results.File("/path/filename.ext")`

Документирование сервисов на основе минимальных API

Для документирования API можно указывать тип возвращаемых данных и возможные коды ответа с помощью методов `Produces<T>()` и `Produces()`. Так можно явно обозначить, какие ответы могут быть возвращены.

- `Produces<T>(StatusCodes.Status200OK)`: в случае успешного выполнения запроса маршрут возвращает данные типа `T` и код ответа `200 OK`.
- `Produces(StatusCodes.Status404NotFound)`: если соответствие маршруту не найдено, возвращается пустой ответ с кодом ответа `404 Not Found`.

Настройка проекта веб-API в ASP.NET Core

Сначала создадим простой проект веб-API в ASP.NET Core, который впоследствии защитим, применяя различные техники: ограничение количества запросов, CORS, проверку подлинности и авторизацию.

Ниже представлена таблица методов API для этого веб-сервиса (табл. 8.2).

Таблица 8.2. Методы API, реализованные в примере

Метод	Путь	Тело запроса	Тело ответа	Код успешного выполнения
`GET`	`/`	Нет	Hello World!	`200`
`GET`	`/api/products`	Нет	Массив объектов `Product` на складе	`200`
`GET`	`/api/products/outofstock`	Нет	Массив отсутствующих на складе объектов `Product`	`200`
`GET`	`/api/products/discontinued`	Нет	Массив объектов `Product` со скидками	`200`
`GET`	`/api/products/{id}`	Нет	Объект `Product`	`200`
`GET`	`/api/products/{name}`	Нет	Массив объектов `Product` с указанным названием	`200`
`POST`	`/api/products`	Объект `Product` (без `id`)	Созданный объект `Product`	`201`
`PUT`	`/api/products/{id}`	Объект `Product`	Нет	`204`
`DELETE`	`/api/products/{id}`	Нет	Нет	`204`

Приступим к реализации.

1. В редакторе кода создайте решение `Chapter08`.
2. Создайте проект веб-API со следующими настройками:
   - выберите шаблон проекта ASP.NET Core Web API (Веб-API ASP.NET Core) / `webapi`;
   - в поле Solution name (Имя решения) укажите значение `Chapter08`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.WebApi.Service`;

- в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
- установите флажок Configure for HTTPS (Настроить для HTTPS);
- снимите флажок Enable Docker (Включить поддержку контейнера);
- установите флажок Enable OpenAPI support (Включить поддержку OpenAPI);
- снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
- снимите флажок Use controllers (Использовать контроллеры).

Чтобы создать проект веб-API с минимальными API с помощью команды dotnet new в SDK версии, предшествующей .NET 8, следует использовать ключ -minimal или --use-minimal-apis. В SDK версии .NET 8 и выше минимальные API включены по умолчанию, а для использования контроллеров необходимо добавить ключ --use-controllers или -controllers.

Внимание! В JetBrains Rider может отсутствовать возможность создания проекта веб-интерфейса на минимальных API. В таком случае рекомендую создать проект с помощью команды dotnet new, а затем добавить его в решение.

3. Добавьте ссылку на проект с контекстом БД Northwind для SQL Server, который был создан в главе 3:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.DataContext
➥ .SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
</ItemGroup>
```

Путь не должен содержать символы разрыва строки. Если в главе 3 вы не создали библиотеки классов, скачайте проекты решений из GitHub-репозитория для этой книги.

4. В файле проекта присвойте параметру `invariantGlobalization` значение `false`, а также настройте строгую обработку предупреждений:

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>false</InvariantGlobalization>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>
```

Возможность активации инвариантной глобализации командой <InvariantGlobalization>true</InvariantGlobalization> — новая функция в шаблонах веб-API ASP.NET Core в версии .NET 8. В этом случае веб-сервис становится независимым от культуры, поэтому после развертывания он функционирует одинаково в любой точке мира. Если же отключить это свойство командой <InvariantGlobalization>false</InvariantGlobalization>, веб-сервис будет настроен на культуру компьютера, на котором он размещен. Подробнее о глобализации и ее настройках можно узнать по адресу https://github.com/dotnet/runtime/blob/main/docs/design/features/globalization-invariant-mode.md.

5. В терминале выполните сборку проекта `Northwind.WebApi.Service`, чтобы все проекты библиотеки классов с моделями сущностей за пределами текущего решения были скомпилированы корректно:

```
dotnet build
```

6. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, присвоив ему порт `5081`:

```
"profiles": {
  ...
  "https": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "launchUrl": "swagger",
    "applicationUrl": "https://localhost:5081",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
```

Программа Visual Studio 2022 проанализирует данный файл настроек и автоматически запустит браузер, если параметру launchBrowser присвоено значение true. В случае с Visual Studio Code и консолью вам нужно будет вручную открыть браузер и перейти по адресу https://localhost:5081/swagger.

7. В файле `Program.cs` удалите строки, связанные с сервисом погоды, и замените их операторами импорта пространства имен и добавления класса `NorthwindContext` в настроенные сервисы:

```
using Northwind.EntityModels; // Для метода AddNorthwindContext

var builder = WebApplication.CreateBuilder(args);

// Добавление сервисов в контейнер
// Подробнее о настройке Swagger/OpenAPI можно узнать на сайте
// https://aka.ms/ aspnetcore/swashbuckle.
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

```
builder.Services.AddNorthwindContext();

var app = builder.Build();

// Настройка конвейера HTTP-запросов
if (app.Environment.IsDevelopment())
{
  app.UseSwagger();
  app.UseSwaggerUI();
}

app.UseHttpsRedirection();
```

8. Создайте файл класса `WebApplication.Extensions.cs`.

Дельный совет: чтобы не загромождать файл Program.cs множеством строк кода, определите методы расширения для типов, которые настраиваются в минимальных API, таких как WebApplication и IServiceCollection.

9. В файле `WebApplication.Extensions.cs` импортируйте необходимые пространства имен для работы с результатами HTTP-запросов, привязки параметров к механизму внедрения зависимостей и работы с моделями сущностей. Затем определите метод расширения для класса `WebApplication`, чтобы настроить ответы на все HTTP-запросы `GET`, как указано в таблице API:

```
using Microsoft.AspNetCore.Http.HttpResults; // Для класса Results
using Microsoft.AspNetCore.Mvc; // Для параметра [FromServices] и т. д.
using Northwind.EntityModels; // Для классов NorthwindContext, Product

namespace Packt.Extensions;

public static class WebApplicationExtensions
{
  public static WebApplication MapGets(this WebApplication app,
    int pageSize = 10)
  {
    app.MapGet("/", () => "Hello World!")
      .ExcludeFromDescription();

    app.MapGet("api/products", (
      [FromServices] NorthwindContext db,
      [FromQuery] int? page) =>
      db.Products
        .Where(p => p.UnitsInStock > 0 && !p.Discontinued)
        .OrderBy(product => product.ProductId)
        .Skip(((page ?? 1) - 1) * pageSize)
        .Take(pageSize)
      )
      .WithName("GetProducts")
      .WithOpenApi(operation =>
      {
```

```
        operation.Description =
          "Get products with UnitsInStock > 0 and Discontinued = false.";
        operation.Summary =
          "Get in-stock products that are not discontinued.";
        return operation;
      })
      .Produces<Product[]>(StatusCodes.Status200OK);

    app.MapGet("api/products/outofstock",
      ([FromServices] NorthwindContext db) => db.Products
        .Where(p => p.UnitsInStock == 0 && !p.Discontinued)
      )
      .WithName("GetProductsOutOfStock")
      .WithOpenApi()
      .Produces<Product[]>(StatusCodes.Status200OK);

    app.MapGet("api/products/discontinued",
      ([FromServices] NorthwindContext db) =>
        db.Products.Where(product => product.Discontinued)
      )
      .WithName("GetProductsDiscontinued")
      .WithOpenApi()
      .Produces<Product[]>(StatusCodes.Status200OK);

    app.MapGet("api/products/{id:int}",
      async Task<Results<Ok<Product>, NotFound>> (
      [FromServices] NorthwindContext db,
      [FromRoute] int id) =>
        await db.Products.FindAsync(id) is Product product ?
          TypedResults.Ok(product) : TypedResults.NotFound())
      .WithName("GetProductById")
      .WithOpenApi()
      .Produces<Product>(StatusCodes.Status200OK)
      .Produces(StatusCodes.Status404NotFound);

    app.MapGet("api/products/{name}", (
      [FromServices] NorthwindContext db,
      [FromRoute] string name) =>
        db.Products.Where(p => p.ProductName.Contains(name)))
      .WithName("GetProductsByName")
      .WithOpenApi()
      .Produces<Product[]>(StatusCodes.Status200OK);

    return app;
  }
}
```

10. В файле `WebApplication.Extensions.cs` определите метод расширения для класса `WebApplication`, чтобы настроить ответ на HTTP-запрос `POST`, задокументированный в таблице API:

```
public static WebApplication MapPosts(this WebApplication app)
{
  app.MapPost("api/products", async ([FromBody] Product product,
    [FromServices] NorthwindContext db) =>
  {
    db.Products.Add(product);
    await db.SaveChangesAsync();
    return Results.Created($"api/products/{product.ProductId}", product);
  }).WithOpenApi()
    .Produces<Product>(StatusCodes.Status201Created);

  return app;
}
```

11. Аналогичным образом настройте ответ на HTTP-запрос PUT:

```
public static WebApplication MapPuts(this WebApplication app)
{
  app.MapPut("api/products/{id:int}", async (
    [FromRoute] int id,
    [FromBody] Product product,
    [FromServices] NorthwindContext db) =>
  {
    Product? foundProduct = await db.Products.FindAsync(id);

    if (foundProduct is null) return Results.NotFound();

    foundProduct.ProductName = product.ProductName;
    foundProduct.CategoryId = product.CategoryId;
    foundProduct.SupplierId = product.SupplierId;
    foundProduct.QuantityPerUnit = product.QuantityPerUnit;
    foundProduct.UnitsInStock = product.UnitsInStock;
    foundProduct.UnitsOnOrder = product.UnitsOnOrder;
    foundProduct.ReorderLevel = product.ReorderLevel;
    foundProduct.UnitPrice = product.UnitPrice;
    foundProduct.Discontinued = product.Discontinued;

    await db.SaveChangesAsync();

    return Results.NoContent();
  }).WithOpenApi()
    .Produces(StatusCodes.Status404NotFound)
    .Produces(StatusCodes.Status204NoContent);

  return app;
}
```

12. И на HTTP-запрос DELETE:

```
public static WebApplication MapDeletes(this WebApplication app)
{
  app.MapDelete("api/products/{id:int}", async (
```

```
    [FromRoute] int id,
    [FromServices] NorthwindContext db) =>
  {
    if (await db.Products.FindAsync(id) is Product product)
    {
      db.Products.Remove(product);
      await db.SaveChangesAsync();
      return Results.NoContent();
    }
    return Results.NotFound();
  }).WithOpenApi()
    .Produces(StatusCodes.Status404NotFound)
    .Produces(StatusCodes.Status204NoContent);

  return app;
}
```

13. В файле `Program.cs` импортируйте пространство имен, добавляющее поддержку только что определенных методов расширения:

```
using Packt.Extensions; // Для метода MapGets и т. д.
```

14. В файле `Program.cs` перед вызовом метода `app.Run()` вызовите все пользовательские методы расширения для отображения запросов `GET`, `POST`, `PUT` и `DELETE`. При этом можно переопределить стандартный размер страницы в десять сущностей при запросе всех товаров:

```
app.MapGets() // Размер страницы по умолчанию: 10
  .MapPosts()
  .MapPuts()
  .MapDeletes();
```

15. Убедитесь, что в конец файла `Program.cs` добавлена последняя инструкция для запуска веб-приложения:

```
app.Run();
```

Тестирование веб-сервисов с помощью Swagger

Теперь, когда веб-сервис настроен, можно запустить его, просмотреть документацию с помощью Swagger и вручную выполнить базовую проверку.

1. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
2. Запустите веб-сервис, выбрав профиль https.
   - В программе Visual Studio 2022 выберите профиль https в раскрывающемся списке, а затем выберите команду меню Debug ▸ Start Without Debugging (Отладка ▸ Запустить без отладки) или нажмите сочетание клавиш Ctrl+F5. Браузер должен автоматически открыть веб-страницу с документацией Swagger.

- В Visual Studio Code введите команду `dotnet run --launch- profile https`, вручную запустите браузер и перейдите по адресу https://localhost:5081/swagger.

Если в операционной системе Windows появится соответствующий запрос, разрешите доступ к локальному веб-сервису через брандмауэр Windows Defender.

3. В консоли/терминале просмотрите информацию о своем веб-сервисе:

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5081
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: C:\apps-services-net8\Chapter08\
      Northwind.WebApi.Service
```

4. В браузере откройте документацию Swagger (рис. 8.1).

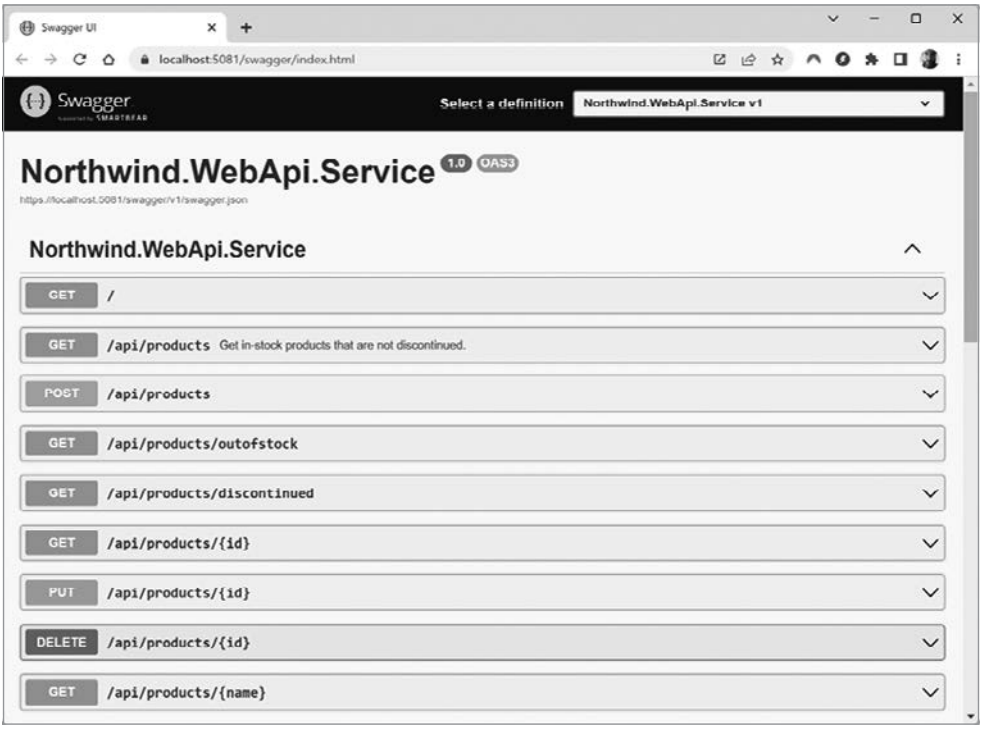


Рис. 8.1. Документация Swagger для сервиса с веб-API Northwind

5. Щелкните кнопкой мыши на строке GET /api/products, чтобы развернуть этот раздел.
6. Нажмите кнопку Try it out (Попробовать), обратите внимание на необязательный параметр строки запроса page, а затем нажмите кнопку Execute (Выполнить).
7. В ответе указаны первые десять товаров без скидки из числа доступных на складе: 1, 2, 3, 4, 6, 7, 8, 10, 11 и 12.
8. Присвойте параметру `page` значение `3` и нажмите кнопку Execute (Выполнить).
9. В ответ отобразится третья страница с десятью товарами без скидки, доступными на складе: 25, 26, 27, 30, 32, 33, 34, 35, 36 и 37.
10. Щелкните кнопкой мыши на строке GET /api/products, чтобы свернуть этот раздел.
11. Попробуйте выполнить запрос GET /api/products/outofstock. Он должен вернуть один товар без скидки — 31 Gorgonzola Telino — с нулевым количеством на складе.
12. Попробуйте выполнить запрос GET /api/products/discontinued. Он вернет восемь товаров со значением `true` свойства `Discontinued`: 5, 9, 17, 24, 28, 29, 42 и 53.
13. Щелкните кнопкой мыши на строке GET /api/products/{id}, чтобы развернуть этот раздел.
14. Нажмите кнопку Try it out (Попробовать). Присвойте параметру `id` значение `77`, а затем нажмите кнопку Execute (Выполнить). В ответе должен быть указан товар Original Frankfurter grüne Soße, как показано в следующем JSON-документе:

```
{
  "productId": 77,
  "productName": "Original Frankfurter grüne Soße",
  "supplierId": 12,
  "categoryId": 2,
  "quantityPerUnit": "12 boxes",
  "unitPrice": 13,
  "unitsInStock": 32,
  "unitsOnOrder": 0,
  "reorderLevel": 15,
  "discontinued": false,
  "category": null,
  "supplier": null,
  "orderDetails": []
}
```

15. Щелкните на строке GET /api/products/{id}, чтобы свернуть этот раздел.
16. Щелкните на строке GET /api/products/{name}, чтобы развернуть этот раздел.
17. Нажмите кнопку Try it out (Попробовать), присвойте параметру `name` значение `man`, а затем нажмите кнопку Execute (Выполнить). В ответе появятся товары с названиями Queso Manchego La Pastora и Manjimup Dried Apples.
18. Не завершайте работу веб-сервиса.

Тестирование веб-сервисов в редакторе кода

Пользовательский интерфейс Swagger для тестирования веб-сервисов может оказаться неудобным. Гораздо эффективнее воспользоваться панелью Endpoints Explorer (Обозреватель конечных точек) и расширением REST Client для Visual Studio Code или инструментами для работы с файлами `.http` в программе Visual Studio 2022 версии 17.6 и новее.

В копилку знаний: дополнительные сведения о программе Visual Studio 2022 и встроенном в нее редакторе HTTP опубликованы по адресу https://learn.microsoft.com/ru-ru/aspnet/core/test/http-files.

В JetBrains Rider есть схожая панель инструментов — Endpoints (Конечные точки).

О возможностях работы с HTTP-файлами в программе JetBrains Rider в Rider можно прочитать по адресу https://www.jetbrains.com/help/rider/Http_client_in__product__code_editor.html. Этот инструмент несколько отличается от других редакторов, особенно в части обработки переменных, как продемонстрировано по адресу https://www.jetbrains.com/help/rider/Exploring_HTTP_Syntax.html#example-working-with-environment-files. Для этого раздела я рекомендую использовать программу Visual Studio Code с расширением REST Client.

Далее мы рассмотрим, как автоматизировать тесты запросов и использовать различные инструменты редакторов для работы с веб-API.

1. Убедитесь, что у вас установлены необходимые инструменты для тестирования (на выбор).
   - Программа Visual Studio 2022 версии 17.6 или более поздней. Возможность сканирования проекта на наличие конечных точек веб-API доступна с помощью панели Endpoints Explorer (Обозреватель конечных точек). Для доступа к ней выберите команду меню View ▸ Other Windows ▸ Endpoints Explorer (Вид ▸ Другие окна ▸ Обозреватель конечных точек).
   - В Visual Studio Code должно быть установлено расширение REST Client от Хуачао Мао (`humao.rest-client`).
2. В выбранном редакторе кода запустите проект `Northwind.WebApi.Service` с профилем `https`, если не сделали это ранее.
3. В папке `apps-services-net8` создайте каталог `HttpRequests`, если она не существует.
4. В папке `HttpRequests` создайте файл `webapi-get-products.http`. Объявите в нем переменную для хранения базового адреса конечной точки веб-API

для продуктов, а также добавьте запрос на получение первой страницы с десятью товарами:

```
### Настройка переменной для базового адреса веб-сервиса
@base_address = https://localhost:5081/api/products/

### Генерация первой страницы с 10 товарами без скидки, которые есть на складе
GET {{base_address}}
```

Дельный совет: в REST Client не требуется явное указание метода GET, так как он по умолчанию применяется к каждому запросу. Однако в Visual Studio необходимо задать метод явно (на момент написания книги). Рекомендуется указывать метод для всех инструментов.

5. Нажмите кнопку Send Request (Отправить запрос) и убедитесь, что ответ аналогичен тому, что возвращает Swagger. Это должен быть JSON-документ, содержащий первые десять товаров без скидки, доступные на складе. Снимок экрана программы Visual Studio 2022 показан на рис. 8.2, а Visual Studio Code — на рис. 8.3.

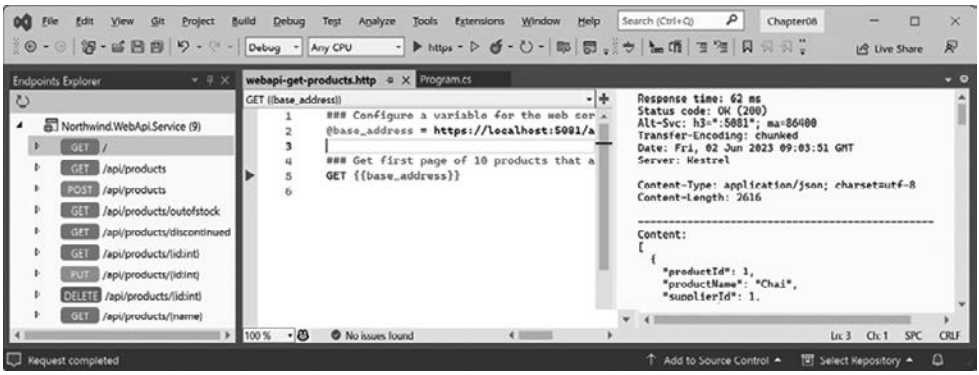


Рис. 8.2. Visual Studio 2022: извлечение товаров через веб-API

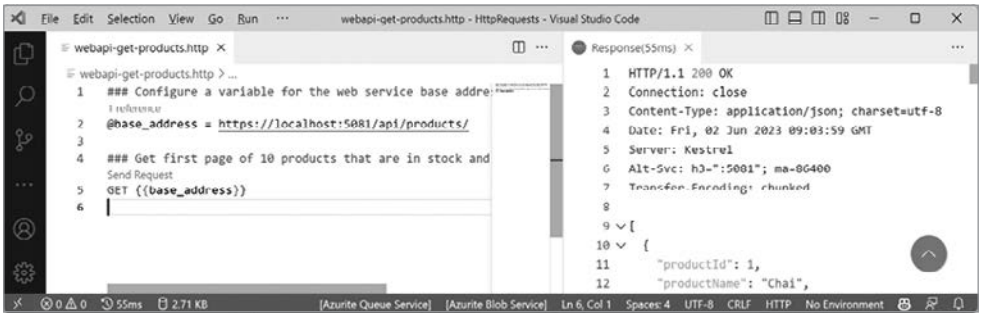


Рис. 8.3. REST-клиент Visual Studio Code, извлекающий товары из сервиса веб-API

6. В файле `webapi-get-products.http` добавьте несколько запросов, разделенных символами `###`, чтобы протестировать другие действия:

```
### Генерация третьей страницы из 10 товаров без скидки,
### которые есть на складе
GET {{base_address}}?page=3

### Выборка товаров без скидки, которые отсутствуют на складе
GET {{base_address}}outofstock

### Выборка товаров со скидкой
GET {{base_address}}discontinued

### Извлечение товара под номером 77
GET {{base_address}}77

### Выборка товаров, содержащих слово "man"
GET {{base_address}}man
```

Чтобы выполнить HTTP-запрос в программе Visual Studio 2022, щелкните правой кнопкой мыши на зеленой кнопке запуска (с треугольником) и выберите в контекстном меню пункт Send Request (Отправить запрос) или нажмите сочетание клавиш Ctrl+Alt+S. В Visual Studio Code нажмите кнопку Send Request (Отправить запрос) около каждого запроса. Вы также можете выбрать команду меню View ▸ Command Palette (Вид ▸ Палитра команд) и выполнить пункт Rest Client: Send Request (Rest Client: Отправить запрос) либо нажать сочетание клавиш Ctrl+Alt+R (в Windows).

7. В папке `HttpRequests` создайте файл `webapi-insert-product.http` и добавьте в него `POST`-запрос на добавление нового товара:

```
POST https://localhost:5081/api/products/
Content-Type: application/json

{
  "productName": "Harry's Hamburgers",
  "supplierId": 7,
  "categoryId": 6,
  "quantityPerUnit": "6 per box",
  "unitPrice": 24.99,
  "unitsInStock": 0,
  "unitsOnOrder": 20,
  "reorderLevel": 10,
  "discontinued": false
}
```

8. Нажмите кнопку Send Request (Отправить запрос) и убедитесь, что товар был успешно добавлен. Об этом говорит код ответа `201`, а в строке `Location` указан идентификатор товара (рис. 8.4).

Рис. 8.4. REST Client добавил новый товар путем вызова веб-API

Изначально в базе данных Northwind содержится 77 товаров. Следующий идентификатор товара будет равен 78. Однако фактический идентификатор, присвоенный автоматически, может отличаться в зависимости от того, добавлялись ли ранее другие товары, и в таком случае номер может быть больше.

9. Для обновления товара с идентификатором 78 (или иным, присвоенным позиции Harry's Hamburgers) в папке `HttpRequests` создайте файл `webapi-update-product.http` с показанным ниже PUT-запросом:

```
PUT https://localhost:5081/api/products/78
Content-Type: application/json

{
  "productName": "Harry's Hamburgers",
  "supplierId": 7,
  "categoryId": 6,
  "quantityPerUnit": "12 per box",
  "unitPrice": 44.99,
  "unitsInStock": 50,
  "unitsOnOrder": 20,
  "reorderLevel": 10,
  "discontinued": false
}
```

10. Отправьте запрос и убедитесь, что товар был обновлен. На это укажет код ответа `204`, означающий успешное обновление.
11. Подтвердите обновление, выполнив `GET`-запрос по идентификатору товара.
12. В папке `HttpRequests` создайте файл `webapi-delete-product.http` и добавьте в него `DELETE`-запрос нового товара:

```
DELETE https://localhost:5081/api/products/78
```

13. Отправьте запрос и убедитесь, что товар был удален (рис. 8.5).

Рис. 8.5. Удаление товара с помощью веб-API

14. Отправьте запрос еще раз и убедитесь, что теперь ответ содержит код ответа `404`, поскольку товар был удален.
15. Завершите работу веб-сервера.

Исключение путей из документации OpenAPI

Иногда возникает необходимость, чтобы путь работал, но не отображался в документации Swagger. Рассмотрим, как исключить корневой путь, который возвращает текст Hello World!, из документации.

1. В файле `WebApplication.Extensions.cs` для корневого пути, который возвращает фразу Hello World!, добавьте метод `ExcludeFromDescription()` для исключения этого пути из документации OpenAPI:

```
app.MapGet("/", () => "Hello World!")
  .ExcludeFromDescription();
```

2. Запустите проект `Northwind.WebApi.Service` с профилем `https`, не используя отладку. Обратите внимание, что этот путь больше не будет отображаться в документации Swagger.

Теперь у нас есть работающий веб-сервис, построенный на минимальных API ASP.NET Core. Далее сосредоточимся на защите этого сервиса от возможных атак.

Создание шаблонного кода для минимальных API в Visual Studio 2022

Важно уметь реализовывать сервис на минимальных API с нуля, чтобы понимать его устройство. Однако, как только вы освоите этот процесс, можно автоматизировать его, используя шаблоны кода. Это особенно полезно, если вы создаете веб-интерфейс, который работает поверх модели сущностей EF Core.

В программе Visual Studio 2022 доступен шаблон проекта API with read/write endpoints, using Entity Framework:

- класса модели сущности, например `Customer`;
- класса конечных точек, который будет содержать все методы сопоставления;
- класса, производного от `DbContext`, например `NorthwindContext`;
- провайдера БД, например SQLite или SQL Server.

В копилку знаний: подробнее об этом шаблоне проекта можно узнать по адресу https://devblogs.microsoft.com/visualstudio/web-api-development-in-visual-studio-2022/#scaffolding-in-visual-studio/.

Управление междоменными запросами с помощью механизма CORS

В современных браузерах используется многодокументный интерфейс с вкладками, благодаря чему пользователи могут эффективно посещать несколько сайтов одновременно. Однако если код, выполняющийся на одной вкладке, получает доступ к ресурсам на другой вкладке, возникает уязвимость для атак со стороны злоумышленников.

Для защиты от таких ситуаций во всех браузерах реализован механизм безопасности, известный как *правило ограничения доменов (Same-Origin Policy)*. Это правило разрешает выполнение запросов только из одного и того же источника. Например, если JavaScript-код загружается с того же домена, что и веб-сервис или элемент `<iframe>`, этот код может взаимодействовать с сервисом и получить данные из элемента `<iframe>`. Но если запрос исходит из другого источника, он не будет выполнен. Что же считается «одним и тем же источником»?

Источник определяется тремя ключевыми компонентами.

- *Схемой*, или протоколом, например `http` или `https`.
- *Портом*, например 801 или 5081. Для `http` по умолчанию используется порт 80, а для `http` — 443.
- *Хостом/доменом/поддоменом*, например `www.example.com`, `www.example.net`, `example.com`.

Если исходный адрес запроса — это `https://www.example.com/about-us/`, то перечисленные ниже источники будут считаться различными.

- *Другая схема*: `http://www.example.com/about-us/`.
- *Другой домен*: `https://www.example.co.uk/about-us/`.
- *Другой поддомен*: `https://careers.example.com/about-us/`.
- *Другой порт*: `https://www.example.com:444/about-us/`.

Важно заметить, что браузер автоматически устанавливает заголовок `Origin` при выполнении запроса и это невозможно изменить.

Внимание! Правило ограничения доменов не применяется к запросам, поступающим не из браузера. В таких случаях разработчик может вручную изменить заголовок Origin. Например, если вы создаете консольное приложение или проект на ASP.NET Core, в котором для отправки запросов используются классы .NET, такие как HttpClient, это правило не применяется, если заголовок Origin не был явно установлен.

Теперь рассмотрим примеры вызова веб-сервиса с веб-страницы, которая использует другой источник, а также из приложения на .NET.

Настройка HTTP-журналирования

Подключим на веб-сервисе HTTP-журналирование и настроим его так, чтобы отображался источник запросов.

1. В файле `Northwind.WebApi.Service` создайте файл класса `IServiceCollection.Extensions.cs`.
2. В созданном файле импортируйте пространство имен для работы с HTTP-заголовками, а также определите метод расширения для интерфейса `IServiceCollection`, который добавит журналирование HTTP, включая заголовок `Origin` и все поля, в том числе тело ответа:

```
using Microsoft.AspNetCore.HttpLogging; // Для флага HttpLoggingFields

namespace Packt.Extensions;

public static class IServiceCollectionExtensions
{
  public static IServiceCollection AddCustomHttpLogging(
    this IServiceCollection services)
  {
    services.AddHttpLogging(options =>
    {
      // Добавьте заголовок Origin, чтобы он не был отредактирован
      options.RequestHeaders.Add("Origin");

      // По умолчанию тело ответа не включается в журналирование
      options.LoggingFields = HttpLoggingFields.All;
    });

    return services;
  }
}
```

3. В файле `Program.cs` перед вызовом метода `builder.Build()` добавьте строку для подключения пользовательского HTTP-журналирования:

```
builder.Services.AddCustomHttpLogging();
```

4. В файле `Program.cs` после вызова метода `UseHttpsRedirection()` добавьте инструкцию для включения HTTP-журналирования:

```
app.UseHttpLogging();
```

5. В файле `appsettings.Development.json` укажите уровень HTTP-журналирования `Information`, применив следующую конфигурацию:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",

      "Microsoft.AspNetCore.HttpLogging": "Information"
    }
  }
}
```

На уровне `Information` фиксируются важные сведения о работе системы, но не включаются подробности, характерные для уровней `Debug` или `Trace`.

Дельный совет: спецификация JSON не допускает комментариев, а JSONC (JSON с комментариями) позволяет их использовать. В файлах JSONC можно добавлять комментарии в стиле JavaScript (// и /* */). Подробнее об этом можно узнать по адресу https://code.visualstudio.com/docs/languages/json#_json-with-comments. Если используемый вами редактор не поддерживает формат JSONC, удалите комментарии.

Создание веб-страницы с клиентским JavaScript

Теперь создадим клиентскую веб-страницу, которая выполнит запрос к веб-сервису с другого порта.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта ASP.NET Core Web App (Model-View-Controller) (Веб-приложение ASP.NET Core (модель-представление-контроллер)) / `mvc`;
   - в поле Solution name (Имя решения) укажите значение `Chapter08`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.WebApi.Client.Mvc`;

- другие параметры Visual Studio 2022:
  - в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
  - установите флажок Configure for HTTPS (Настроить для HTTPS);
  - снимите флажок Enable Docker (Включить поддержку контейнера);
  - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
  - в Visual Studio 2022 присвойте параметру Startup Project (Запускаемый проект) значение Current selection (Текущий вариант).

2. В папке `Properties` проекта `Northwind.WebApi.Client.Mvc` откройте файл `launchSettings.json` и измените параметр `applicationUrl` для профиля `https`, указав в нем порт `5082`:

```
"applicationUrl": "https://localhost:5082",
```

3. В файле созданного проекта добавьте строгую обработку предупреждений.
4. В файле `Index.cshtml` (папка `Views/Home`) замените существующую разметку на новую. Она должна содержать:
   - поле для ввода текста;
   - кнопку для запроса;
   - JavaScript-код, который отправит на веб-сервис запрос для поиска товаров по фрагменту названия:

```
@{
  ViewData["Title"] = "Products using JavaScript";
}

<div class="text-center">
  <h1 class="display-4">@ViewData["Title"]</h1>
  <div>
    Go to <a href="/home/products">Products using .NET</a>
  </div>
  <div>
    <input id="productName" placeholder="Enter part of a product name" />
    <input id="getProductsButton" type="button" value="Get Products" />
  </div>
  <div>
    <table id="productsTable" class="table">
      <thead>
        <tr>
          <th scope="col">Product Name</th>
        </tr>
      </thead>
      <tbody id="tableBody">
```

```
      </tbody>
    </table>
  </div>
  <script>
    var baseaddress = "https://localhost:5081/";

    function xhr_load() {
        console.log(this.responseText);

        var products = JSON.parse(this.responseText);

        var out = "";
        var i;

        for (i = 0; i < products.length; i++) {
            out += '<tr><td><a href="' + baseaddress + 'api/products/' +
                products[i].productId + '">' +
                products[i].productName + '</a></td></tr>';
        }
        document.getElementById("tableBody").innerHTML = out;
    }

    function getProductsButton_click() {
        xhr.open("GET", baseaddress + "api/products/" +
          document.getElementById("productName").value);

        xhr.send();
    }

    document.getElementById("getProductsButton")
      .addEventListener("click", getProductsButton_click);

    var xhr = new XMLHttpRequest();
    xhr.addEventListener("load", xhr_load);
  </script>
</div>
```

5. Запустите `Northwind.WebApi.Service` с профилем `https` без отладки.
6. Запустите `Northwind.WebApi.Client.Mvc` также с `https` без отладки.

В Visual Studio Code браузер не запустится автоматически. Откройте Google Chrome вручную и перейдите по адресу https://localhost:5082.

7. В Chrome откройте панель Developer Tools (Инструменты разработчика) и перейдите на вкладку Console (Консоль).

8. На веб-странице Products using JavaScript в текстовом поле введите слово `man`, нажмите кнопку Get Products и посмотрите, какая ошибка появится в консоли (показано в выводе ниже и на рис. 8.6):

```
Access to XMLHttpRequest at 'https://localhost:5081/api/products/man' from origin
'https://localhost:5082' has been blocked by CORS policy:
No 'Access-Control-Allow-Origin' header is present on the requested resource.
GET https://localhost:5081/api/products/man net::ERR_FAILED 200
```

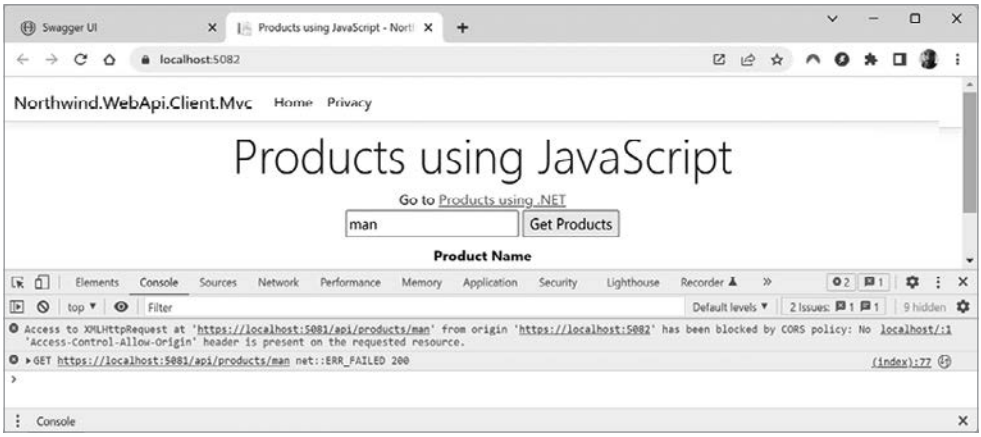


Рис. 8.6. Ошибка CORS на вкладке Console панели Developer Tools

9. В терминале найдите HTTP-журнал запроса для проекта `Northwind.WebApi.Service`. Обратите внимание, что адреса в заголовках `Host` и `Origin` содержат разные порты, поэтому браузер воспринимает их как разные источники:

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/2
      Method: GET
      Scheme: https
      PathBase:
      Path: /api/products/man
      Accept: */*
      Host: localhost:5081
      User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/113.0.0.0 Safari/537.36
      Accept-Encoding: gzip, deflate, br
      Accept-Language: en-US,en;q=0.9,sv;q=0.8
```

```
      Origin: https://localhost:5082
      Referer: [Redacted]
      sec-ch-ua: [Redacted]
      sec-ch-ua-mobile: [Redacted]
      sec-ch-ua-platform: [Redacted]
      sec-fetch-site: [Redacted]
      sec-fetch-mode: [Redacted]
      sec-fetch-dest: [Redacted]
```

10. Веб-сервис обработает запрос, обратится к базе данных и отправит ответ в формате JSON:

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
      Content-Type: application/json; charset=utf-8
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[4]
      ResponseBody: [{"productId":12,"productName":
      "Queso Manchego La Pastora","supplierId":5,"categoryId":4,
      "quantityPerUnit":"10 - 500 g pkgs.","unitPrice":38.0000,
      "unitsInStock":86,"unitsOnOrder":0,"reorderLevel":0,
      "discontinued":false,"category":null,"supplier":null,
      "orderDetails":[]},
      {"productId":51,"productName":"Manjimup Dried Apples",
      "supplierId":24,"categoryId":7,
      "quantityPerUnit":"50 - 300 g pkgs.","unitPrice":53.0000,
      "unitsInStock":20,"unitsOnOrder":0,"reorderLevel":10,
      "discontinued":false,"category":null,"supplier":null,
      "orderDetails":[]}]
```

Браузер получил JSON-ответ, но из-за политики безопасности CORS он заблокировал его для JavaScript. То есть сам веб-сервис не защищен CORS, но браузер не позволяет JavaScript-коду использовать данные из ответа.

11. Закройте браузер (браузеры) и завершите работу веб-сервера.

Создание .NET-клиента

Далее создадим для веб-сервиса .NET-клиент. Он поможет убедиться, что правило источника не применяется к программам, отличным от браузеров.

1. В файле проекта `Northwind.WebApi.Client.Mvc` добавьте ссылку на проект моделей сущностей, чтобы использовать класс `Product`:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.EntityModels
    .SqlServer\Northwind.Common.EntityModels.SqlServer.csproj" />
</ItemGroup>
```

2. Выполните сборку проекта `Northwind.WebApi.Client.Mvc` в терминале, выполнив команду `dotnet build`.
3. В файле `Program.cs` проекта `Northwind.WebApi.Client.Mvc` импортируйте пространство имен для работы с HTTP-заголовками:

```
using System.Net.Http.Headers; // Для класса MediaTypeWithQualityHeaderValue
```

4. В файле `Program.cs` перед вызовом метода `builder.Build()` добавьте выражения для настройки фабрики HTTP-клиентов, которая будет вызывать веб-сервис:

```
builder.Services.AddHttpClient(name: "Northwind.WebApi.Service",
  configureClient: options =>
  {
    options.BaseAddress = new("https://localhost:5081/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
        "application/json", 1.0));
  });
```

5. В файле `HomeController.cs` (папка `Controllers`) импортируйте пространство имен моделей сущностей:

```
using Northwind.EntityModels; // Для класса Product
```

6. В том же файле добавьте приватное поле, доступное только для чтения, предназначенное для хранения зарегистрированной фабрики HTTP-клиентов:

```
private readonly ILogger<HomeController> _logger;
private readonly IHttpClientFactory _httpClientFactory;

public HomeController(ILogger<HomeController> logger,
  IHttpClientFactory httpClientFactory)
{
  _logger = logger;
  _httpClientFactory = httpClientFactory;
}
```

7. В том же файле добавьте асинхронный метод `Products`, который с помощью фабрики HTTP-клиентов будет запрашивать у веб-сервиса товары, содержащие в названии введенное в текстовом поле имя (необязательный параметр `name`):

```
[Route("home/products/{name?}")]
public async Task<IActionResult> Products(string? name)
{
  HttpClient client = _httpClientFactory.CreateClient(
    name: "Northwind.WebApi.Service");

  HttpRequestMessage request = new(
    method: HttpMethod.Get, requestUri: $"api/products/{name}");

  HttpResponseMessage response = await client.SendAsync(request);
```

```
  IEnumerable<Product>? model = await response.Content
    .ReadFromJsonAsync<IEnumerable<Product>>();

  ViewData["baseaddress"] = client.BaseAddress;

  return View(model);
}
```

8. В папке `Views/Home` создайте файл `Products.cshtml`.
   - В Visual Studio 2022 выберите шаблон Razor View — Empty.
   - В JetBrains Rider выберите шаблон Razor MVC View.
9. Измените содержимое файла `Products.cshtml` так, чтобы программа выводила таблицу с товарами, содержащими введенное в поле поиска название:

```
@using Northwind.EntityModels
@model IEnumerable<Product>?
@{
  ViewData["Title"] = "Products using .NET";
}
<div class="text-center">
  <h1 class="display-4">@ViewData["Title"]</h1>
  <div>
    Go to <a href="/">Products using JavaScript</a>
  </div>
  <form action="/home/products">
    <input name="name" placeholder="Enter part of a product name" />
    <input type="submit" value="Get Products" />
  </form>
  <div>
    <table class="table">
      <thead>
        <tr>
          <th scope="col">Product Name</th>
        </tr>
      </thead>
      <tbody>
        @if (Model is not null)
        {
          @foreach (Product p in Model)
          {
            <tr><td><a href="@(ViewData["baseaddress"])api/products/
            ➥ @p.ProductId">@p.ProductName</a></td></tr>
          }
        }
      </tbody>
    </table>
  </div>
</div>
```

10. Запустите проект `Northwind.WebApi.Service` в режиме `https` без отладки.
11. Запустите проект `Northwind.WebApi.Client.Mvc` также в режиме `https` без отладки.
12. На главной странице щелкните на ссылке Products using .NET и обратите внимание, что в таблице отображаются первые десять товаров без скидки, имеющиеся на складе: от Chai до Queso Manchego La Pastora.
13. В текстовом поле введите слово `man` и нажмите кнопку Get Products. В таблице появятся два соответствующих товара (рис. 8.7).

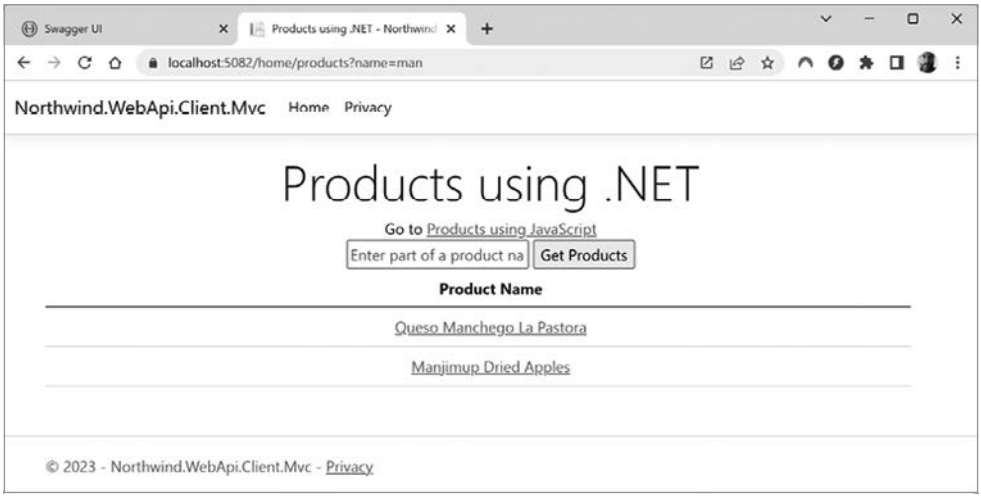


Рис. 8.7. Запрос двух товаров через веб-сервис на платформе .NET

Поскольку запрос отправляется HTTP-клиентом .NET, правило источника не применяется. В журналах указано, что порты у Host и Origin разные, но это не влияет на работу клиента.

14. Выберите один из товаров, чтобы отправить прямой запрос к веб-сервису и увидеть ответ, как показано ниже:

```
{"productId":12,"productName":"Queso Manchego La Pastora",
"supplierId":5,"categoryId":4, "quantityPerUnit":"10 - 500 g
pkgs.","unitPrice":38.0000, "unitsInStock":86,"unitsOnOrder":0,
"reorderLevel":0, "discontinued":false,"category":null,"supplier":
null,"orderDetails":[]}
```

15. Закройте браузер и завершите работу веб-сервера.

Механизм CORS

Совместное использование ресурсов между разными источниками (Cross-Origin Resource Sharing, CORS) — это функциональная возможность, основанная на HTTP-заголовках. Она позволяет браузеру отключить правило ограничения домена в определенных сценариях. HTTP-заголовки указывают, какие источники допускаются, помимо основного.

Добавим поддержку CORS на веб-сервисе, чтобы он мог отправлять дополнительные заголовки, разрешающие доступ к ресурсам из других источников.

1. В файле `WebApplication.Extensions.cs` проекта `Northwind.WebApi.Service` добавьте метод расширения для настройки на веб-сервисе механизма CORS:

```
public static IServiceCollection AddCustomCors(
  this IServiceCollection services)
{
  services.AddCors(options =>
  {
    options.AddPolicy(name: "Northwind.Mvc.Policy",
      policy =>
      {
        policy.WithOrigins("https://localhost:5082");
      });
  });
  return services;
}
```

2. В файле `Program.cs` после создания конструктора (`builder`) вызовите метод расширения, чтобы включить CORS:

```
builder.Services.AddCustomCors();
```

3. В файле `Program.cs` после вызова метода `UseHttpLogging` перед сопоставлением `GET`-запросов добавьте код для применения политики CORS:

```
app.UseCors(policyName: "Northwind.Mvc.Policy");
```

4. Запустите проект `Northwind.WebApi.Service` в режиме `https` без отладки.
5. Запустите проект `Northwind.WebApi.Client.Mvc`, также выбрав профиль `https` и режим без отладки.
6. В программе Chrome откройте панель Developer Tools (Инструменты разработчика) и перейдите на вкладку Console (Консоль).
7. На главной веб-странице приложения в текстовом поле введите слово `man` и нажмите кнопку Get Products. Убедитесь, что в консоли отображается JSON-документ, возвращенный веб-сервисом, а таблица содержит два товара (рис. 8.8).

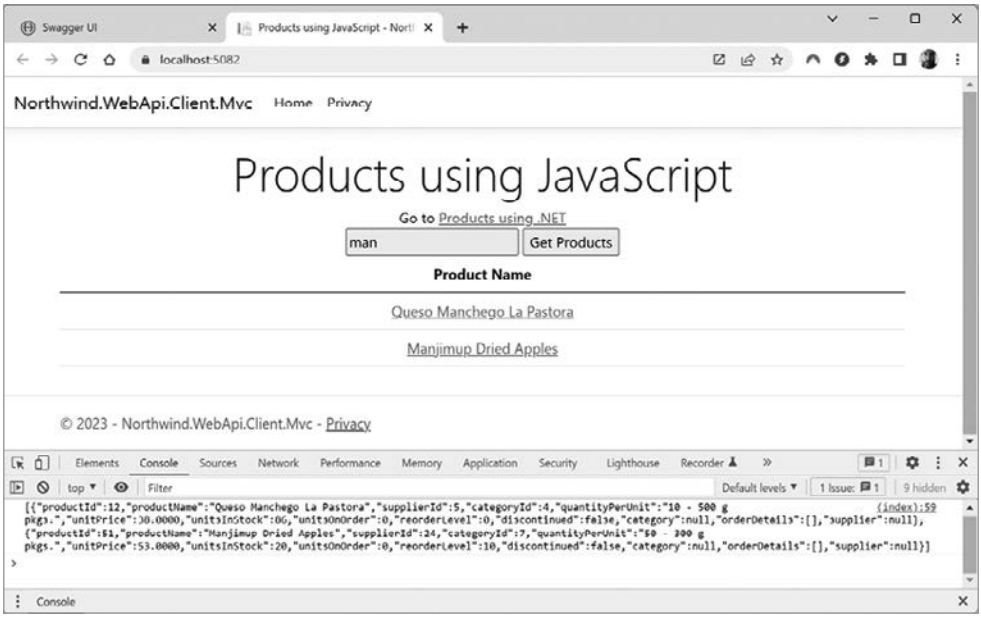


Рис. 8.8. Успешный запрос из разных источников к веб-сервису с помощью JavaScript

8. Закройте браузер и завершите работу веб-сервера.

Включение CORS для определенных конечных точек

В предыдущем примере одна и та же политика CORS применялась ко всему веб-сервису. Однако в некоторых случаях может потребоваться более тонкий контроль определенных конечных точек.

1. В файле `Program.cs` проекта `Northwind.WebApi.Service` измените вызов метода `UseCors`, опустив название политики:

```
// app.UseCors(policyName: "Northwind.Mvc.Policy");

// Без именованной политики промежуточное ПО добавлено, но неактивно
app.UseCors();
```

2. В файле `WebApplication.Extensions.cs` в конце вызова метода `MapGet`, который извлекает товары по фрагменту названия, добавьте вызов метода `RequiresCors`:

```
app.MapGet("api/products/{name}", (
  [FromServices] NorthwindContext db,
  [FromRoute] string name) =>
    db.Products.Where(p => p.ProductName.Contains(name)))
```

```
    .WithName("GetProductsByName")
    .WithOpenApi()
    .Produces<Product[]>(StatusCodes.Status200OK)
    .RequireCors(policyName: "Northwind.Mvc.Policy");
```

3. Запустите проект `Northwind.WebApi.Service` в режиме `https` без отладки.
4. Запустите проект `Northwind.WebApi.Client.Mvc` в режиме `https` без отладки.
5. В Chrome откройте панель Developer Tools (Инструменты разработчика) и перейдите на вкладку Console (Консоль).
6. На главной веб-странице приложения в текстовом поле введите слово `cha` и нажмите кнопку Get Products. Убедитесь, что в консоли отображается JSON-документ, а таблица содержит три товара.
7. Закройте браузер и завершите работу веб-сервера.

Другие политики CORS

Вы можете управлять следующими параметрами:

- *допустимыми источниками*, например https://*.example.com/;
- *допустимыми HTTP-методами*, например `GET`, `POST`, `DELETE` и т. д.;
- *допустимыми заголовками HTTP-запросов*, например `Content-Type`, `Content-Language`, `x-custom-header` и т. д.;
- *открытыми заголовками HTTP-ответов*, указывающими, какие заголовки включать в ответ без редактирования (по умолчанию заголовки могут изменяться), например `x-custom-header`.

Подробнее о политиках CORS можно узнать на официальной странице документации по адресу https://learn.microsoft.com/ru-ru/aspnet/core/security/cors#cors-policy-options.

Теперь, разобравшись, что CORS не защищает веб-сервис, рассмотрим способы предотвращения распространенных атак.

Защита от DoS-атак путем ограничения количества запросов

Атака типа *«отказ в обслуживании» (Denial of Service, DoS)* — это злонамеренная попытка перегрузить веб-сервис, отправляя ему чрезмерное количество запросов. Если все запросы поступают с одного IP-адреса, их можно заблокировать. Однако в случае *распределенных DoS-атак (Distributed DoS, DDoS)*, когда атака идет с множества IP-адресов, отфильтровать вредоносные запросы от легитимных становится сложнее.

Один из способов защиты — ограничение количества запросов для всех пользователей, предоставляя более высокий лимит авторизованным клиентам. Допускается необходимый минимум запросов (в минуту) со стороны легитимных клиентов, зависящий от веб-сервиса. Это полезно не только для защиты от атак, но и для предотвращения случайной перегрузки сервиса добросовестными пользователями. Это также может быть основой для монетизации API, когда разные тарифные планы предполагают разное количество допустимых запросов. Так, коммерческие веб-сервисы, такие как Twitter/X и Reddit, взимают немалые деньги за доступ к своим веб-интерфейсам. При превышении установленного лимита сервис должен возвращать HTTP-коды:

- `429 Too Many Requests` — превышен лимит запросов;
- `503 Service Unavailable` — сервис временно недоступен.

Дельный совет: если веб-сервис должен быть масштабируемым и защищенным (в том числе и его API), стоит воспользоваться облачными решениями, такими как Azure API Management, вместо попыток реализовать собственное ограничение запросов. Подробнее об этом можно узнать по адресу https://learn.microsoft.com/ru-ru/azure/api-management/.

Ограничение количества запросов с помощью пакета AspNetCoreRateLimit

Для .NET 6 и новее существует сторонний пакет `AspNetCoreRateLimit`, позволяющий гибко управлять ограничением запросов по IP-адресу или идентификатору клиента.

1. В файле проекта `Northwind.WebApi.Service` подключите пакет `AspNetCoreRateLimit`:

```
<PackageReference Include="AspNetCoreRateLimit" Version="5.0.0" />
```

2. Выполните сборку проекта `Northwind.WebApi.Service` для восстановления зависимостей.
3. В файле `appsettings.Development.json` настройте параметры ограничения запросов по умолчанию и специфичные для определенных клиентов:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.HttpLogging": "Information"
    }
  },
  "ClientRateLimiting": {
    "EnableEndpointRateLimiting": false,
```

```
    "StackBlockedRequests": false,
    "ClientIdHeader": "X-Client-Id",
    "HttpStatusCode": 429,
    "EndpointWhitelist": [ "get:/api/license", "*:/api/status" ],
    "ClientWhitelist": [ "dev-id-1", "dev-id-2" ],
    "GeneralRules": [
      {
        "Endpoint": "*",
        "Period": "10s",
        "Limit": 2
      },
      {
        "Endpoint": "*",
        "Period": "12h",
        "Limit": 100
      }
    ]
  },
  "ClientRateLimitPolicies": {
    "ClientRules": [
      {
        "ClientId": "console-client-abc123",
        "Rules": [
          {
            "Endpoint": "*",
            "Period": "10s",
            "Limit": 5
          },
          {
            "Endpoint": "*",
            "Period": "12h",
            "Limit": 250
          }
        ]
      }
    ]
  }
}
```

Ключевые моменты

- Параметр `EnableEndpointRateLimiting: false` — одно правило применяется ко всем конечным точкам.
- Клиент может идентифицировать себя, передавая в заголовке `X-Client-Id` уникальное строковое значение.
- При превышении лимита возвращается код ответа `429 Too Many Requests`.
- Некоторые конечные точки (например, проверка лицензии и статус сервиса) можно добавить в белый список, исключив их из ограничений. Мы не будем сейчас реализовывать эти функции, так как количество запросов для этих

конечных точек тоже стоит ограничить; иначе этой уязвимостью может воспользоваться злоумышленник и перегрузить сервер.

- В белом списке клиентов можно указать внутренние учетные записи разработчиков. В примере это клиенты с идентификаторами `dev-id-1` и `dev-id-2`. Их не касаются ограничения количества запросов.
- Общие лимиты (по умолчанию): 2 запроса каждые 10 секунд (первое правило), 100 запросов каждые 12 часов (второе правило).
- Специальные (менее ограниченные) лимиты: клиент `console-client-abc123` может выполнять до 5 запросов каждые 10 секунд и до 250 запросов каждые 12 часов.

4. В файле `IServiceCollection.Extensions.cs` импортируйте пространство имен для настройки ограничения количества запросов:

```
using AspNetCoreRateLimit; // Для параметра ClientRateLimitOptions и т. д.
```

5. В том же файле определите метод расширения для загрузки конфигурации и применения параметров ограничения количества запросов:

```
public static IServiceCollection AddCustomRateLimiting(
  this IServiceCollection services, ConfigurationManager configuration)
{
  // Добавление сервисов для хранения в памяти счетчиков
  // и правил ограничения количества запросов
  services.AddMemoryCache();
  services.AddInMemoryRateLimiting();

  // Загрузка параметров ограничения количества запросов по умолчанию
  // из файла appsettings.json
  services.Configure<ClientRateLimitOptions>(
    configuration.GetSection("ClientRateLimiting"));

  // Загрузка политик, специфичных для клиента, из файла appsettings.json
  services.Configure<ClientRateLimitPolicies>(
    configuration.GetSection("ClientRateLimitPolicies"));

  // Регистрация конфигурации
  services.AddSingleton
    <IRateLimitConfiguration, RateLimitConfiguration>();

  return services;
}
```

6. В файле `Program.cs` после создания конструктора добавьте код для загрузки конфигурации и применения параметров ограничения количества запросов:

```
builder.Services.AddCustomRateLimiting(builder.Configuration);
```

7. В файле `IServiceCollection.Extensions.cs`, в настройках HTTP-журналирования, добавьте выражение, запрещающее редактирование заголовков с ограничением запросов:

```
services.AddHttpLogging(options =>
{
  // Добавление заголовка Origin для защиты от редактирования
  options.RequestHeaders.Add("Origin");

  // Добавление заголовков ограничения количества запросов
  // для защиты от изменения
  options.RequestHeaders.Add("X-Client-Id");
  options.ResponseHeaders.Add("Retry-After");

  // По умолчанию тело ответа не включается в результат
  options.LoggingFields = HttpLoggingFields.All;
});
```

8. В файле `WebApplication.Extensions.cs` импортируйте пространство имен для ограничения количества запросов:

```
using AspNetCoreRateLimit; // Для интерфейса IClientPolicyStore и т. д.
```

9. В том же файле добавьте метод расширения для загрузки политик клиентов. Таким образом, политики будут загружаться из файлов конфигурации, а затем будут накладываться клиентские ограничения на количество запросов:

```
public static async Task
  UseCustomClientRateLimiting(this WebApplication app)
{
  using (IServiceScope scope = app.Services.CreateScope())
  {
    IClientPolicyStore clientPolicyStore = scope.ServiceProvider
      .GetRequiredService<IClientPolicyStore>();

    await clientPolicyStore.SeedAsync();
  }

  app.UseClientRateLimiting();
}
```

10. В файле `Program.cs` после вызова `UseHttpLogging` добавьте вызов метода для применения политики ограничения запросов:

```
await app.UseCustomClientRateLimiting();
```

После этих шагов сервис будет защищен от перегрузки запросами и сможет контролировать доступ клиентов.

Создание консольного клиента с ограничением скорости

Теперь можно создать консольное приложение, которое будет клиентом веб-сервиса.

1. В редакторе кода создайте проект консольного приложения `Northwind.WebApi.Client.Console` и поместите его в решение `Chapter08`.
2. В файле созданного проекта включите строгую обработку предупреждений, статически импортируйте класс `System.Console` на глобальном уровне и добавьте ссылку на проект моделей сущностей:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.EntityModels
➥ .SqlServer\Northwind.Common.EntityModels.SqlServer.csproj" />
</ItemGroup>
```

3. Выполните сборку проекта `Northwind.WebApi.Client.Console` в терминале, чтобы скомпилировать его и скопировать сборку в соответствующую папку `bin`.
4. В этом же проекте создайте файл класса `Program.Helpers.cs`.
5. В созданном файле `Program.Helpers.cs` добавьте метод для вывода в консоль текста, окрашенного в основной цвет:

```
partial class Program
{
  private static void WriteInColor(string text, ConsoleColor foregroundColor)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = foregroundColor;
    Write(text);
    ForegroundColor = previousColor;
  }
}
```

6. Удалите содержимое файла `Program.cs`. Добавьте запрос имени клиента для идентификации. Создайте HTTP-клиент, который ежесекундно будет запрашивать первую страницу с товарами из веб-сервиса. Приложение должно работать до тех пор, пока пользователь не нажмет Ctrl+C, завершив тем самым работу консольного приложения:

```
using Northwind.EntityModels; // Для класса Product
using System.Net.Http.Json; // Для метода ReadFromJsonAsync<T>

Write("Enter a client name or press Enter: ");
string? clientName = ReadLine();
```

```
if (string.IsNullOrEmpty(clientName))
{
  clientName = $"console-client-{Guid.NewGuid()}";
}

WriteLine($"X-Client-Id will be: {clientName}");

HttpClient client = new();

client.BaseAddress = new("https://localhost:5081");

client.DefaultRequestHeaders.Accept.Add(new("application/json"));

// Указание идентификатора клиента с ограничением количества
// запросов для этого консольного приложения
client.DefaultRequestHeaders.Add("X-Client-Id", clientName);

while (true)
{
  WriteInColor(string.Format("{0:hh:mm:ss}: ",
    DateTime.UtcNow), ConsoleColor.DarkGreen);

  int waitFor = 1; // В секундах

  try
  {
    HttpResponseMessage response = await client.GetAsync("api/products");

    if (response.IsSuccessStatusCode)
    {
      Product[]? products =
        await response.Content.ReadFromJsonAsync<Product[]>();

      if (products != null)
      {
        foreach (Product product in products)
        {
          Write(product.ProductName);
          Write(", ");
        }
        WriteLine();
      }
    }
    else
    {
      WriteInColor(string.Format("{0}: {1}", (int)response.StatusCode,
        await response.Content.ReadAsStringAsync()),
        ConsoleColor.DarkRed);

      WriteLine();
    }
  }
```

```
    catch (Exception ex)
    {
      WriteLine(ex.Message);
    }

    await Task.Delay(TimeSpan.FromSeconds(waitFor));
  }
```

7. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
8. Запустите проект `Northwind.WebApi.Service` в режиме `https` без отладки.
9. Запустите проект `Northwind.WebApi.Client.Console` без отладки.
10. В консольном приложении нажмите клавишу Enter, чтобы сгенерировать идентификатор клиента на основе GUID.
11. Запустите второй экземпляр `Northwind.WebApi.Client.Console` в режиме `https` без отладки. Теперь у нас запущены два клиента.
12. В консольном приложении нажмите клавишу Enter, чтобы сгенерировать идентификатор клиента на основе GUID.
13. Обратите внимание, что каждый клиент может выполнить два запроса, после чего начнет получать код ответа `429` (см. вывод ниже и рис. 8.9):

```
Enter a client name or press Enter:
X-Client-Id will be: console-client-d54c61ba-66bb-4e39-9c1a-7af6e2bf647e
07:32:18: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's
Boysenberry Spread, Uncle Bob's Organic Dried Pears, Northwoods Cranberry
Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora, 07:32:20: Chai, Chang,
Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread,
Uncle Bob's Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso
Cabrales, Queso Manchego La Pastora,
07:32:21: 429: API calls quota exceeded! maximum admitted 2 per 10s.
07:32:22: 429: API calls quota exceeded! maximum admitted 2 per 10s.
07:32:23: 429: API calls quota exceeded! maximum admitted 2 per 10s.
07:32:24: 429: API calls quota exceeded! maximum admitted 2 per 10s.
07:32:25: 429: API calls quota exceeded! maximum admitted 2 per 10s.
07:32:26: 429: API calls quota exceeded! maximum admitted 2 per 10s.
07:32:27: 429: API calls quota exceeded! maximum admitted 2 per 10s.
07:32:28: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's
Boysenberry Spread, Uncle Bob's Organic Dried Pears, Northwoods Cranberry
Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora, 07:32:29: Chai, Chang,
Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread,
Uncle Bob's Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso
Cabrales, Queso Manchego La Pastora,
07:32:30: 429: API calls quota exceeded! maximum admitted 2 per 10s.
07:32:31: 429: API calls quota exceeded! maximum admitted 2 per 10s.
07:32:32: 429: API calls quota exceeded! maximum admitted 2 per 10s.
```

```
C:\apps-services-net8\Chapter08\Northwind.WebApi.Client.Console\bin\Debug\net8.0\Northwind.WebApi.Client.Console.exe

Enter a client name or press Enter:
X-Client-Id will be: console-client-8aeb2e7e-3679-4fba-ba4b-93dc2fa7429f
12:48:40: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's
Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
12:48:42: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's
Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
12:48:43: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:44: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:45: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:46: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:47: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:48: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:49: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:50: 429: API calls quota exceeded! maximum admitted 2 per 10s.
12:48:51: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's
Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
12:48:52: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's Boysenberry Spread, Uncle Bob's
Organic Dried Pears, Northwoods Cranberry Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
12:48:53: 429: API calls quota exceeded! maximum admitted 2 per 10s.
```

Рис. 8.9. Консольное приложение, превышающее лимит частоты запросов к веб-сервису

14. Завершите работу обоих консольных приложений. Останавливать веб-сервис не требуется.
15. В терминале веб-сервиса изучите HTTP-журналы.
    - В каждом запросе указан идентификатор клиента (в заголовке `X-Client-Id`).
    - Если клиент превысил лимит, запрос блокируется.
    - В ответе содержится заголовок `Retry-After`, указывающий, сколько секунд нужно подождать перед следующей попыткой:

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/1.1
      Method: GET
      Scheme: https
      PathBase:
      Path: /api/products
      Accept: application/json
      Host: localhost:5081
      X-Client-Id: console-client-d54c61ba-66bb-4e39-9c1a-7af6e2bf647e
info: AspNetCoreRateLimit.ClientRateLimitMiddleware[0]
      Request get:/api/products from ClientId console-client-d54c61ba-66bb-
4e39-9c1a-7af6e2bf647e has been blocked, quota 2/10s exceeded by 3.
Blocked by rule *, TraceIdentifier 0HMIKGNJQEK5P:0000000E. MonitorMode: False
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 429
      Content-Type: text/plain
      Retry-After: 6
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[4]
      ResponseBody: API calls quota exceeded! maximum admitted 2 per 10s.
```

16. В файле `Program.cs` проекта `Northwind.WebApi.Client.Console` добавьте обработку заголовка `Retry-After`, чтобы перед выводом сообщения об ошибке темно-красным цветом приложение определяло время ожидания:

```
string retryAfter = response.Headers
  .GetValues("Retry-After").ToArray()[0];

if (int.TryParse(retryAfter, out waitFor))
{
  retryAfter = string.Format(
    "I will retry after {0} seconds.", waitFor);
}

WriteInColor(string.Format("{0}: {1} {2}", (int)response.StatusCode,
  await response.Content.ReadAsStringAsync(), retryAfter),
  ConsoleColor.DarkRed);
```

Значение переменной waitFor присваивается на основе заголовка Retry-After и используется для приостановки работы консольного приложения через асинхронную задержку:

```
await Task.Delay(TimeSpan.FromSeconds(waitFor));
```

17. Запустите проект `Northwind.WebApi.Client.Console` без отладки.
18. В консольном приложении нажмите клавишу Enter, чтобы сгенерировать идентификатор клиента на основе GUID.
19. Теперь приложение будет ждать предложенное количество секунд перед следующим запросом:

```
Enter a client name:
X-Client-Id will be: console-client-add7613f-51a9-4c4a-8ec7-0244203d2e19
07:45:01: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's
Boysenberry Spread, Uncle Bob's Organic Dried Pears, Northwoods Cranberry
Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
07:45:02: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's
Boysenberry Spread, Uncle Bob's Organic Dried Pears, Northwoods Cranberry
Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
07:45:03: 429: API calls quota exceeded! maximum admitted 2 per 10s.
I will retry after 8 seconds.
07:45:11: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's
Boysenberry Spread, Uncle Bob's Organic Dried Pears, Northwoods Cranberry
Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
07:45:12: Chai, Chang, Aniseed Syrup, Chef Anton's Cajun Seasoning, Grandma's
Boysenberry Spread, Uncle Bob's Organic Dried Pears, Northwoods Cranberry
Sauce, Ikura, Queso Cabrales, Queso Manchego La Pastora,
07:45:13: 429: API calls quota exceeded! maximum admitted 2 per 10s.
I will retry after 8 seconds.
```

20. Перезапустите проект `Northwind.WebApi.Client.Console` без отладки.
21. В консольном приложении введите имя `dev-id-1` и убедитесь, что для этого клиента ограничение по количеству запросов не применяется (он может быть учетной записью разработчика).
22. Перезапустите проект `Northwind.WebApi.Client.Console` без отладки.
23. В консольном приложении введите имя `console-client-abc123` — для этого клиента применяется другое ограничение количества запросов:

```
info: AspNetCoreRateLimit.ClientRateLimitMiddleware[0]
      Request get:/api/products from ClientId console-client-abc123 has
been blocked, quota 5/10s exceeded by 1. Blocked by rule *, TraceIdentifier
0HMIKGS1TPSHJ:00000006. MonitorMode: False
```

Ограничение количества запросов в ASP.NET Core с помощью промежуточного ПО

В версии ASP.NET Core 7 появилось встроенное промежуточное ПО для ограничения количества запросов. Ранее оно распространялось как отдельный NuGet-пакет, а теперь включено в состав ASP.NET Core.

Оно зависит от пакета `System.Threading.RateLimiting`, разработанного компанией Microsoft. Этот механизм менее гибкий, чем сторонние решения, поэтому в книге он не рассматривается. Тем не менее дополнительную информацию о нем я опубликовал по адресу https://github.com/markjprice/apps-services-net8/blob/main/docs/ch08-rate-limiting.md.

Официальная документация по ограничению количества запросов в ASP.NET Core доступна по адресу https://learn.microsoft.com/ru-ru/aspnet/core/performance/rate-limit.

Защита веб-сервисов от атак — критически важная задача. А можно ли параллельно повысить производительность веб-сервисов? Какие методы для этого существуют?

Оптимизация производительности с помощью технологии Native AOT

Технология *Native AOT* позволяет создавать приложения и сервисы, которые:

- *самодостаточны* — могут работать без установленной среды выполнения .NET;
- *компилируются заранее (ahead-of-time, AOT)* — преобразуют код в нативный машинный код во время публикации, а не на этапе выполнения. Это обеспе-

чивает более быстрое время запуска и потенциально снижает потребление памяти. Такой подход особенно полезен для масштабируемых микросервисов (со множеством экземпляров), которые часто останавливаются и перезапускаются.

Вместо *JIT-компиляции (just-in-time)* во время выполнения *Native AOT* предлагает компиляцию промежуточного кода *IL (intermediate language)* в машинный на этапе публикации. Однако это требует указания целевой платформы, например *Windows x64* или *Linux ARM*.

Во время разработки в редакторе кода используется *JIT-компилятор*, даже если Native AOT включена. Тем не менее функции, несовместимые с Native AOT, будут отключены или вызовут исключения. Кроме того, включается анализатор кода, который предупреждает о возможных проблемах совместимости.

Ограничения Native AOT

При всех преимуществах Native AOT имеет ряд ограничений:

- отсутствует динамическая загрузка сборок;
- *нет генерации кода на этапе выполнения* (например, через `System.Reflection.Emit`);
- требуется сокращение кода, что накладывает ограничения на некоторые библиотеки;
- сборка должна быть самодостаточной, то есть включать все используемые библиотеки, что увеличивает ее размер.

Даже если ваше приложение не использует эти функции, компоненты самой платформы .NET активно применяют их. Например, ASP.NET Core MVC (включая сервисы веб-API с контроллерами) использует динамическую генерацию кода. EF Core также полагается на механизмы генерации кода во время выполнения.

Разработчики .NET работают над улучшением совместимости Native AOT. В версии .NET 8 уже есть базовая поддержка ASP.NET Core (для минимальных API), но EF Core пока не поддерживается. Ожидается, что поддержка ASP.NET Core MVC и частично EF Core появится в .NET 9, а полноценная поддержка Native AOT может быть реализована только в .NET 10. Вот тогда мы можем создавать приложения и сервисы с помощью Native AOT со всеми присущими этой технологии преимуществами.

При публикации таких проектов анализаторы кода предупреждают о несовместимых функциях. Однако далеко не все библиотеки адаптированы для работы с Native AOT.

Наиболее распространенный атрибут, указывающий на несовместимость, — `[RequiresDynamicCode]`.

В копилку знаний: общие сведения о предупреждениях AOT опубликованы на сайте по адресу https://learn.microsoft.com/ru-ru/dotnet/core/deploying/native-aot/fixing-warnings.

Рефлексия и Native AOT

Рефлексия используется для анализа типов, вызова методов и динамической генерации кода во время выполнения.

Технология Native AOT допускает частичное использование рефлексии, но сокращение кода при компиляции может привести к удалению членов типов, доступ к которым возможен только через рефлексию. Это вызовет исключение во время выполнения.

Дельный совет: если в коде есть члены, к которым обращаются только через рефлексию, их следует аннотировать с помощью атрибута [DynamicallyAccessedMembers]. Это предотвратит их удаление Native AOT.

Поддержка Native AOT в ASP.NET Core

В версии .NET 7 механизм Native AOT поддерживался только для консольных приложений и библиотек классов под Windows и Linux, но не под macOS и не в ASP.NET Core.

В версии .NET 8 была реализована поддержка macOS и ASP.NET Core, но далеко не всех компонентов.

Полностью поддерживаются следующие функциональные возможности ASP.NET Core:

- `gRPC`
- `CORS`
- `HealthChecks`
- `HttpLogging`
- `Localization`
- `OutputCaching`
- `RateLimiting`
- `RequestDecompression`
- `ResponseCaching`
- `ResponseCompression`
- `Rewrite`
- `StaticFiles`
- `WebSockets`

Частично поддерживаются: минимальные API.

Не поддерживаются (на момент написания книги):

- MVC;
- Blazor Server;
- SignalR;
- проверка подлинности (кроме JWT);
- сессии;
- SPA.

Итак, веб-сервис на основе минимальных API ASP.NET Core реализуется путем сопоставления HTTP-запроса с лямбда-выражением:

```
app.MapGet("/", () => "Hello World!");
```

Во время выполнения ASP.NET Core использует класс `RequestDelegateFactory` *(RDF)* для преобразования вызовов `MapX` в `RequestDelegate`. Однако RDF — это динамический код, который несовместим с Native AOT.

В .NET 8 эта проблема решается с помощью *генератора исходного кода (Request Delegate Generator, RDG)*, который выполняет аналогичную работу на этапе компиляции, а не во время выполнения. Это позволяет Native AOT анализировать код статически при публикации.

В копилку знаний: о создании собственного генератора исходного кода можно узнать по адресу https://github.com/markjprice/apps-services-net8/blob/main/docs/ch01-dynamic-code.md#creating-source-generators.

Требования к Native AOT

Для разных операционных систем необходимо выполнить дополнительные настройки.

- В Windows установите в Visual Studio 2022 рабочую нагрузку Desktop development with C++ со всеми компонентами по умолчанию.
- В Linux необходимо установить компиляторы и зависимости для .NET. Например, в Ubuntu 18.04+ выполните команду `sudo apt-get install clang zlib1g-dev`.

Внимание! Кросс-платформенная публикация Native AOT не поддерживается. То есть проект Native AOT должен публиковаться в той же ОС, на которой будет выполняться. Нельзя опубликовать приложение в Linux и запустить его в Windows (и наоборот).

Добавление Native AOT в проект

Чтобы включить публикацию Native AOT, необходимо добавить в файл проекта элемент `<PublishAot>`:

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <PublishAot>true</PublishAot>
```

Включение сериализации JSON в Native AOT

Для работы с JSON в Native AOT требуется использовать генератор исходного кода `System.Text.Json`. Все типы моделей, передаваемые в качестве параметров или возвращаемые из методов, должны быть зарегистрированы в классе `JsonSerializerContext`:

```
[JsonSerializable(typeof(Product)] // Отдельный элемент Product
[JsonSerializable(typeof(Product[]))] // Массив элементов Products
public partial class MyJsonSerializerContext : JsonSerializerContext { }
```

Контекст сериализатора необходимо добавить в зависимости сервиса:

```
builder.Services.ConfigureHttpJsonOptions(options =>
{
  options.SerializerOptions.AddContext<MyJsonSerializerContext>();
});
```

Создание проекта Native AOT

Теперь рассмотрим практический пример с созданием нового шаблона проекта.

1. В решении `Chapter08` создайте проект веб-сервиса, совместимый с Native AOT:
   - выберите шаблон проекта ASP.NET Core Web API (native AOT) (Веб-API ASP.NET Core (native AOT)) / `webapiaot`.

Этот шаблон появился в .NET 8 и отличается от стандартного проекта Web API (Веб-API) / webapi. Он не поддерживает контроллеры, так как Native AOT работает только с минимальными API. В нем отсутствует поддержка протокола HTTPS, поскольку в облачных развертываниях шифрование обычно обрабатывается обратным прокси.

В JetBrains Rider выберите шаблон ASP.NET Core Web Application (Веб-приложение ASP.NET Core), а затем в раскрывающемся списке Type (Тип) выберите вариант Web API (native AOT) (Веб-API (native AOT)):

- в поле Solution name (Имя решения) укажите значение `Chapter08`;
- в поле Project name (Имя проекта) укажите значение `Northwind.MinimalAot.Service`;

- снимите флажок Enable Docker (Включить поддержку контейнера);
- снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
- в файле `launchSettings.json` в каталоге `Properties` убедитесь, что настроен только протокол HTTP. Удалите параметр `launchUrl` и измените порт на `5083`:

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "",
      "applicationUrl": "http://localhost:5083",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

2. В файле проекта присвойте параметру инвариантной глобализации (`InvariantGlobalization`) значение `false` и добавьте строгую обработку предупреждений. Убедитесь, что публикация Native AOT включена, и подключите пакет `Microsoft.Data.SqlClient`:

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>false</InvariantGlobalization>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
    <PublishAot>true</PublishAot>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Data.SqlClient" Version="5.1.2" />
  </ItemGroup>

</Project>
```

3. Создайте файл класса `Product.cs` и определите в нем класс, содержащий только три столбца таблицы `Products`, которые требуется получить:

```
namespace Northwind.Models;

public class Product
```

```
{
  public int ProductId { get; set; }
  public string? ProductName { get; set; }
  public decimal? UnitPrice { get; set; }
}
```

4. Создайте файл класса `NorthwindJsonSerializerContext.cs`.
5. Настройте в нем сериализацию класса `Product` и обработку объектов `Product` в формат JSON:

```
using System.Text.Json.Serialization; // Для класса JsonSerializerContext
using Northwind.Models;               // Для класса Product

namespace Northwind.Serialization;

[JsonSerializable(typeof(Product))]
[JsonSerializable(typeof(List<Product>))]
internal partial class NorthwindJsonSerializerContext
  : JsonSerializerContext { }
```

6. Удалите файл `Northwind.MinimalAot.Service.http`.
7. Создайте файл класса `WebApplication.Extensions.cs`.
8. Добавьте в него метод расширения для класса `WebApplication`, сопоставляющий HTTP-запросы `GET` и возвращающий из БД Northwind через ADO.NET текстовый ответ со списком или всех товаров, или только позиций с минимальной ценой:

```
using Microsoft.Data.SqlClient; // Для объекта SqlConnection и т. д.
using Northwind.Models;         // Для класса Product
using System.Data;              // Для свойства CommandType

namespace Packt.Extensions;

public static class WebApplicationExtensions
{
  public static WebApplication MapGets(this WebApplication app)
  {
    // app.MapGet(pattern, handler);

    app.MapGet("/", () => "Hello from a native AOT minimal API web service.");

    app.MapGet("/products", GetProducts);

    app.MapGet("/products/{minimumUnitPrice:decimal?}", GetProducts);

    return app;
  }

  private static List<Product> GetProducts(decimal? minimumUnitPrice = null)
  {
    SqlConnectionStringBuilder builder = new();
```

```
    builder.InitialCatalog = "Northwind";
    builder.MultipleActiveResultSets = true;
    builder.Encrypt = true;
    builder.TrustServerCertificate = true;
    builder.ConnectTimeout = 10; // По умолчанию 30 секунд
    builder.DataSource = ".";    // Локальный SQL-сервер
    builder.IntegratedSecurity = true;

    /*
    // Для проверки подлинности SQL Server Authentication:
    builder.UserID = Environment.GetEnvironmentVariable("MY_SQL_USR");
    builder.Password = Environment.GetEnvironmentVariable("MY_SQL_PWD");
    builder.PersistSecurityInfo = false;
    */

    SqlConnection connection = new(builder.ConnectionString);

    connection.Open();

    SqlCommand cmd = connection.CreateCommand();

    cmd.CommandType = CommandType.Text;
    cmd.CommandText =
      "SELECT ProductId, ProductName, UnitPrice FROM Products";

    if (minimumUnitPrice.HasValue)
    {
      cmd.CommandText += " WHERE UnitPrice >= @minimumUnitPrice";

      cmd.Parameters.AddWithValue("minimumUnitPrice", minimumUnitPrice);
    }

    SqlDataReader r = cmd.ExecuteReader();

    List<Product> products = new();

    while (r.Read())
    {
      Product p = new()
      {
        ProductId = r.GetInt32("ProductId"),
        ProductName = r.GetString("ProductName"),
        UnitPrice = r.GetDecimal("UnitPrice")
      };
      products.Add(p);
    }

    r.Close();

    return products;
  }
}
```

Платформа EF Core не поддерживается в Native AOT, поэтому используется ADO.NET SqlClient — более низкоуровневый, но быстрый и эффективный API. В будущем, возможно, в версии .NET 9 или .NET 10 поддержка EF Core будет реализована в полном объеме.

9. Откройте файл `Program.cs` и обратите внимание на вызов метода `CreateSlimBuilder()`, который включает только базовые функции ASP.NET Core и минимизирует размер веб-сервиса:

```
var builder = WebApplication.CreateSlimBuilder(args);
```

Метод CreateSlimBuilder() не поддерживает HTTPS и HTTP/3 по умолчанию, но их можно добавить вручную. Этот метод включает конфигурацию JSON-файлов (appsettings.json) и журналирование.

10. В файле `Program.cs` после вызова метода `builder.Build()` удалите примеры `todo/todos` и их конечные точки:

```
var sampleTodos = new Todo[]
{
  new(1, "Walk the dog"),
  new(2, "Do the dishes", DateOnly.FromDateTime(DateTime.Now)),
  new(3, "Do the laundry", DateOnly.FromDateTime(DateTime.Now.AddDays(1))),
  new(4, "Clean the bathroom"),
  new(5, "Clean the car", DateOnly.FromDateTime(DateTime.Now.AddDays(2)))
};

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
  sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
    ? Results.Ok(todo)
    : Results.NotFound());
```

11. В нижней части файла `Program.cs` удалите определение `Todo` и класс `AppJsonSerializerContext`:

```
public record Todo(int Id, string? Title, DateOnly? DueBy = null, bool
IsComplete = false);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

12. В верхней части файла `Program.cs` удалите операторы импорта ненужных пространств имен. Импортируйте пространство имен для поддержки JSON-сериализатора и методов расширения. Измените выражение, добавляющее контекст JSON-сериализатора, чтобы использовать Northwind, а перед запуском веб-приложения вызовите метод `MapGets` для настройки маршрутов:

```
using Northwind.Serialization;
using Packt.Extensions; // Для метода MapGets()

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
  options.SerializerOptions.TypeInfoResolverChain
    .Insert(0, NorthwindJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGets();

app.Run();
```

13. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
14. Запустите веб-сервис с профилем `http`.
    - В Visual Studio 2022: выберите профиль HTTP в раскрывающемся списке, а затем откройте меню Debug ▸ Start Without Debugging (Отладка ▸ Запустить без отладки) или нажмите Ctrl+F5.
    - В программе Visual Studio Code выполните команду `dotnet run --launch-profile http`, вручную запустите браузер и перейдите по адресу `https://localhost:5083/`.
15. Убедитесь, что в браузере отображается текстовый ответ:

```
Hello from a native AOT minimal API web service.
```

16. В адресной строке добавьте `/products` и убедитесь, что сервер возвращает JSON-массив товаров (ниже показан фрагмент):

```
[{"productId":1,"productName":"Chai","unitPrice":18.0000},
{"productId":2,"productName":"Chang","unitPrice":19.0000},
{"productId":3,"productName":"Aniseed Syrup","unitPrice":10.0000},
{"productId":4,"productName":"Chef Anton's Cajun Seasoning",
"unitPrice":22.0000},
{"productId":5,"productName":"Chef Anton's Gumbo Mix", "unitPrice":21.3500},
```

```
{"productId":6,"productName":"Grandma's Boysenberry Spread",
"unitPrice":25.0000},
{"productId":7,"productName":"Uncle Bob's Organic Dried Pears",
"unitPrice":30.0000},
{"productId":8,"productName":"Northwoods Cranberry Sauce",
"unitPrice":40.0000},
{"productId":9,"productName":"Mishi Kobe Niku","unitPrice":97.0000},
{"productId":10,"productName":"Ikura","unitPrice":31.0000},
{"productId":11,"productName":"Queso Cabrales","unitPrice":21.0000},
{"productId":12,"productName":"Queso Manchego La Pastora",
"unitPrice":38.0000},
```

17. В адресной строке добавьте `/products/100`, чтобы получить список товаров с минимальной ценой (ниже показан фрагмент):

```
[{"productId":29,"productName":"Thüringer Rostbratwurst","unitPrice":123.
7900},{"productId":38,"productName":"Côte de Blaye","unitPrice":263.5000}]
```

18. Закройте браузер и завершите работу веб-сервера.

Публикация проекта Native AOT

При публикации веб-сервиса с Native AOT могут возникнуть проблемы, даже если он корректно работает во время разработки, будучи скомпилированным JIT. Поэтому следует выполнить публикацию, а не питать иллюзии, что проект будет работать.

Если в процессе публикации не возникает предупреждений AOT, можно быть уверенными, что веб-сервис успешно запустится после развертывания.

Рассмотрим исходный код и опубликуем наш веб-сервис.

1. В файле проекта `Northwind.MinimalAot.Service` добавьте элемент для эмуляции файлов, сгенерированных компилятором:

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  ...
  <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
</PropertyGroup>
```

2. Выполните сборку проекта `Northwind.MinimalAot.Service`.
3. В Visual Studio 2022 нажмите кнопку Show All Files (Показать все файлы) на панели Solution Explorer (Обозреватель решений).
4. Откройте папку `obj\Debug\net8.0\generated` и посмотрите файлы, созданные генераторами исходного кода для AOT и JSON-сериализации. Некоторые из них понадобятся в следующих шагах (рис. 8.10).

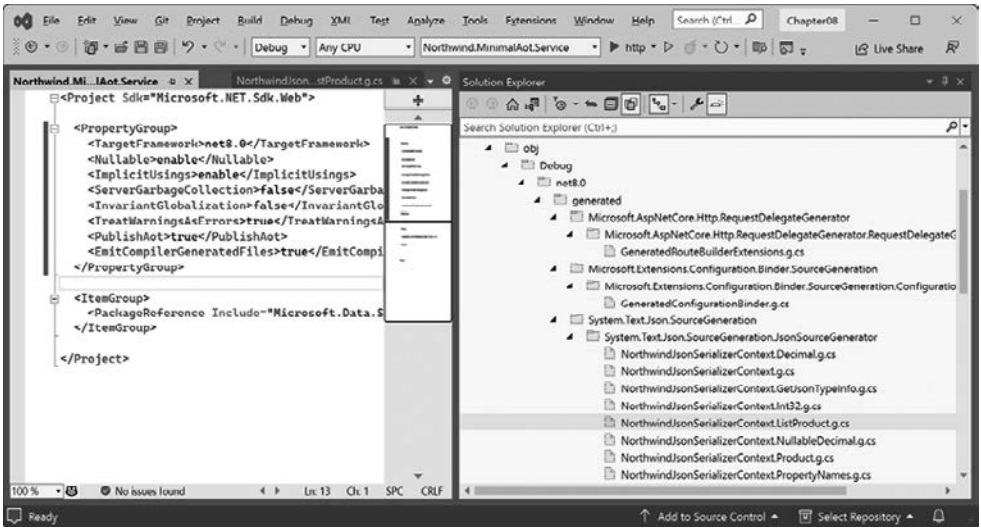


Рис. 8.10. Папки и файлы, созданные генераторами исходного кода

5. Откройте файл `GeneratedRouteBuilderExtensions.g.cs` — в нем определяется сопоставление маршрутов для минимального API веб-сервиса.
6. Откройте файл `NorthwindJsonSerializerContext.Decimal.g.cs` — он содержит код для сериализации десятичных значений, передаваемых как параметр минимальной цены товара.
7. Откройте файл `NorthwindJsonSerializerContext.ListProduct.g.cs` — в нем реализована сериализация списка объектов `Product`, возвращаемых в качестве ответа по двум маршрутам.
8. В терминале в папке `Northwind.MinimalAot`.Service выполните показанную ниже команду для публикации сервиса с Native AOT:

```
dotnet publish
```

9. Проверьте сообщение о генерации нативного кода. Обратите внимание на возможные предупреждения о сокращении кода для некоторых пакетов, например `Microsoft.Data.SqlClient` (ниже показан фрагмент):

```
Generating native code
C:\Users\markj\.nuget\packages\microsoft.data.sqlclient\5.1.1\runtimes\
win\lib\net6.0\Microsoft.Data.SqlClient.dll : warning IL2104: Assembly
'Microsoft.Data.SqlClient' produced trim warnings. For more information see
https://aka.ms/dotnet-illink/libraries [C:\apps-services-net8\Chapter08\
Northwind.MinimalAot.Service\Northwind.MinimalAot.Service.csproj]
```

```
/_/src/libraries/System.Data.Common/src/System/Data/DataTable.cs(6704): Trim
analysis warning IL2026: System.Data.DataTable.System.Xml.Serialization.
IXmlSerializable.ReadXml(XmlReader): Using member 'System.Data.DataTable.
ReadXmlSerializableInternal(XmlReader)' which has 'RequiresUnreferenced
CodeAttribute' can break functionality when trimming application code.
DataTable.ReadXml uses XmlSerialization underneath which is not trimming safe.
Members from serialized types may be trimmed if not referenced directly.
[C:\apps-services-net8\Chapter08\Northwind.MinimalAot.Service\
Northwind.MinimalAot.Service.csproj]
```

Метод DataTable.ReadXml не использовался, поэтому можно проигнорировать предупреждение о потенциально обрезанном коде.

10. На панели File Explorer (Обозреватель файлов) откройте папку `bin\Release\net8.0\win-x64\publish` и обратите внимание на EXE-файл размером около 30 Мбайт. Для развертывания веб-сервиса на другом компьютере под управлением операционной системы Windows понадобятся только этот файл и библиотека `Microsoft.Data.SqlClient.SNI.dll`. Файл `appsettings.json` требуется только для переопределения конфигурации, а PDB-файл — для отладки.
11. Запустите файл `Northwind.MinimalAot.Service.exe` и убедитесь, что веб-сервис запускается быстро и использует порт 5000 (рис. 8.11).

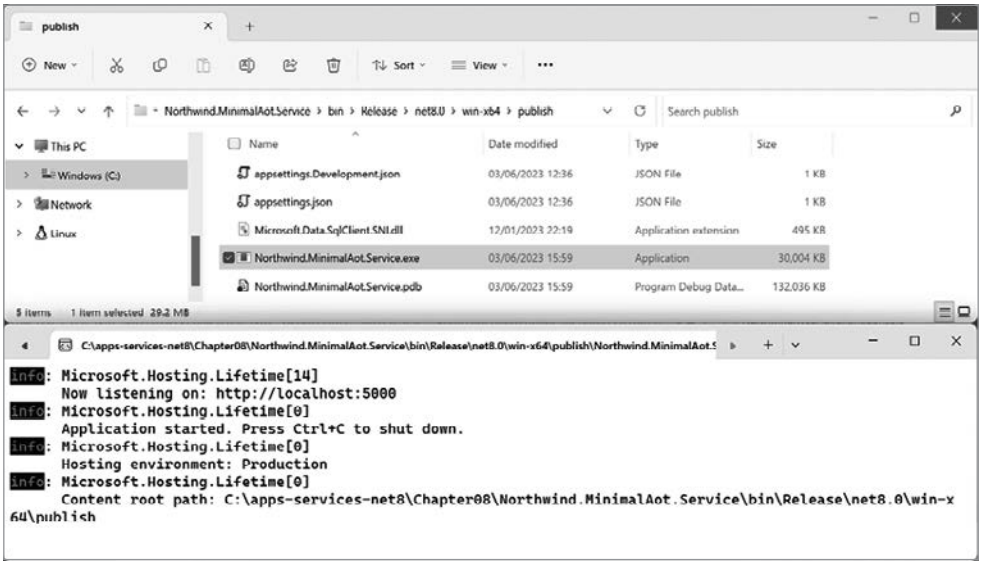


Рис. 8.11. На панели File Explorer показан опубликованный исполняемый файл. Также показан запуск веб-сервиса AOT

12. Откройте браузер и перейдите по адресу `http://localhost:5000/` — веб-сервис должен вернуть текстовый ответ.
13. Перейдите по `http://localhost:5000/products/100` — в ответе появятся две записи товаров с минимальной ценой `100`.
14. Закройте браузер и завершите работу веб-сервиса.
15. В терминале выполните публикацию веб-сервиса *`Northwind.WebApi.Service`* следующей командой:

```
dotnet publish
```

16. На панели File Explorer (Обозреватель файлов) откройте папку `bin\Release\net8.0\win-x64\publish`. Файл `Northwind.WebApi.Service.exe` занимает менее 154 Кбайт, так как он использует .NET-фреймворк и требует его установки. Вместе с EXE-файлом разворачивается набор файлов, в сумме занимающий около 14 Мбайт.
17. Запустите файл `Northwind.MinimalAot.Service.exe` и сравните скорость его загрузки с версией AOT. Теперь веб-сервис запускается медленнее и по умолчанию использует порт `5000`.
18. Откройте браузер и перейдите по адресу `http://localhost:5000/` — веб-сервис должен вернуть текстовый ответ.
19. Перейдите по `http://localhost:5000/products/100` — в ответе появятся две записи товаров с минимальной ценой `100`, но отклик происходит медленнее, чем в версии AOT.
20. Закройте браузер и завершите работу веб-сервиса.

Многие разработчики .NET давно ждали поддержки компиляции Ahead-of-Time (AOT). Компания Microsoft наконец реализовала этот механизм, и в ближайших крупных релизах он будет распространяться на все большее количество типов проектов. Технология AOT весьма перспективна, и за ее развитием стоит следить.

Работа с сервисами идентификации

Сервисы идентификации предназначены для проверки подлинности и авторизации пользователей. Важно, чтобы эти сервисы реализовывали открытые стандарты, позволяя интегрировать разрозненные системы. Среди наиболее распространенных стандартов можно выделить *OpenID Connect* и *OAuth 2.0*.

Корпорация Microsoft не планирует официальную поддержку сторонних серверов аутентификации, таких как *IdentityServer4*. В компании объясняют это тем, что «разработка и поддержка сервера аутентификации требуют плотной и полной занятости, а у Microsoft уже есть собственное решение в этой области — Azure Active Directory, которое позволяет бесплатно использовать до 500 000 объектов».

Авторизация с помощью токена JWT Bearer

JSON Web Token (JWT) — это стандарт для компактной и безопасной передачи информации в формате JSON. Передаваемые JSON-объекты подписаны цифровой подписью, поэтому данной информации можно доверять. JWT широко используется для авторизации.

Процесс выглядит так.

1. Пользователь проходит проверку подлинности в доверенной системе, используя учетные данные (например, имя пользователя и пароль, биометрию или двухфакторную проверку).
2. Доверенная сторона выдает ему JWT.
3. Токен отправляется с каждым запросом к защищенному веб-сервису.

Токен JWT состоит из трех частей, разделенных точками: заголовка, полезной нагрузки и подписи в виде `aaa.bbb.ccc`. Заголовок и полезная нагрузка кодируются в формате Base64.

Проверка подлинности клиентов сервисов с помощью JWT Bearer

При локальной разработке для создания и управления JWT можно воспользоваться консольным инструментом `dotnet user-jwts`. Значения токенов хранятся в JSON-файле в папке профиля пользователя на локальном компьютере.

Давайте защитим веб-сервис путем проверки подлинности по токену JWT и протестируем систему с помощью локального токена.

1. В файле проекта `Northwind.WebApi.Service` подключите пакет для проверки подлинности по токену JWT:

```
<PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer"
  Version="8.0.0" />
```

2. Выполните сборку проекта `Northwind.WebApi.Service` для восстановления зависимостей.
3. В файле `Program.cs` после создания конструктора добавьте код для авторизации и проверки подлинности с помощью JWT:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();

builder.Services.AddAuthentication(defaultScheme: "Bearer")
  .AddJwtBearer();
```

4. В файле `Program.cs` после создания приложения добавьте команду для использования авторизации:

```
var app = builder.Build();

app.UseAuthorization();
```

5. В файле `WebApplication.Extensions.cs` импортируйте пространство имен для управления аспектами безопасности:

```
using System.Security.Claims; // Для класса ClaimsPrincipal
```

6. В том же файле `WebApplication.Extensions.cs` после отображения HTTP-запроса `GET` для корневого пути, возвращающего простой текст, добавьте обработку запроса `GET` для защищенного пути `/secret`. В ответе должно возвращаться имя аутентифицированного пользователя, если он авторизован:

```
app.MapGet("/", () => "Hello World!")
  .ExcludeFromDescription();

app.MapGet("/secret", (ClaimsPrincipal user) =>
  string.Format("Welcome, {0}. The secret ingredient is love.",
    user.Identity?.Name ?? "secure user"))
  .RequireAuthorization();
```

7. Открыв папку проекта `Northwind.WebApi.Service` в терминале, выполните команду для создания локального токена JWT:

```
dotnet user-jwts create
```

8. Обратите внимание на автоматически сгенерированные поля `ID`, `Name` и `Token` (ниже показан фрагмент):

```
New JWT saved with ID 'd7e22000'.
Name: markjprice
Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJ1bmlxdWVfbmFtZSI6Im1hcmtqcHJpY2UiLCJzdWIiOiJtYXJran...lci1qd3RzIn0.
pGEbYKRjU98dEjxLSx7GAEm41LXMS0J80iIjuZbqrj4
```

9. В терминале выведите на экран всю информацию о сгенерированном токене:

```
dotnet user-jwts print d7e22000 --show-all
```

10. Проверьте, что схема аутентификации — `Bearer`, а токен передается с каждым запросом. Поле `Audience(s)` содержит список допустимых клиентских доменов и портов. Срок действия токена составляет три месяца. JSON-объекты содержат

заголовок и полезную нагрузку. Компактный токен JWT состоит из трех частей, разделенных точками. Заголовок и полезная нагрузка кодируются в формате Base64 (ниже показан фрагмент):

```
Found JWT with ID 'd7e22000'.
ID: d7e22000
Name: markjprice
Scheme: Bearer
Audience(s): http://localhost:30225, https://localhost:44344, http://
localhost:5080, https://localhost:5081
Not Before: 2023-09-26T10:58:18.0000000+00:00
Expires On: 2023-12-26T10:58:18.0000000+00:00
Issued On: 2023-09-26T10:58:19.0000000+00:00
Scopes: none
Roles: [none]
Custom Claims: [none]
Token Header: {„alg":"HS256","typ":"JWT"}
Token Payload: {„unique_
name":"markjprice","sub":"markjprice","jti":"d7e22000","aud":
["http://localhost:30225","https://localhost:44344","http://
localhost:5080","https://localhost:5081"],"nbf":1664189898,"exp":
1672052298,"iat":1664189899,"iss":"dotnet-user-jwts"}
Compact Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJ1bmlxdWVfbmFtZSI6Im1hcmtqcHJpY2UiLCJzdWIiOiJtYXJranByaWNl...
uZXQtdXNlci1qd3RzIn0.pGEbYKRjU98dEjxLSx7GAEm41LXMS0J80iIjuZbqrj4
```

11. В файле `appsettings.Development.json` проекта `Northwind.WebApi.Service` появится новый раздел `Authentication` с параметрами конфигурации:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.HttpLogging": "Information"
    }
  },
  "Authentication": {
    "Schemes": {
      "Bearer": {
        "ValidAudiences": [
          "http://localhost:30225",
          "https://localhost:44344",
          "http://localhost:5080"
          "https://localhost:5081"
        ],
        "ValidIssuer": "dotnet-user-jwts"
      }
    }
  }
}
```

12. Запустите проект `Northwind.WebApi.Service` в режиме `https` без отладки.
13. В браузере измените относительный путь на `/secret` и убедитесь, что сервер возвращает код ошибки `401 Unauthorized`.
14. В Visual Studio Code откройте папку `HttpRequests`.
15. Создайте в ней файл `webapi-secure-request.http` и добавьте в него запрос для получения защищенного ресурса, используя ваш токен `Bearer` (указывайте, разумеется, собственный токен `Bearer`):

```
### Получение защищенного ресурса
GET https://localhost:5081/secret/
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJ1bmlxdWVfbmFtZSI6Im1hcmtqcHJpY2UiLCJzdWIiOiJtYXJranByaWNl...
uZXQtdXNlci1qd3RzIn0.pGEbYKRjU98dEjxLSx7GAEm41LXMS0J80iIjuZbqrj4
```

16. Нажмите кнопку `Send Request` (Отправить запрос) и проанализируйте ответ сервера:

```
Welcome, secure user. The secret ingredient is love.
```

17. Закройте браузер завершите работу веб-сервиса.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 8.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. Назовите шесть методов, которые могут быть указаны в HTTP-запросе.
2. Перечислите шесть кодов HTTP-ответа и их описания.
3. Чем технология минимальных API в ASP.NET Core отличается от технологии веб-API в ASP.NET Core?
4. Как сопоставить HTTP-запрос `PUT` к `api/customers` с блоком лямбда-выражения, используя минимальные API в ASP.NET Core?
5. Как сопоставить метод/параметр лямбды со значением из маршрута, строки запроса или тела запроса в минимальных API ASP.NET Core?
6. Улучшает ли механизм CORS безопасность веб-сервиса?
7. Вы добавили в файл `Program.cs` код для включения HTTP-журналирования, но запросы и ответы не протоколируются. Почему возникла эта проблема и как ее устранить?

8. Как ограничить количество запросов для конкретного клиента с помощью пакета `AspNetCoreRateLimit`?
9. Как ограничить количество запросов для определенной конечной точки с помощью пакета `Microsoft.AspNetCore.RateLimiting`?
10. Как расшифровывается аббревиатура JWT?

Упражнение 8.2. Обзор философии дизайна Microsoft HTTP API

Сотрудники корпорации Microsoft подготовили внутреннее руководство по проектированию HTTP/REST API. Они ориентируются на этот документ при разработке своих HTTP API. Документ может стать отличной основой для создания новых стандартов. Вы можете ознакомиться с ним по адресу https://github.com/microsoft/api-guidelines.

Упражнение 8.3. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-8---building-and-securing-web-services-using-minimal-apis.

Упражнение 8.4. Шаблоны проекта Auth0

Если требуется реализовать проверку подлинности и авторизацию через Auth0, можно воспользоваться готовыми шаблонами проектов. Подробности об их использовании — в статье по адресу https://auth0.com/blog/introducing-auth0-templates-for-dotnet/.

Резюме

В этой главе вы узнали, как:

- создать веб-сервис на основе минимальных API, реализующий архитектурный стиль REST;
- настроить механизм CORS для управления доступом между разными доменами;
- реализовать разные механизмы ограничения количества запросов для защиты от DoS-атак;
- обеспечить безопасность сервисов с помощью авторизации JWT Bearer.

В следующей главе речь пойдет о том, как создать надежные и масштабируемые сервисы с функциями кэширования, очередями и автоматической обработкой временных сбоев, используя библиотеки типа Polly.

9 Кэширование, очереди и отказоустойчивые фоновые сервисы

В этой главе вы познакомитесь с различными технологиями и методами, повышающими масштабируемость и надежность сервисов независимо от выбранной технологии их реализации.

Архитектура сервисов

В главе 8 вы узнали, как создать веб-сервис на основе минимальных API. Прежде чем изучать альтернативные технологии, важно понять архитектуру сервисов и определить возможные узкие места, влияющие на их производительность и масштабируемость.

Самые медленные компоненты системы

Традиционно наиболее медленные компоненты системы следующие:

- сеть (самая медленная);
- диск;
- память;
- кэш процессора (самый быстрый).

Доступ к каждому следующему уровню хранения данных происходит быстрее, чем к предыдущему, причем разница может составлять 5–10 раз.

Стоит отметить, что современные сетевые технологии передачи данных значительно ускорились, а сервисы часто работают в облачных дата-центрах. Например, что быстрее — считать данные с локального диска или запросить их с другого сервера?

- Обращение между серверами в пределах одного центра обработки данных: 500 000 наносекунд (нс).
- Чтение с диска: 10 000 000 наносекунд.

Время доступа к данным

Старший научный сотрудник компании Google Джефф Дин в своих презентациях приводит реальное время доступа к данным (в наносекундах), демонстрируя разницу между локальными и удаленными источниками. В табл. 9.1 указаны эти данные с удобным эквивалентом, облегчающим восприятие.

Таблица 9.1. Время доступа/считывания данных для различных технологий в наносекундах и эквивалентных единицах

Технология	**Фактическое время**	**Эквивалент в реальном времени**
Цикл процессора	0,1 нс	1 секунда
Доступ к кэшу L1	0,5 нс	5 секунд
Доступ к кэшу L2	5 нс	1 минута
Блокировка/разблокировка мьютекса	25 нс	4 минуты
Доступ к оперативной памяти	100 нс	15 минут
Передача 1 Кбайт по сети 1 Гбит/с	10 000 нс	28 часов
Последовательное считывание 1 Мбайт из памяти	250 000 нс	29 дней
Запрос к серверу в одном дата-центре	500 000 нс	2 месяца
Последовательное чтение 1 Мбайт с SSD	1 000 000 нс	4 месяца
Поиск данных на диске	10 000 000 нс	3 года 3 месяца
Последовательное чтение 1 Мбайт с HDD	20 000 000 нс	6 лет 3 месяца
Передача пакета Калифорния → Нидерланды → Калифорния	150 000 000 нс	*47 лет 6 месяцев*

Источник: презентация Джеффа Дина *Designs, Lessons and Advice from Building Large Distributed Systems*, https://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf.

Здесь не столько важно сравнивать скорость чтения диска и SSD со скоростью передачи сетевых запросов, сколько осознать разницу между локальными и удаленными данными. Разница между ними огромна.

Например, если процессору нужны данные, уже находящиеся в кэше L1, он обработает их за 1 секунду. Если данные хранятся в оперативной памяти — 15 минут. Запрос к серверу в одном дата-центре займет 2 месяца. А если данные находятся на другом континенте — почти 50 лет!

Кэширование позволяет временно хранить данные как можно ближе к месту их использования, снижая задержки при доступе. Это один из ключевых инструментов для повышения производительности и масштабируемости сервисов.

Кэширование в ASP.NET Core

Кэширование позволяет системам копировать определенные данные из удаленного центра обработки данных в локальный или с сервера либо диска в память. В кэшах данные хранятся в виде пар «ключ — значение».

Одна из главных сложностей при кэшировании — найти баланс между хранением достаточного количества данных и их актуальностью. Чем больше данных копируется, тем больше ресурсов используется. Дополнительно важно продумать механизм синхронизации копий с исходными данными.

Общие рекомендации по кэшированию

Кэширование наиболее эффективно для данных, которые требуют значительных затрат на генерацию и редко изменяются. При кэшировании рекомендуется соблюдать следующие условия.

- Код не должен зависеть от кэшированных данных. Они всегда должны быть доступны из первоисточника, если они отсутствуют в кэше.
- Независимо от места хранения кэша (в памяти или в БД), этот ресурс ограничен. Следует намеренно ограничивать объем и срок хранения данных, настраивая лимиты по времени хранения и размеру. Важно отслеживать количество успешных запросов к кэшу, чтобы найти оптимальный баланс для конкретных сценариев использования.

В практических заданиях этого раздела предстоит реализовать все эти рекомендации. Для начала рассмотрим технологии кэширования, встроенные в ASP.NET Core.

Создание сервиса веб-API на основе контроллера

Создадим базовый веб-сервис для изучения технологий кэширования.

1. В редакторе кода создайте новый проект на основе контроллера веб-API, следуя этим настройкам:
   - выберите шаблон проекта ASP.NET Core Web API (Веб-API ASP.NET Core) / `webapi --use-controllers`;
   - в поле Solution name (Имя решения) укажите значение `Chapter09`;

- в поле Project name (Имя проекта) укажите значение `Northwind.WebApi.Service`;
- в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
- установите флажок Configure for HTTPS (Настроить для HTTPS);
- снимите флажок Enable Docker (Включить поддержку контейнера);
- установите флажок Use controllers (Использовать контроллеры);
- установите флажок Enable OpenAPI support (Включить поддержку OpenAPI);
- снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня).

Обязательно установите флажок Use controllers (Использовать контроллеры) или укажите ключ --use-controllers / -controllers. Мы не будем использовать минимальные API, которые применяются по умолчанию в шаблонах проектов .NET 8.

Если используется среда JetBrains Rider, лучше выполнять команду `dotnet new`, пока Rider не начнет поддерживать контроллеры.

2. Добавьте ссылку на проект Northwind для SQL Server, созданный в главе 3:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.DataContext.
SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
</ItemGroup>
```

Путь не должен содержать символы разрыва строки. Если в главе 3 вы не создали библиотеки классов, скачайте проекты решений из GitHub-репозитория для этой книги.

3. В файле проекта присвойте параметру инвариантной глобализации (`InvariantGlobalization`) значение `false` и задействуйте строгую обработку предупреждений:

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>false</InvariantGlobalization>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>
```

В шаблоне проекта веб-API ASP.NET Core для .NET 8 появилась возможность явного указания инвариантной глобализации, присвоив соответствующему параметру значение true, чтобы веб-сервис мог работать независимо от культуры. Если изменить значение на false, веб-сервис будет использовать культуру операционной системы компьютера, на котором он работает.

Подробнее о неизменном режиме глобализации можно прочитать по адресу https://github.com/dotnet/runtime/blob/main/docs/design/features/globalization-invariant-mode.md.

4. В терминале выполните сборку проекта `Northwind.WebApi.Service`, чтобы убедиться, что библиотека классов моделей сущностей компилируется корректно:

```
dotnet build
```

5. В файле `launchSettings.json` папки `Properties` измените параметр `applicationUrl` профиля `https`, установив порт `5091` для `https` и порт `5092` для `http`:

```
"profiles": {
  ...
  "https": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "launchUrl": "swagger",
    "applicationUrl": "https://localhost:5091;http://localhost:5092",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
```

Программы Visual Studio 2022 и JetBrains Rider автоматически прочитают этот файл и запустят браузер, определив параметр launchBrowser: true. Откроется страница по адресу, указанному в качестве значения параметра applicationUrl.

В Visual Studio Code после выполнения команды dotnet run браузер придется запустить вручную и перейти по адресу https://localhost:5091/swagger.

6. Удалите файл `WeatherForecast.cs`.
7. В папке `Controllers` удалите файл `WeatherForecastController.cs`.
8. В файле `Program.cs` импортируйте пространство имен `NorthwindContext` для добавления в настроенные сервисы:

```
using Northwind.EntityModels; // Для метода AddNorthwindContext

var builder = WebApplication.CreateBuilder(args);
```

```
// Добавление сервисов в контейнер
builder.Services.AddNorthwindContext();
builder.Services.AddControllers();
```

9. В папке `Controllers` создайте файл класса `ProductsController.cs`.
10. Определите в нем веб-API на основе контроллера для работы с товарами в БД Northwind (как делалось ранее для минимальных API):

```
using Microsoft.AspNetCore.Mvc; // Для атрибута [HttpGet] и т. д.
using Northwind.EntityModels; // Для классов NorthwindContext, Product

namespace Northwind.WebApi.Service.Controllers;

[Route("api/products")]
[ApiController]
public class ProductsController : ControllerBase
{
  private int pageSize = 10;

  private readonly ILogger<ProductsController> _logger;
  private readonly NorthwindContext _db;

  public ProductsController(ILogger<ProductsController> logger,
    NorthwindContext context)
  {
    _logger = logger;
    _db = context;
  }

  // GET: api/products
  [HttpGet]
  [Produces(typeof(Product[]))]
  public IEnumerable<Product> Get(int? page)
  {
    return _db.Products
      .Where(p => p.UnitsInStock > 0 && !p.Discontinued)
      .OrderBy(product => product.ProductId)
      .Skip(((page ?? 1) - 1) * pageSize)
      .Take(pageSize);
  }

  // GET: api/products/outofstock
  [HttpGet]
  [Route("outofstock")]
  [Produces(typeof(Product[]))]
  public IEnumerable<Product> GetOutOfStockProducts()
  {
    return _db.Products
      .Where(p => p.UnitsInStock == 0 && !p.Discontinued);
  }

  // GET: api/products/discontinued
  [HttpGet]
```

```
[Route("discontinued")]
[Produces(typeof(Product[]))]
public IEnumerable<Product> GetDiscontinuedProducts()
{
  return _db.Products
    .Where(product => product.Discontinued);
}

// GET api/products/5
[HttpGet("{id:int}")]
public async ValueTask<Product?> Get(int id)
{
  return await _db.Products.FindAsync(id);
}

// GET api/products/cha
[HttpGet("{name}")]
public IEnumerable<Product> Get(string name)
{
  return _db.Products.Where(p => p.ProductName.Contains(name));
}

// POST api/products
[HttpPost]
public async Task<IActionResult> Post([FromBody] Product product)
{
  _db.Products.Add(product);
  await _db.SaveChangesAsync();
  return Created($"api/products/{product.ProductId}", product);
}

// PUT api/products/5
[HttpPut("{id}")]
public async Task<IActionResult> Put(int id, [FromBody] Product product)
{
  Product? foundProduct = await _db.Products.FindAsync(id);

  if (foundProduct is null) return NotFound();

  foundProduct.ProductName = product.ProductName;
  foundProduct.CategoryId = product.CategoryId;
  foundProduct.SupplierId = product.SupplierId;
  foundProduct.QuantityPerUnit = product.QuantityPerUnit;
  foundProduct.UnitsInStock = product.UnitsInStock;
  foundProduct.UnitsOnOrder = product.UnitsOnOrder;
  foundProduct.ReorderLevel = product.ReorderLevel;
  foundProduct.UnitPrice = product.UnitPrice;
  foundProduct.Discontinued = product.Discontinued;

  await _db.SaveChangesAsync();

  return NoContent();
}
```

```
    // DELETE api/products/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> Delete(int id)
    {
      if (await _db.Products.FindAsync(id) is Product product)
      {
        _db.Products.Remove(product);
        await _db.SaveChangesAsync();
        return NoContent();
      }
      return NotFound();
    }
  }
```

11. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
12. Запустите веб-сервис в режиме `https` без отладки.
    - В Visual Studio 2022 выберите профиль https в раскрывающемся списке, а затем выполните команду меню Debug ▸ Start Without Debugging (Отладка ▸ Запустить без отладки) или нажмите Ctrl+F5. Браузер автоматически откроет страницу документации Swagger.
    - В Visual Studio Code выполните команду `dotnet run --launch- profile https`, вручную запустите браузер и перейдите на веб-страницу с документацией Swagger по адресу https://localhost:5091/swagger.

В операционной системе Windows при появлении соответствующего запроса может потребоваться разрешить доступ к локальному веб-сервису в брандмауэре Windows Defender.

13. В Swagger в рамках тестирования различных конечных точек выполните следующие запросы.
    - Получите первую страницу (10 товаров).
    - Получите шестую страницу (10 товаров).
    - Получите товар с идентификатором `77`.
    - Найдите товар, отсутствующий на складе.
    - Найдите семь товаров со скидкой.
    - Найдите товары, названия которых начинаются с `cha`.
    - Создайте (`POST`), обновите (`PUT`) и удалите (`DELETE`) товар.

    Подсказки по тестированию вы найдете по адресу https://github.com/markjprice/apps-services-net8/blob/main/docs/ch09-swagger-tests.md.

Если была завершена глава 8, можно использовать созданные .http-файлы вместо Swagger. В этом случае измените порт с 5081 на 5091.

Теперь, когда базовый веб-сервис готов, можно приступить к активации кэширования.

Кэширование объектов в оперативной памяти

Интерфейс `IMemoryCache` представляет кэш, хранящийся в локальной памяти сервера. Если ваш сервис/сайт размещен на нескольких серверах, то необходимо включить «липкие сессии». Это позволит направлять запросы от клиента на тот же сервер, на котором были выполнены предыдущие запросы, что обеспечит доступ к кэшированным данным именно с этого сервера.

Пакет `Microsoft.Extensions.Caching.Memory` содержит современную реализацию `IMemoryCache`. Рекомендуется избегать использования устаревших пакетов `System.Runtime.Caching`.

Размеры объектов в кэше могут быть определены в пользовательских единицах. Например, если вы храните строки, в качестве единицы можно использовать длину строки. Если размер неизвестен, можно определить единицу для каждой записи, чтобы ограничить их количество.

При добавлении объекта в кэш необходимо ограничить срок его хранения. Существует два вида сроков: абсолютный и скользящий. Вы можете использовать как один из них, так и оба или не использовать вовсе.

- *Абсолютный срок хранения:* указываются фиксированные дата и время, например 1 час ночи 24 декабря 2023 года. Когда наступают эти дата/время, объект удаляется. Для этого свойству `AbsoluteExpiration` записи кэша присвойте значение `DateTime`. Выбирайте этот метод, если нужны гарантии, что данные будут обновляться в определенный момент времени.
- *Скользящий срок хранения:* это временной интервал, например 20 секунд. По истечении этого времени объект удаляется. Однако при каждом считывании объекта срок его хранения сбрасывается, то есть продлевается на указанное значение. Такой вариант подходит, если данные могут оставаться актуальными в течение длительного времени. Чтобы использовать этот тип, присвойте свойству `SlidingExpiration` записи кэша значение `TimeSpan`. Способ подходит для систем управления контентом, где содержимое часто обновляется и должно быть доступно в течение длительного времени.
- *Оба метода:* если вы настроите только скользящий срок, объект может остаться в кэше навсегда. Для этого свойству `AbsoluteExpirationRelativeToNow`

присваивается значение `TimeSpan`, определяющее максимальное время хранения объекта. Этот метод совмещает преимущества обоих вариантов.

- *Ни один из методов:* можно установить для записи кэша приоритет `CacheItemPriority.NeverRemove`, что предотвратит удаление объекта из кэша.

Можно также настроить метод для обратного вызова, когда объект вытесняется из кэша. Это позволяет выполнить дополнительную бизнес-логику для обновления объекта, добавив его обратно в кэш. Для этого используется метод `RegisterPostEvictionCallback`.

Рассмотрим пример использования кэша в памяти.

1. В файле `Program.cs` проекта `Northwind.WebApi.Service` импортируйте пространство имен для управления кэшем в оперативной памяти:

```
using Microsoft.Extensions.Caching.Memory; // Для интерфейса IMemoryCache и т. д.
```

2. В том же файле `Program.cs`, в разделе настройки сервисов, после вызова `CreateBuilder` зарегистрируйте реализацию для кэша, ограничив его возможностью хранения максимум 50 товаров:

```
builder.Services.AddSingleton<IMemoryCache>(new MemoryCache(
  new MemoryCacheOptions
  {
    TrackStatistics = true,
    SizeLimit = 50 // Товары
  }));
```

3. В файле `ProductsController.cs` импортируйте пространство имен для работы с кэшем в памяти:

```
using Microsoft.Extensions.Caching.Memory; // Для интерфейса IMemoryCache
```

4. В том же файле объявите поля для хранения кэша и ключа для товаров, отсутствующих на складе:

```
private readonly IMemoryCache _memoryCache;
private const string OutOfStockProductsKey = "OOSP";

public ProductsController(ILogger<ProductsController> logger,
  NorthwindContext context,
  IMemoryCache memoryCache)
{
  _logger = logger;
  _db = context;
  _memoryCache = memoryCache;
}
```

5. В методе `GetOutOfStockProducts` файла `ProductsController.cs` добавьте код для попытки получения кэшированных товаров. Если товары не найдены в кэше,

получите их из базы данных и добавьте в кэш с установкой скользящего срока хранения 5 секунд:

```
// GET: api/products/outofstock
[HttpGet]
[Route("outofstock")]
[Produces(typeof(Product[]))]
public IEnumerable<Product> GetOutOfStockProducts()
{
  // Попытка получить кэшированное значение
  if (!_memoryCache.TryGetValue(OutOfStockProductsKey,
    out Product[]? cachedValue))
  {
    // Если кэшированное значение не найдено, получить значение из БД
    cachedValue = _db.Products
      .Where(p => p.UnitsInStock == 0 && !p.Discontinued)
      .ToArray();

      MemoryCacheEntryOptions cacheEntryOptions = new()
      {
        SlidingExpiration = TimeSpan.FromSeconds(5),
        Size = cachedValue?.Length
    };

    _memoryCache.Set(OutOfStockProductsKey, cachedValue, cacheEntryOptions);
  }

  MemoryCacheStatistics? stats = _memoryCache.GetCurrentStatistics();

  _logger.LogInformation("Memory cache. Total hits: {stats?
    .TotalHits}. Estimated size: {stats?.CurrentEstimatedSize}.");

  return cachedValue ?? Enumerable.Empty<Product>();

}
```

6. Запустите веб-сервис в режиме `https` без отладки.
7. Расположите окна так, чтобы одновременно с веб-страницей было видно терминал.
8. На веб-странице Swagger щелкните на строке GET /api/product/outofstock, чтобы развернуть этот раздел.
9. Нажмите кнопку Try it out (Попробовать).
10. Нажмите кнопку Execute (Выполнить) и обратите внимание, что EF Core выполняет SQL-запрос для получения товаров, общий счетчик просмотров равен нулю, а один товар теперь кэширован:

```
info: Northwind.WebApi.Service.Controllers.ProductsController[0]
      Memory cache. Total hits: 0. Estimated size: 1.
```

11. Нажмите кнопку Execute (Выполнить) еще несколько раз в течение 5 секунд.
    - Обратите внимание, что SQL-запрос не выполняется повторно, так как товары кэшируются, и если компонент читает их в течение 5 секунд (срока хранения), они остаются в памяти.
    - Кроме того, счетчик кэшированных товаров увеличивается каждый раз, когда в кэше обнаруживаются товары, отсутствующие на складе:

```
info: Northwind.WebApi.Service.Controllers.ProductsController[0]
      Memory cache. Total hits: 1. Estimated size: 1.
info: Northwind.WebApi.Service.Controllers.ProductsController[0]
      Memory cache. Total hits: 2. Estimated size: 1.
info: Northwind.WebApi.Service.Controllers.ProductsController[0]
      Memory cache. Total hits: 3. Estimated size: 1.
```

12. Подождите не менее 5 секунд.
13. Нажмите кнопку Execute (Выполнить) и обратите внимание на то, что EF Core снова выполняет SQL-запрос, так как товары не были прочитаны в течение 5 секунд (то есть срок хранения истек).
14. Закройте браузер и завершите работу веб-сервера.

Распределенное кэширование объектов

Распределенные кэши обладают рядом преимуществ по сравнению с кэшами в оперативной памяти. Основные особенности кэшированных таким образом объектов:

- согласованность между запросами, поступающими на несколько серверов;
- сохранение данных при перезагрузке сервера или развертывании сервиса;
- экономия ресурсов локальной памяти серверов;
- централизованное хранение данных, что исключает необходимость включать «липкие» сессии при работе с серверной фермой.

Внимание! Недостаток распределенных кэшей заключается в том, что, в отличие от кэша в оперативной памяти, который может хранить любые объекты, распределенные кэши могут работать только с массивами значений типа byte. Поэтому объект должен быть сериализован и отправлен по сети в удаленный кэш.

Интерфейс `IDistributedCache` компании Microsoft включает в себя набор методов для работы с элементами в любой реализации распределенного кэша:

- `Set/SetAsync` — для сохранения объекта в кэше;
- `Get/GetAsync` — для получения объекта из кэша;
- `Remove/RemoveAsync` — для удаления объекта из кэша;

- `Refresh/RefreshAsync` — для сброса скользящего срока хранения объекта в кэше.

Существует несколько популярных реализаций распределенного кэширования:

- SQL Server: https://learn.microsoft.com/ru-ru/aspnet/core/performance/caching/distributed#-distributed-sql-server-cache;
- Redis: https://learn.microsoft.com/ru-ru/aspnet/core/performance/caching/distributed#-distributed-redis-cache;
- NCache: http://www.alachisoft.com/ncache/aspnet-core-idistributedcache-ncache.html.

В рамках этого примера будет использоваться модель *Distributed Memory Cache* — встроенная реализация интерфейса `IDistributedCache` компании Microsoft, которая хранит данные в памяти на сервере, где запущен сервис. Это не истинный распределенный кэш, но такая реализация полезна в сценариях модульного тестирования или обучения, как в данном примере.

В дальнейшем достаточно будет изменить только конфигурацию распределенного кэша, а не код, который его использует, поскольку все взаимодействия осуществляются через зарегистрированную реализацию интерфейса `IDistributedCache`.

Приступим!

1. В проекте `Northwind.WebApi.Service` откройте файл `Program.cs` и в разделе настройки сервисов после вызова метода `CreateBuilder` добавьте реализацию для распределенного кэша в памяти:

```
builder.Services.AddDistributedMemoryCache();
```

2. В файле `ProductsController.cs` импортируйте пространство имен для работы с распределенным кэшем и сериализованными JSON-документами:

```
using Microsoft.Extensions.Caching.Distributed; // Для интерфейса
IDistributedCache
using System.Text.Json; // Для класса JsonSerializer
```

3. В том же файле объявите поля для хранения реализации распределенного кэша и ключа для товаров со скидкой:

```
private readonly IMemoryCache _memoryCache;
private const string OutOfStockProductsKey = "OOSP";

private readonly IDistributedCache _distributedCache;
private const string DiscontinuedProductsKey = "DISCP";

public ProductsController(ILogger<ProductsController> logger,
  NorthwindContext context,
  IMemoryCache memoryCache,
  IDistributedCache distributedCache)
```

```
{
  _logger = logger;
  _db = context;
  _memoryCache = memoryCache;
  _distributedCache = distributedCache;
}
```

4. В том же файле создайте приватный метод для извлечения из БД товаров со скидкой и помещения их в распределенный кэш. Настройте скользящий срок хранения 5 секунд и абсолютный срок — 20 секунд:

```
private Product[]? GetDiscontinuedProductsFromDatabase()
{
  Product[]? cachedValue = _db.Products
    .Where(product => product.Discontinued)
    .ToArray();

  DistributedCacheEntryOptions cacheEntryOptions = new()
  {
    // Разрешение читателям сбросить время хранения записи в кэше
    SlidingExpiration = TimeSpan.FromSeconds(5),

    // Установка абсолютного времени хранения записи в кэше
    AbsoluteExpirationRelativeToNow = TimeSpan.FromSeconds(20),
  };

  byte[]? cachedValueBytes =
    JsonSerializer.SerializeToUtf8Bytes(cachedValue);

  _distributedCache.Set(DiscontinuedProductsKey,
    cachedValueBytes, cacheEntryOptions);

  return cachedValue;
}
```

5. В том же файле в методе действия `GetDiscontinuedProducts` добавьте код для попытки извлечь товары со скидкой из кэша. Если они не кэшированы, загрузите их из БД и добавьте в кэш. Байтовый массив из кэша попробуйте десериализовать в товары. В случае неудачи извлеките товары из БД:

```
// GET: api/products/discontinued
[HttpGet]
[Route("discontinued")]
[Produces(typeof(Product[]))]
public IEnumerable<Product> GetDiscontinuedProducts()
{
  // Попытка получить кэшированное значение

  byte[]? cachedValueBytes = _distributedCache.Get(DiscontinuedProductsKey);
```

```
    Product[]? cachedValue = null;

    if (cachedValueBytes is null)
    {
      cachedValue = GetDiscontinuedProductsFromDatabase();
    }
    else
    {
      cachedValue = JsonSerializer.Deserialize<Product[]?>(cachedValueBytes);

      if (cachedValue is null)
      {
        cachedValue = GetDiscontinuedProductsFromDatabase();
      }
    }

    return cachedValue ?? Enumerable.Empty<Product>();
  }
```

В отличие от кэша в памяти, способного хранить любые объекты, данные в распределенном кэше сериализуются в массивы байтов, так как их необходимо передавать по сети.

6. Запустите веб-сервис в режиме `https` без отладки.
7. Расположите окна так, чтобы одновременно можно было видеть и веб-страницу, и терминал.
8. На веб-странице Swagger разверните раздел GET /api/product/discontinued.
9. Нажмите кнопку Try it out (Попробовать).
10. Нажмите кнопку Execute (Выполнить) и обратите внимание на то, что EF Core выполняет SQL-запрос для получения товаров.
11. Нажмите кнопку Execute (Выполнить) несколько раз в течение 5 секунд. Обратите внимание, что EF Core не выполняет повторный SQL-запрос, так как товары уже находятся в кэше. Если данные были прочитаны в течение этого времени, они останутся в памяти.
12. Подождите не менее 5 секунд.
13. Вновь нажмите кнопку Execute (Выполнить). Теперь EF Core снова выполняет SQL-запрос, так как данные не были прочитаны в течение пятисекундного скользящего срока хранения.
14. Продолжайте нажимать кнопку Execute (Выполнить) и обратите внимание, что через 20 секунд EF Core выполнит SQL-запрос для обновления данных.
15. Закройте браузер и завершите работу веб-сервера.

Новая абстракция для распределенного кэширования

Команда разработчиков ASP.NET Core работает над новой абстракцией для распределенного кэширования, которая упростит его использование. На момент написания книги ее реализация только планировалась. Возможно, она будет включена в выпуск исправлений, например, в версии 8.1, но более вероятно, что она будет добавлена в платформу .NET 9 или более поздние релизы.

Некоторые методы расширения `GetAsync` и вспомогательные методы были написаны Марком Грейвеллом, имеющим внушительный опыт работы с распределенным кэшированием. Марк известен своим вкладом в экосистему .NET и поддержкой самого популярного пакета для интеграции .NET с Redis. В ожидании официального релиза вы можете ознакомиться с исходным кодом расширений, доступным по адресу https://github.com/mgravell/DistributedCacheDemo/blob/main/DistributedCacheExtensions.cs. Файл содержит лишь 137 строк кода, поэтому его легко интегрировать в разрабатываемые проекты.

Основное отличие новых методов расширения заключается в том, что теперь не нужно вручную вызывать методы `Set` и `SetAsync`, — эти действия абстрагированы внутри новых методов `GetAsync`:

```
// Методы интерфейса IDistributedCache
    objectFromDatabase = GetFromDatabase(...);
    cache.Set(key: "ITEM_KEY", value: objectFromDatabase, options: ...);
    dataFromCache = cache.Get(key: "ITEM_KEY");

// Новые методы расширения
    dataFromCache = await cache.GetAsync(key: "ITEM_KEY",
      getMethod: GetFromDatabase(...), options: ..., cancellation: ...);
```

Кроме того, все новые методы расширения асинхронны и обобщены с типом `T`, который по умолчанию сериализуется как JSON. Однако это поведение можно переопределить, выбрав альтернативные варианты, такие как бинарный формат `protobuf`.

В копилку знаний: подробнее о планах по созданию новых методов расширения можно узнать в блоге Microsoft по ссылке https://devblogs.microsoft.com/dotnet/caching-abstraction-improvements-in-aspnetcore/.

HTTP-кэширование веб-ответов

Распределенное кэширование и кэширование в оперативной памяти применимы к любым приложениям и сервисам, независимо от используемой транспортной технологии, поскольку вся логика работы кэша выполняется на сервере.

В отличие от них кэширование HTTP-ответов основано на обработке HTTP-запросов `GET` и заголовков ответа. Это означает, что оно работает только с при-

ложениями и сервисами, использующими протокол HTTP, например веб-API, минимальные API и OData.

В копилку знаний: подробнее о стандартах HTTP-кэширования можно узнать в официальной спецификации по адресу https://www.rfc-editor.org/rfc/rfc9111.

Чтобы HTTP-ответ мог быть кэширован, должны выполняться следующие условия.

- Запрос должен быть `GET` или `HEAD`. Запросы `POST`, `PUT`, `DELETE` и другие методы не кэшируются.
- Ответ должен содержать код состояния `200 OK`.
- Если запрос содержит заголовок `Authorization`, ответ не кэшируется.
- Если в заголовке `Vary` указано значение * или недопустимые параметры, ответ также не кэшируется.

Заголовки кэширования устанавливаются на веб-сервере, а промежуточные прокси-серверы и клиенты обязаны соблюдать директивы этих заголовков для правильной работы кэширования.

Дельный совет: HTTP-кэширование не всегда полезно для веб-интерфейсов, так как браузеры часто добавляют заголовки запросов, препятствующие его использованию. Для пользовательских интерфейсов лучше применять кэширование вывода, о котором речь пойдет в главе 14.

Для управления кэшированием используется заголовок `Cache-Control`, который поддерживает несколько директив (табл. 9.2).

Таблица 9.2. Распространенные директивы HTTP-заголовка Cache-Control

Директива	Описание
`public`	Разрешает кэширование ответа клиентами и прокси-серверами
`private`	Разрешает кэширование только на стороне клиента
`max-age`	Указывает максимальный возраст ответа в секундах
`no-cache`	Клиент запрашивает новый ответ, игнорируя кэш. Сервер сообщает клиенту и посредникам, что ответ не должен кэшироваться
`no-store`	Запрещает кэширование запроса и ответа

Помимо `Cache-Control`, существуют и другие заголовки, влияющие на кэширование (табл. 9.3).

Таблица 9.3. HTTP-заголовки общего назначения для кэширования

Заголовок	Описание
`Age`	Время (в секундах), прошедшее с момента получения ответа
`Expires`	Абсолютное время, после которого ответ считается устаревшим
`Vary`	Определяет условия, при которых можно использовать кэшированный ответ. Для его отправки все поля должны совпадать. В противном случае отправляется ответ из БД. Например, строка запроса `color`

Например, если клиент запрашивает актуальный список товаров со скидками, сервис может вернуть HTTP-ответ, запрещающий использование устаревших данных из кэша:

```
GET api/products/discontinued
Cache-Control: no-cache
```

А если ответ можно кэшировать только на стороне клиента, но не у посредников, он будет представлен в виде JSON-массива с заголовком:

```
content-type: application/json; charset=utf-8
date: Fri,09 Jun 2023 06:05:13 GMT
server: Kestrel
cache-control: private

[
  {
    "productId": 5,
    "productName": "Chef Anton's Gumbo Mix",
    ...
```

В ASP.NET Core кэширование HTTP-ответов можно настроить с помощью атрибута `[ResponseCache]` (на стороне клиента). Он применяется к методам контроллеров и страницам Razor, а его параметры позволяют управлять кэшированием (табл. 9.4).

Таблица 9.4. Распространенные параметры атрибута [ResponseCache]

Параметр	Описание
`Duration`	Время хранения кэша в секундах
`Location`	Где может храниться кэш: `Any` (`cache-control: public`), `Client` (`cache-control: private`), `None` (`cache-control: no-cache`)
`NoStore`	Полностью запрещает кэширование (`cache-control: no-store`)
`VaryByHeader`	Определяет заголовки, по которым кэшируются разные версии ответа
`VaryByQueryKeys`	Позволяет кэшировать разные версии ответа по ключам запроса

Настроим кэширование в веб-сервисе.

1. В файле `Program.cs` проекта `Northwind.WebApi.Service` после добавления распределенного кэша в память зарегистрируйте промежуточное ПО для кэширования ответов в качестве механизма внедрения зависимостей:

```
builder.Services.AddResponseCaching();
```

2. В том же файле после настройки перенаправления HTTPS добавьте код для использования промежуточного ПО с целью кэширования ответов:

```
app.UseResponseCaching();
```

Дельный совет: если в проекте используется CORS, метод UseCors должен вызываться перед методом UseResponseCaching.

3. В файле `ProductsController.cs` добавьте кэширование для метода `Get(int id)`, используя атрибут `[ResponseCache]`:

```
// GET api/products/5
[HttpGet("{id:int}")]
[ResponseCache(Duration = 5, // Cache-Control: max-age=5
  Location = ResponseCacheLocation.Any, // Cache-Control: public
  VaryByHeader = "User-Agent" // Vary: User-Agent
  )]
public async ValueTask<Product?> Get(int id)
{
  return await _db.Products.FindAsync(id);
}
```

Атрибут [ResponseCache] может применяться к Razor Pages, классам контроллеров MVC и методам действий MVC как для веб-сервисов, так и для сайтов.

4. Запустите веб-сервис в режиме `https` без отладки.
5. В папке `HttpRequests` откройте файл `webapi-get-products.http`.
6. Измените порт на `5091` и отправьте запрос на получение товара с идентификатором `77`:

```
GET {{base_address}}77
```

7. В ответ сервера должны быть включены заголовки кэширования:

```
Response time: 89 ms
Status code: OK (200)
Alt-Svc: h3=":5091"; ma=86400
```

```
Transfer-Encoding: chunked
Vary: User-Agent
Cache-Control: public, max-age=5
Date: Fri, 09 Jun 2023 06:26:45 GMT
Server: Kestrel

Content-Type: application/json; charset=utf-8
Content-Length: 270

-----------------------------------------------
Content:
{
  "productId": 77,
  "productName": "Original Frankfurter grüne Soße",
  "supplierId": 12,
  "categoryId": 2,
  "quantityPerUnit": "12 boxes",
  "unitPrice": 85.0,
  "unitsInStock": 32,
  "unitsOnOrder": 0,
  "reorderLevel": 15,
  "discontinued": false,
  "category": null,
  "orderDetails": [],
  "supplier": null
}
```

8. Закройте браузер и завершите работу веб-сервера.

Дельный совет: включать кэширование стоит только для анонимных запросов. Аутентифицированные запросы и их ответы кэшировать не рекомендуется.

Кэширование — это мощный инструмент для повышения производительности и масштабируемости веб-сервисов. Далее мы рассмотрим, как защитить сервис от сбоев и повысить его отказоустойчивость.

Повышение отказоустойчивости с помощью библиотеки Polly

Polly — это «библиотека .NET для обработки временных сбоев и переходных процессов. Она позволяет разработчикам выражать такие стратегии отказоустойчивости, как Retry, Circuit Breaker, Hedging, Timeout, Rate Limiter и Fallback, в свободной и безопасной для потоков форме». Так говорится в официальном GitHub-репозитории Polly, доступном по адресу https://github.com/App-vNext/Polly.

Временные сбои — это ошибки, вызванные кратковременными проблемами, такими как временная недоступность сервиса или сбои в сети. В распределенных системах критически важно уметь обрабатывать такие сбои, иначе система может стать практически непригодной для работы.

Паттерны Retry и Circuit Breaker

Паттерн *Retry* («Повторная попытка») позволяет клиентам автоматически повторять неудачное действие с расчетом на то, что ошибка исчезнет, если повторить действие после небольшой задержки. Однако важно помнить, что, если реализовать данный паттерн в простой форме, это может усугубить проблему!

Например, если между попытками установлено фиксированное время задержки, то все клиенты, получившие ошибку, будут пытаться повторить операции одновременно, что приведет к перегрузке сервиса. Для избежания этого часто используют экспоненциально увеличивающиеся интервалы между попытками или добавляют случайность с помощью алгоритмов джиттера (рандомизатора).

Паттерн *Circuit Breaker* («Автоматический выключатель») предотвращает дальнейшие вызовы после достижения определенного порога ошибок, предполагая, что отказ работы нельзя считать временным или достаточно краткосрочным, чтобы продолжать повторять попытки.

В копилку знаний: в репозитории Polly на GitHub есть полезная сводная таблица стратегий отказоустойчивости, доступная по ссылке https://github.com/App-vNext/Polly#resilience-strategies.

Определение и применение политик

В проектах .NET, взаимодействующих с ненадежным кодом, можно подключить пакет Polly и определить политики с помощью класса `Policy`. Библиотека Polly используется не внутри ненадежного кода/сервиса, а всеми клиентами, которые к нему обращаются.

Например, если необходимо вызвать два метода, потенциально выбрасывающих исключения (арифметические или пользовательские), и автоматически повторить действие три раза, можно определить политику следующим образом:

```
RetryPolicy policy = Policy
  .Handle<CustomException>().Or<ArithmeticException>()
  .Retry(3);
```

Затем можно использовать эту политику для выполнения методов:

```
policy.Execute(() => GetProducts());
policy.Execute(() => GetCustomers());
```

Каждый вызов метода `Execute` будет отслеживать собственный счетчик попыток, поэтому, если для вызова метода `GetProducts` требуются две попытки, вызов метода `GetCustomers` все равно будет осуществляться тремя попытками.

Для неограниченного количества попыток можно использовать метод `RetryForever`, но его применение не рекомендуется.

Для асинхронных методов существуют аналогичные асинхронные методы, например `RetryAsync` вместо `Retry`.

Кроме того, метод `Retry` может содержать обратный вызов, позволяя выполнять дополнительные действия при попытках, например, для журналирования информации:

```
RetryPolicy policy = Policy
  .Handle<CustomException>().Or<ArithmeticException>()
  .Retry(3, onRetry: (exception, retryCount) =>
  {
    // Запись в журнал текущего значения счетчика попыток
    // и информации об исключениях
  });
```

Настройка интервалов между попытками

Лучше не пытаться сразу повторить попытку после ошибки, а сделать паузу. Это даст время для устранения временных проблем и предотвратит перегрузку системы:

```
RetryPolicy policy = Policy
  .Handle<CustomException>().Or<ArithmeticException>()
  .WaitAndRetry(new[]
  {
    TimeSpan.FromSeconds(1), // 1 секунда между 1-й и 2-й попытками
    TimeSpan.FromSeconds(2), // 2 секунды между 2-й и 3-й попытками
    TimeSpan.FromSeconds(5)  // 5 секунд между 3-й и 4-й попытками
  });
```

Вместо фиксированных значений задержки можно определить функцию для их генерации — более гибкий подход:

```
RetryPolicy policy = Policy
  .Handle<CustomException>().Or<ArithmeticException>()
  .WaitAndRetry(3, retryAttempt =>
    TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)));

// 2 ^ 1 = 2 секунды
// 2 ^ 2 = 4 секунды
// 2 ^ 3 = 8 секунд
```

Однако если передать массив фиксированных задержек, то представьте, что произойдет, когда в перегруженном веб-сервисе случится сбой. Все клиенты получат

исключение, все будут ожидать секунду и затем попытаются вызвать сервис одновременно. Возникнет эффект в духе флуд-атаки, только усугубляющий проблему!

Джиттер помогает избежать этой ситуации, рандомизируя задержки между попытками. Для этого есть различные решения, и одно из самых удобных встроено в библиотеку Polly. Мы воспользуемся им для генерации случайных временных задержек в нашем примере.

Применение политик к HTTP-клиентам

При вызове веб-сервиса рекомендуется настроить фабрику HTTP-клиентов и зарегистрировать ее в коллекции служб зависимостей.

В этом случае вы не будете вызывать методы, которые могут выбрасывать исключения. Вместо этого определите политику и прикрепите ее к зарегистрированному HTTP-клиенту, чтобы тот автоматически следовал этой политике.

Для этого используется класс расширения `HttpPolicyExtensions`, который помогает создавать политики для распространенных HTTP-запросов и ошибок:

```
AsyncRetryPolicy<HttpResponseMessage> retryPolicy = HttpPolicyExtensions
  // Обработка сетевых сбоев, кодов ответа 408 и 5xx
  .HandleTransientHttpError()
  // Определение политики с теми же параметрами
  .RetryAsync(3);
```

Чтобы прикрепить политику к HTTP-клиенту, необходимо вызвать метод расширения `AddPolicyHandler` после настройки фабрики. Как это сделать на практике, показано далее.

Внесение в веб-сервис случайных ошибок

Для демонстрации добавим случайные ошибки в веб-сервис.

1. Откройте файл `ProductsController.cs` в проекте `Northwind.WebApi.Service`. В методе действия `Get`, который принимает параметр `name`, добавьте операторы для рандомного выброса исключения в 2/3 случаев:

```
// GET api/products/cha
[HttpGet("{name}")]
public IEnumerable<Product> Get(string name)
{
  // Работает правильно один раз из трех
  if (Random.Shared.Next(1, 4) == 1)
  {
    return _db.Products.Where(p => p.ProductName.Contains(name));
  }
```

```
    // Во всех остальных случаях выбрасывается исключение
    throw new Exception("Randomized fault.");
  }
```

2. Выполните сборку проекта.

Создание проекта MVC для вызова отказавшего веб-сервиса

Теперь создадим клиент на основе ASP.NET Core MVC, который будет обращаться к веб-сервису со случайными сбоями. На начальном этапе, если веб-сервис выбросит исключение, клиент просто обработает его. Позже мы добавим поддержку переходных ошибок с помощью Polly.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта ASP.NET Core Web App (Model-View-Controller) (Веб-приложение ASP.NET Core (Модель — представление — контроллер)) / `mvc`;
   - в поле Solution name (Имя решения) укажите значение `Chapter09`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.WebApi.Client.Mvc`;
   - другие параметры Visual Studio 2022:
     - в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
     - установите флажок Configure for HTTPS (Настроить для HTTPS);
     - снимите флажок Enable Docker (Включить поддержку контейнера);
     - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня).
2. В папке `Properties` проекта `Northwind.WebApi.Client.Mvc` откройте файл `launchSettings.json`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5093 для протокола HTTPS и 5094 — для HTTP:

```
"applicationUrl": "https://localhost:5093;http://localhost:5094",
```

3. В файле проекта `Northwind.WebApi.Client.Mvc` добавьте строгую обработку предупреждений и добавьте ссылку на проект моделей сущностей, чтобы использовать класс `Product`:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.EntityModels.
SqlServer\Northwind.Common.EntityModels.SqlServer.csproj" />
</ItemGroup>
```

4. Выполните сборку проекта `Northwind.WebApi.Client.Mvc` в терминале, введя команду `dotnet build`.

5. В файле `Program.cs` проекта `Northwind.WebApi.Client.Mvc` импортируйте пространство имен для работы с HTTP-заголовками:

```
using System.Net.Http.Headers; // Для класса MediaTypeWithQualityHeaderValue
```

6. В том же файле перед вызовом метода `builder.Build()` добавьте настройку фабрики HTTP-клиентов для вызова веб-сервиса:

```
builder.Services.AddHttpClient(name: "Northwind.WebApi.Service",
  configureClient: options =>
  {
    options.BaseAddress = new("https://localhost:5091/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
        "application/json", 1.0));
  });
```

7. В папке `Models` создайте файл класса `HomeProductsViewModel.cs`.
8. В созданном файле создайте класс для хранения информации, необходимой для представления, например фрагмента названия товара и списка товаров, с возможным сообщением об ошибке:

```
using Northwind.EntityModels; // Для класса Product

namespace Northwind.WebApi.Client.Mvc.Models;

public class HomeProductsViewModel
{
  public string? NameContains { get; set; }
  public Uri? BaseAddress { get; set; }
  public IEnumerable<Product>? Products { get; set; }
  public string? ErrorMessage { get; set; }
}
```

9. В папке `Controllers` откройте файл `HomeController.cs` и импортируйте в нем пространство имен для моделей сущностей:

```
using Northwind.EntityModels; // Для класса Product
```

10. В этом же файле добавьте приватное поле с модификатором `readonly`, чтобы хранить зарегистрированную фабрику HTTP-клиентов:

```
private readonly ILogger<HomeController> _logger;
private readonly IHttpClientFactory _httpClientFactory;

public HomeController(ILogger<HomeController> logger,
  IHttpClientFactory httpClientFactory)
{
  _logger = logger;
  _httpClientFactory = httpClientFactory;
}
```

11. В том же файле добавьте асинхронный метод действия `Products`, использующий HTTP-фабрику для запроса товаров, название которых содержит значение параметра `name` в маршруте MVC:

```
[Route("home/products/{name?}")]
public async Task<IActionResult> Products(string? name = "cha")
{
  HomeProductsViewModel model = new();

  HttpClient client = _httpClientFactory.CreateClient(
    name: "Northwind.WebApi.Service");

  model.NameContains = name;
  model.BaseAddress = client.BaseAddress;

  HttpRequestMessage request = new(
    method: HttpMethod.Get,
    requestUri: $"api/products/{name}");

  HttpResponseMessage response = await client.SendAsync(request);

  if (response.IsSuccessStatusCode)
  {
    model.Products = await response.Content
      .ReadFromJsonAsync<IEnumerable<Product>>();
  }
  else
  {
    model.Products = Enumerable.Empty<Product>();

    string content = await response.Content.ReadAsStringAsync();

    // Использование оператора диапазона .., чтобы начать с нуля
    // и перейти к первому возврату каретки
    string exceptionMessage = content[..content.IndexOf("\r")];

    model.ErrorMessage = string.Format("{0}: {1}:",
      response.ReasonPhrase, exceptionMessage);
  }

  return View(model);
}
```

12. В папке `Views/Home` создайте файл `Products.cshtml`:
    - в Visual Studio 2022 выберите шаблон Razor View — Empty;
    - в JetBrains Rider выберите шаблон Razor MVC View.
13. Измените содержимое файла `Products.cshtml`, указав код для вывода таблицы товаров, чьи названия совпадают с введенным значением:

```
@using Northwind.EntityModels
@model HomeProductsViewModel
@{
  ViewData["Title"] = "Products using Polly";
}
<div class="text-center">
  <h1 class="display-4">@ViewData["Title"]</h1>
    <div class="alert alert-info">
      <p>
        This page calls a web service endpoint that will randomly
        fail two out of three times. It will use Polly to retry
        the call automatically.
      </p>
    </div>

  @if (Model is not null)
  {
    if (!string.IsNullOrWhiteSpace(Model.ErrorMessage))
    {
      <div class="alert alert-danger">
        @Model.ErrorMessage
      </div>
    }
    <form action="/home/products">
      <input name="name" placeholder="Enter part of a product name"
        value="@Model.NameContains" />
      <input type="submit" value="Get Products" />
      @if (!string.IsNullOrWhiteSpace(Model.NameContains))
      {
        <p>
          Searched for product names that start with:
          <span class="badge bg-primary rounded-pill">
          @Model.NameContains</span>
        </p>
      }
    </form>
    <div>
      @if (Model.Products is not null)
      {
        <table class="table">
          <thead>
            <tr>
              <th scope="col">Product Name</th>
            </tr>
          </thead>
          <tbody>
            @if (Model.Products.Any())
            {
              @foreach (Product p in Model.Products)
```

```
                {
                  <tr>
                    <td>
                      <a href="@(Model.BaseAddress)api/products/@p.ProductId">
                        @p.ProductName</a>
                    </td>
                  </tr>
                }
              }
              else
              {
                <tr><td>0 products found.</td></tr>
              }
            </tbody>
          </table>
        }
      </div>
    }
  </div>
```

14. В каталоге `Views/Home` откройте файл `Index.cshtml` и добавьте в него ссылку на страницу товаров:

```
<p><a href="home/products">Search for products by name</a></p>
```

15. Запустите проект `Northwind.WebApi.Service` в режиме `https` без отладки.
16. Запустите проект `Northwind.WebApi.Client.Mvc` в режиме `https` без отладки.

В программе Visual Studio Code: браузер не запускается автоматически. Откройте Chrome и перейдите по адресу https://localhost:5093.

17. На главной странице нажмите кнопку Search for products by name.
18. Если поиск успешен, вы увидите результаты, как на рис. 9.1.

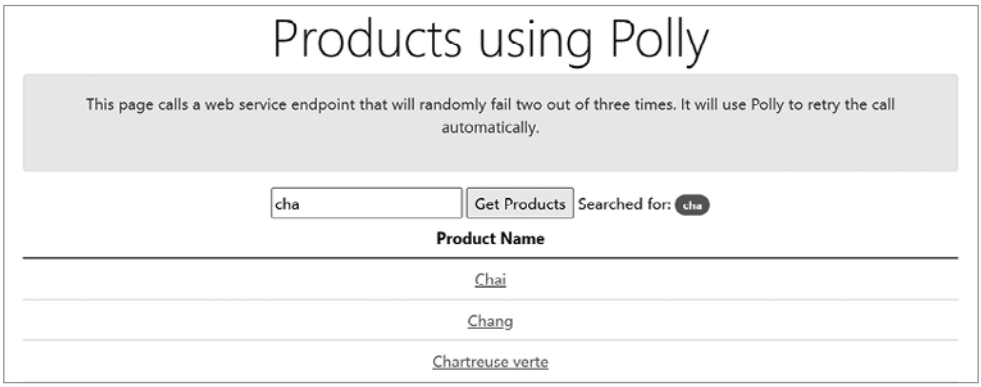


Рис. 9.1. Успешный вызов веб-сервиса со случайными отказами

19. В случае отказа вы увидите сообщение об ошибке (рис. 9.2).

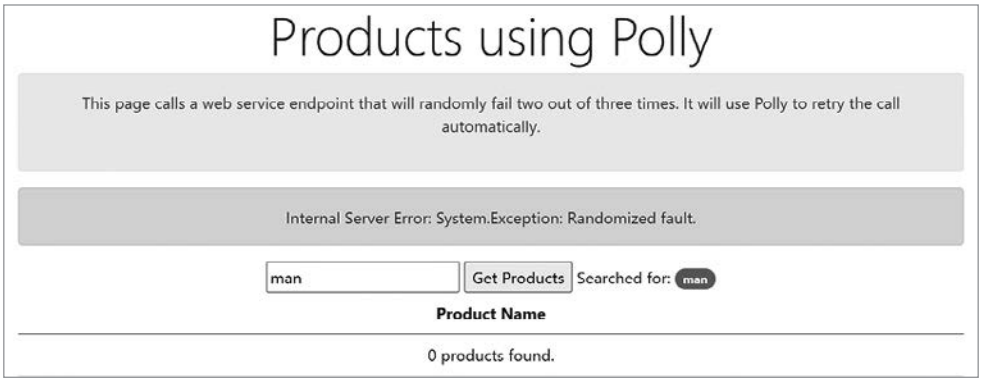


Рис. 9.2. Неудачный вызов веб-сервиса со случайными отказами

20. В терминале при возникновении ошибки будет выведено исключение (ниже показан фрагмент):

```
fail: Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware[1]
      An unhandled exception has occurred while executing the request.
      System.Exception: Randomized fault.
```

21. Вводите разные части названий и нажимайте кнопку Get Products, анализируя успешные и неудачные результаты поиска.
22. Закройте браузер и завершите работу веб-сервера.

Реализация шаблона Retry для обработки временных сбоев

Настроив веб-сервис и клиент со случайными ошибками, давайте реализуем обработку временных отказов с помощью шаблона Retry.

1. В файле проекта `Northwind.WebApi.Client.Mvc` статически импортируйте класс `System.Console` на глобальном уровне. Добавьте ссылку на пакет Microsoft для интеграции библиотеки Polly в ASP.NET Core (с зависимостью от пакета `Polly`). Это необходимо для добавления случайных интервалов (джиттера) между попытками:

```
<ItemGroup>
  <PackageReference Include="Microsoft.Extensions.Http.Polly"
                    Version="8.0.0" />
  <PackageReference Include="Polly.Contrib.WaitAndRetry"
                    Version="1.1.1" />
</ItemGroup>
```

2. Выполните сборку проекта `Northwind.WebApi.Client.Mvc` для восстановления зависимостей.
3. В файле `Program.cs` импортируйте пространство имен `Polly`, которое будет использоваться для обработки ошибок и выполнения попыток:

```
using Polly; // Для метода AddTransientHttpErrorPolicy
using Polly.Contrib.WaitAndRetry; // Для техники Backoff
using Polly.Extensions.Http; // Для пакета HttpPolicyExtensions
using Polly.Retry; // Для политики AsyncRetryPolicy<T>
```

4. В файле `Program.cs` перед добавлением HTTP-клиента в сервисы настройте генерацию пяти случайных интервалов. Эти интервалы должны увеличиваться прерывисто и экспоненциально. Выведите их в консоль для отслеживания. Используйте эти интервалы для настройки асинхронной политики ожидания и попыток. Затем добавьте политику попыток в фабрику HTTP-клиентов:

```
// Создание пяти джиттерных задержек начиная примерно с 1 секунды
IEnumerable<TimeSpan> delays = Backoff.DecorrelatedJitterBackoffV2(
  medianFirstRetryDelay: TimeSpan.FromSeconds(1), retryCount: 5);

WriteLine("Jittered delays for Polly retries:");
foreach (TimeSpan item in delays)
{
  WriteLine($" {item.TotalSeconds:N2} seconds.");
}

AsyncRetryPolicy<HttpResponseMessage> retryPolicy = HttpPolicyExtensions
  // Обработка сетевых сбоев, кодов ответа 408 и 5xx
  .HandleTransientHttpError().WaitAndRetryAsync(delays);

builder.Services.AddHttpClient(name: "Northwind.WebApi.Service",
  configureClient: options =>
  {
    options.BaseAddress = new("https://localhost:5091/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
        "application/json", 1.0));
  })
  .AddPolicyHandler(retryPolicy);
```

5. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
6. Запустите проект `Northwind.WebApi.Service` в режиме `https` без отладки.
7. Запустите проект `Northwind.WebApi.Client.Mvc` в режиме `https` без отладки.

8. В терминале для проекта MVC вы увидите временные интервалы:

```
Jittered delays for Polly retries:
  1.38 seconds.
  0.15 seconds.
  2.65 seconds.
  3.06 seconds.
  6.46 seconds.
```

У вас пять задержек будут иными, первая составит примерно 1 секунду, а последующие будут увеличиваться.

9. Разместите окна терминала с веб-сервисом и браузера с сайтом MVC так, чтобы видеть их одновременно.
10. На главной странице нажмите кнопку Search for products by name.
11. Обратите внимание, что для отображения страницы сайт MVC может выполнить несколько запросов. Обычно это занимает около 15 секунд. В моем случае сайт MVC сделал четыре неудачные попытки, прежде чем успешно завершил пятый запрос. В процессе вы увидите исключения, зарегистрированные в журнале веб-сервиса.
12. Введите фрагмент названия товара, нажмите кнопку Get Products. Страница отобразится, даже если потребуется выполнить несколько запросов.
13. Вероятно, лимит из пяти попыток будет достигнут, и в таком случае вы вновь увидите сообщение об ошибке, как и раньше.

Сотрудники корпорации Microsoft разработали собственные пакеты — обертки Polly, упрощающие ее использование, — Microsoft.Extensions.Http.Resilience и Microsoft.Extensions.Resilience. Более подробную информацию о них вы найдете на странице https://devblogs.microsoft.com/dotnet/building-resilient-cloud-services-with-dotnet-8/.

Теперь, когда вы познакомились с двумя методами улучшения работы сервисов — кэшированием и обработкой временных отказов, — давайте перейдем к третьей мощной технике — очередям.

Создание очередей с помощью RabbitMQ

Очередь помогает улучшить масштабируемость сервиса по аналогии с реальным миром. Когда излишне много клиентов одновременно обращаются к сервису, очередь позволяет равномерно распределить нагрузку.

Существует множество систем очередей для различных платформ разработки. Одна из самых популярных — RabbitMQ. В ней реализован протокол *Advanced Message Queuing Protocol (AMQP)*.

В протоколе AMQP сообщения публикуются на обменниках (exchange), распределяющих копии сообщений по очередям с помощью привязок (binding). Затем брокер передает сообщения потребителям, подписанным на очередь (иногда называемую топиком), либо потребитель может забирать сообщения самостоятельно.

Поскольку сети и системы подвержены сбоям, в AMQP используются подтверждения сообщений. Потребитель сообщает брокеру об успешной обработке, и только после этого сообщение удаляется из очереди.

Протокол RabbitMQ поддерживает четыре типа обмена.

- *Прямой* (direct) — отправляет сообщения в очередь на основе ключа маршрутизации. К одному обменнику можно привязать несколько очередей, но сообщения доставляются только в те, чей ключ маршрутизации совпадает. Прямой обмен часто используется для одноадресных сообщений. По умолчанию (с пустым именем) используется именно этот тип, он привязан к ключу маршрутизации, который совпадает с именем очереди. В этой книге тоже будет использоваться именно он.
- *Веерный* (fanout) — отправляет сообщения во все очереди, привязанные к обменнику, игнорируя ключ маршрутизации. Такой механизм удобен для широковещательной рассылки.
- *Топик* (topic) — передает сообщения на основе ключа маршрутизации и условий, указанных при привязке очереди. Это полезно в сценариях публикации/подписки, где потребители получают только нужные им сообщения, например по географическому признаку или по категориям интересов.
- *Заголовки* (headers) — маршрутизирует сообщения, используя атрибуты в заголовке, а не ключ маршрутизации.

В API RabbitMQ задействуются следующие основные интерфейсы.

- `IConnection` — AMQP-соединение.
- `ConnectionFactory` — создает экземпляры `IConnection` с настройками по умолчанию. Например: `UserName` = `guest`, `Password` = `guest`, `VirtualHost` = `/`, `HostName` = `localhost`, `Port` = `5672`.
- `IModel` — представляет AMQP-канал. Позволяет, например, объявлять очереди (`QueueDeclare`) и отправлять сообщения (`BasicPublish`).
- `IBasicConsumer` — потребитель сообщений.
- `EventBasicConsumer` — реализация потребителя, интегрированная с системой событий .NET. Позволяет клиентскому приложению обрабатывать сообщения сразу после их поступления.

Дельный совет: системы очередей могут быстро усложняться. В этой книге рассмотрены только основы. Если вы планируете внедрение RabbitMQ в рабочие проекты, изучите дополнительные материалы о его настройке и администрировании.

RabbitMQ можно установить локально, но для удобства лучше использовать контейнер Docker.

Инструкции по установке RabbitMQ на компьютер для разных операционных систем опубликованы по адресу https://www.rabbitmq.com/download.html.

Настройка RabbitMQ с помощью Docker

Образ с контейнером Docker, который будет использован, содержит RabbitMQ версии 3.12.0 с настройками по умолчанию.

В копилку знаний: подробнее об образах Docker можно прочитать по адресу https://hub.docker.com/_/rabbitmq/.

Приступим к работе с RabbitMQ в контейнере Docker.

1. Установите Docker согласно инструкциям по адресу https://docs.docker.com/engine/install/.
2. Запустите приложение Docker.
3. Откройте терминал и найдите последний образ контейнера RabbitMQ. Запустите контейнер, открыв порты 5672 (сервис очереди) и 15 672 (сервис управления очередью), которые по умолчанию используются AMQP:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672
rabbitmq:3.12-management
```

4. При первом выполнении команды, поскольку образ RabbitMQ не был загружен ранее, Docker не обнаружит его на локальном компьютере:

```
Unable to find image 'rabbitmq:3.12-management' locally
```

5. Обратите внимание, что образ будет загружен автоматически:

```
3.12-management: Pulling from library/rabbitmq
99803d4b97f3: Pull complete
8fb904ec525a: Pull complete
```

```
ba4d114a87c0: Pull complete
c869b027f1e1: Pull complete
729c8b3166a8: Pull complete
7de098b90abf: Pull complete
4f206ad5199f: Pull complete
1f40437d763f: Pull complete
f4cbf27a2d68: Pull complete
5a4db5ea38b2: Pull complete
99886074092c: Pull complete
Digest:
sha256:da98d468cf2236171da94e34953619ddd01b5db155ee326b653675d1e5017f0f
Status: Downloaded newer image for rabbitmq:3.12-management
```

6. При запуске контейнера указывается, что RabbitMQ работает на языке Erlang, а также приведена информация об авторских правах и лицензии:

```
2023-06-11 14:03:22.785019+00:00 [info] <0.230.0> Starting RabbitMQ 3.12.0 on
Erlang 25.3.2.2 [jit]
2023-06-11 14:03:22.785019+00:00 [info] <0.230.0> Copyright (c) 2007-2023
VMware, Inc. or its affiliates.
2023-06-11 14:03:22.785019+00:00 [info] <0.230.0> Licensed under the MPL 2.0.
Website: https://rabbitmq.com

  ##  ##      RabbitMQ 3.12.0
  ##  ##
  ##########  Copyright (c) 2007-2023 VMware, Inc. or its affiliates.
  ######  ##
  ##########  Licensed under the MPL 2.0. Website: https://rabbitmq.com
```

7. Убедитесь, что сервис RabbitMQ прослушивает порт 5672 и запустил необходимые плагины:

```
2023-06-11 14:03:27.574844+00:00 [info] <0.744.0> started TCP listener on
[::]:5672 completed with 4 plugins.
2023-06-11 14:03:27.659139+00:00 [info] <0.599.0> Server startup complete; 4
plugins started.
2023-06-11 14:03:27.659139+00:00 [info] <0.599.0> * rabbitmq_prometheus
2023-06-11 14:03:27.659139+00:00 [info] <0.599.0> * rabbitmq_management
2023-06-11 14:03:27.659139+00:00 [info] <0.599.0> * rabbitmq_web_dispatch
2023-06-11 14:03:27.659139+00:00 [info] <0.599.0> * rabbitmq_management_agent
```

8. Не закрывайте терминал.
9. (Опционально.) В приложении Docker Desktop можно убедиться, что контейнер запущен и использует порты 5672 (текущий сервис очереди) и 15 672 (сервис управления очередью) (рис. 9.3).

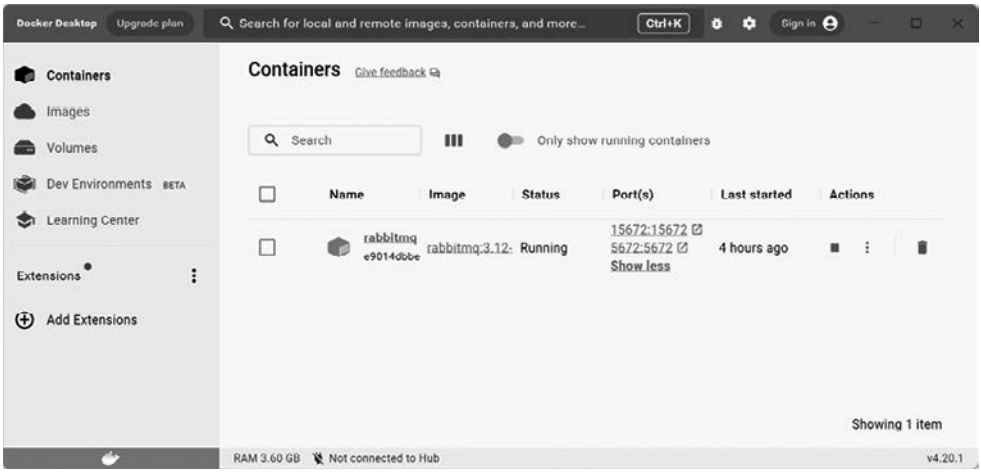


Рис. 9.3. RabbitMQ, запущенный в контейнере Docker

Отправка сообщений в очередь с сайта MVC

Теперь, когда система RabbitMQ запущена, можно добавить клиентский пакет RabbitMQ в проект сайта MVC, чтобы отправлять сообщения в очередь.

Но сначала создадим библиотеку классов для моделей, необходимых для реализации очереди.

1. В редакторе кода создайте проект библиотеки классов со следующими настройками:
   - выберите шаблон проекта Class Library (Библиотека классов);
   - в поле Solution name (Имя решения) укажите значение `Chapter09`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Queue.Models`.
2. Добавьте ссылку на проект моделей сущностей Northwind для SQL Server, созданный в главе 3:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.EntityModels.
SqlServer\Northwind.Common.EntityModels.SqlServer.csproj" />
</ItemGroup>
```

3. Выполните сборку проекта в терминале, чтобы убедиться, что библиотеки классов и модели сущностей компилируются корректно:

```
dotnet build
```

4. Удалите файл `Class1.cs`.
5. Создайте файл `ProductQueueMessage.cs`.
6. В созданном файле определите класс для сообщений в очереди. Класс должен содержать простое текстовое свойство и свойство сложного типа модели сущности `Product`:

```
using Northwind.EntityModels; // Для класса Product

namespace Northwind.Queue.Models;

public class ProductQueueMessage
{
  public string? Text { get; set; }
  public Product Product { get; set; } = null!;
}
```

7. В файле проекта `Northwind.WebApi.Client.Mvc` добавьте ссылку на проект `Northwind.Queue.Models`, чтобы использовать класс `ProductQueueMessage`:

```
<ItemGroup>
  <ProjectReference Include="..\Northwind.Queue.Models\Northwind.Queue.Models.
  ➥ csproj" />
</ItemGroup>
```

8. В файле проекта `Northwind.WebApi.Client.Mvc` подключите клиентский пакет для работы с RabbitMQ:

```
<PackageReference Include="RabbitMQ.Client" Version="6.7.0" />
```

Актуальную версию пакета можно узнать по адресу https://www.nuget.org/packages/RabbitMQ.Client/.

9. Выполните сборку проекта `Northwind.WebApi.Client.Mvc`.
10. В папке Models создайте файл класса `HomeSendMessageViewModel.cs`.
11. Определите в нем класс для представления информации, которая будет отображаться в интерфейсе отправки сообщений. Добавьте два свойства:
    - для вывода сообщения об успешной отправке;
    - для вывода ошибки при возникновении проблем:

```
using Northwind.Queue.Models; // Для класса ProductQueueMessage

namespace Northwind.WebApi.Client.Mvc.Models;

public class HomeSendMessageViewModel
{
```

```
    public string? Info { get; set; }
    public string? Error { get; set; }
    public ProductQueueMessage? Message { get; set; }
  }
```

12. В файле `Index.cshtml` (в папке `Views\Home`) добавьте ссылку на страницу отправки сообщений в очередь:

```
<p><a href="home/sendmessage">Send a message</a></p>
```

13. В файле `HomeControllers.cs` импортируйте пространство имен для работы с RabbitMQ и сериализации JSON:

```
using RabbitMQ.Client;  // Для класса ConnectionFactory и т. д.
using System.Text.Json; // Для класса JsonSerializer
```

14. В том же файле добавьте метод для обработки `GET`-запроса, который отобразит веб-форму отправки сообщений:

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

15. В том же файле добавьте метод для обработки `POST`-запроса, который отправит сообщение в очередь:

```
// POST: home/sendmessage
// Body: message=Hello&productId=1
[HttpPost]
public async Task<IActionResult> SendMessage(
  string? message, int? productId)
{
  HomeSendMessageViewModel model = new();
  model.Message = new();

  if (message is null || productId is null)
  {
    model.Error = "Please enter a message and a product ID.";
    return View(model);
  }

  model.Message.Text = message;

  model.Message.Product = new() { ProductId = productId.Value };

  HttpClient client = _httpClientFactory.CreateClient(
    name: "Northwind.WebApi.Service");

  HttpRequestMessage request = new(
```

```
      method: HttpMethod.Get,
      requestUri: $"api/products/{productId}");

    HttpResponseMessage response = await client.SendAsync(request);

    if (response.IsSuccessStatusCode)
    {
      Product? product = await response.Content.ReadFromJsonAsync<Product>();

      if (product is not null)
      {
        model.Message.Product = product;
      }
    }

    // Создание фабрики RabbitMQ
    ConnectionFactory factory = new() { HostName = "localhost" };

    using IConnection connection = factory.CreateConnection();

    using IModel channel = connection.CreateModel();

    string queueNameAndRoutingKey = "product";

    // Если очередь не существует, она будет создана.
    // Если контейнер Docker будет перезапущен, очередь будет потеряна.
    // Очередь может быть общей для нескольких потребителей.
    // Очередь не будет удалена, если последнее сообщение в ней клиентское.

    channel.QueueDeclare(queue: queueNameAndRoutingKey, durable: false,
      exclusive: false, autoDelete: false, arguments: null);

    byte[] body = JsonSerializer.SerializeToUtf8Bytes(model.Message);

    // Обменник пуст, потому что мы используем обменник по умолчанию
    channel.BasicPublish(exchange: string.Empty,
      routingKey: queueNameAndRoutingKey,
      basicProperties: null, body: body);

    model.Info = "Message sent to queue successfully.";

    return View(model);
  }
```

16. В каталоге `Views\Home` создайте Razor-представление `SendMessage.cshtml`.
17. Определите веб-страницу с формой для отправки сообщения:

```
@model HomeSendMessageViewModel
@{
  ViewData["Title"] = "Send a Message";
}
```

```
<div class="text-center">
  <h1 class="display-4">@ViewData["Title"]</h1>

  @if (Model is not null)
  {
    if (Model.Error is not null)
    {
      <div class="alert alert-danger">
        <h2>Error</h2>
        <p>@Model.Error</p>
      </div>
    }
    if (Model.Info is not null)
    {
      <div class="alert alert-info">
        <h2>Information</h2>
        <p>@Model.Info</p>
      </div>
    }
  }
  <form asp-controller="Home" asp-action="SendMessage" method="post">
    <div>
      <label for="message">Message</label>
      <input id="message" name="message" />
    </div>
    <div>
      <label for="productId">Product ID</label>
      <input id="productId" name="productId" />
    </div>
    <input type="submit" value="Send" />"
  </form>
</div>
```

Консольное приложение для обработки сообщений в очереди

Теперь можно создать консольное приложение для обработки сообщений в очереди.

1. В редакторе кода создайте консольное приложение со следующими настройками:
   - выберите шаблон проекта Console App (Консольное приложение);
   - в поле Solution name (Имя решения) укажите значение `Chapter09`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Queue.Consumer`.
2. В файле проекта добавьте строгую обработку предупреждений, подключите пакет RabbitMQ и ссылки на проекты `Northwind` (модели сущностей)

и `Northwind.Queue.Models` (модели сообщений). Также статически импортируйте класс `System.Console` на глобальном уровне:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="RabbitMQ.Client" Version="6.7.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include=
      "..\..\Chapter03\Northwind.Common.EntityModels.
         SqlServer\Northwind.Common.EntityModels.SqlServer.csproj" />
    <ProjectReference Include=
      "..\Northwind.Queue.Models\Northwind.Queue.Models.csproj" />
  </ItemGroup>

  <ItemGroup>
    <Using Include="System.Console" Static="true" />
  </ItemGroup>

</Project>
```

3. В терминале выполните сборку проекта:

```
dotnet build
```

4. В файле `Program.cs` удалите весь код, а затем добавьте логику для чтения сообщений из очереди товаров:

```
using Northwind.Queue.Models; // Для класса ProductQueueMessage
using RabbitMQ.Client; // Для класса ConnectionFactory
using RabbitMQ.Client.Events; // Для класса EventingBasicConsumer
using System.Text.Json; // Для класса JsonSerializer

string queueName = "product";

ConnectionFactory factory = new () { HostName = "localhost" };

using IConnection connection = factory.CreateConnection();

using IModel channel = connection.CreateModel();
```

```
WriteLine("Declaring queue...");

QueueDeclareOk response = channel.QueueDeclare(
  queue: queueName,
  durable: false,
  exclusive: false,
  autoDelete: false,
  arguments: null);

WriteLine("Queue name: {response.QueueName}, Message count: {
  response.MessageCount}, Consumer count: {response.ConsumerCount}.");

WriteLine("Waiting for messages...");

EventingBasicConsumer consumer = new(channel);

consumer.Received += (model, args) =>
{
  byte[] body = args.Body.ToArray();

  ProductQueueMessage? message = JsonSerializer
    .Deserialize<ProductQueueMessage>(body);

  if (message is not null)
  {
    WriteLine("Received product. Id: {message.Product.ProductId},
      Name: { message.Product.ProductName}, Message: {message.Text}");
  }
  else
  {
    WriteLine($"Received unknown: {args.Body.ToArray()}.");
  }
};

// Потребление начинается по мере поступления сообщений в очередь
channel.BasicConsume(queue: queueName,
  autoAck: true,
  consumer: consumer);

WriteLine(">>> Press Enter to stop consuming and quit. <<<");
ReadLine();
```

5. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
6. Запустите проект `Northwind.WebApi.Service` в режиме `https` без отладки.
7. Запустите проект `Northwind.WebApi.Client.Mvc` в режиме `https` без отладки.
8. Запустите консольное приложение `Northwind.Queue.Consumer` (с отладкой или без нее). Можно настроить запуск всех трех проектов одновременно, как показано для Visual Studio 2022 на рис. 9.4.

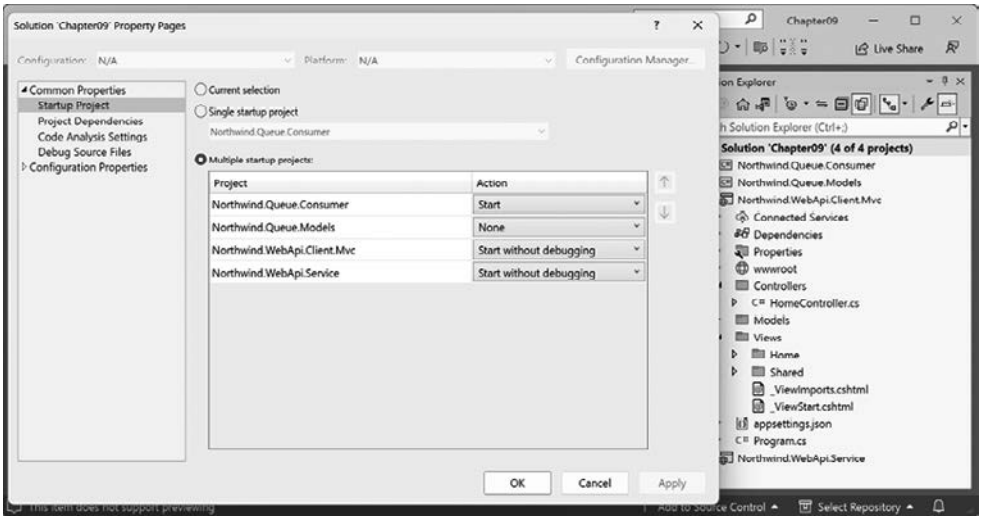


Рис. 9.4. Настройка запуска трех проектов для тестирования очередей сообщений

9. Расположите окна консольного приложения и веб-страницы MVC так, чтобы оба были видны. Затем нажмите кнопку Send a message и введите текст сообщения и идентификатор товара (от 1 до 77). Пример интерфейса отправки сообщения показан на рис. 9.5.

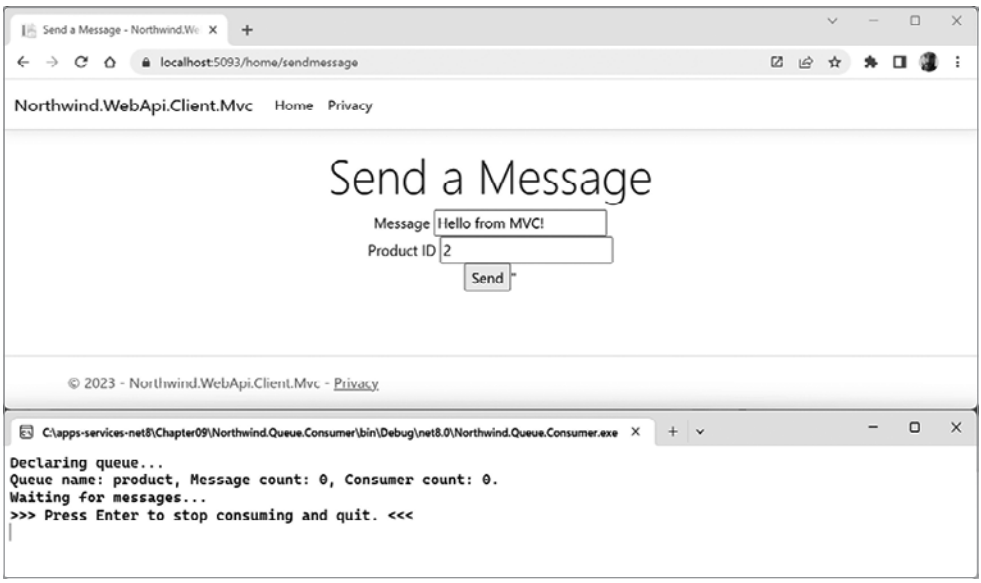


Рис. 9.5. Сайт ASP.NET Core MVC, отправляющий сообщение в очередь

10. Нажмите кнопку Send и обратите внимание на сообщение, появившееся в консольном приложении (рис. 9.6).

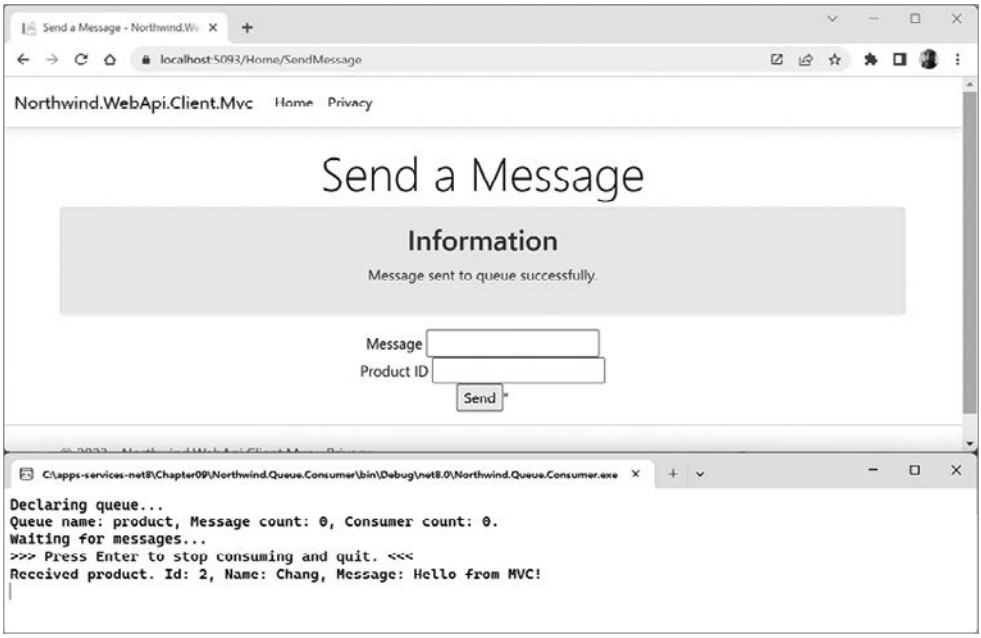


Рис. 9.6. Консольное приложение с сообщением, полученным из очереди

11. В терминале с Docker нажмите Ctrl+C, чтобы выключить контейнер, и обратите внимание на результат:

```
2023-06-11 17:42:31.006172+00:00 [info] <0.744.0> stopped TCP listener on
[::]:5672
2023-06-11 17:42:31.008574+00:00 [info] <0.1552.0> Closing all connections in
vhost '/' on node 'rabbit@e9014dbbe5f5' because the vhost is stopping
2023-06-11 17:42:31.017407+00:00 [info] <0.557.0> Stopping message store for
directory '/var/lib/rabbitmq/mnesia/rabbit@e9014dbbe5f5/msg_stores/vhosts/
628WB79CIFDYO9LJI6DKMI09L/msg_store_persistent'
2023-06-11 17:42:31.024661+00:00 [info] <0.557.0> Message store for directory
'/var/lib/rabbitmq/mnesia/rabbit@e9014dbbe5f5/msg_stores/vhosts/
628WB79CIFDYO9LJI6DKMI09L/msg_store_persistent' is stopped
2023-06-11 17:42:31.024937+00:00 [info] <0.553.0> Stopping message store for
directory '/var/lib/rabbitmq/mnesia/rabbit@e9014dbbe5f5/msg_stores/vhosts/
628WB79CIFDYO9LJI6DKMI09L/msg_store_transient'
2023-06-11 17:42:31.031218+00:00 [info] <0.553.0> Message store for directory
'/var/lib/rabbitmq/mnesia/rabbit@e9014dbbe5f5/msg_stores/vhosts/
628WB79CIFDYO9LJI6DKMI09L/msg_store_transient' is stopped
2023-06-11 17:42:31.037584+00:00 [info] <0.489.0> Management plugin: to stop
collect_statistics.
```

12. В приложении Docker Desktop контейнер исчезнет из списка, но образ останется для быстрого запуска в следующий раз.

В копилку знаний: подробнее об использовании RabbitMQ в .NET можно прочитать по адресу https://www.rabbitmq.com/dotnet-api-guide.html.

Сочетание кэширования, постановки в очередь и обработки временных отказов позволяет сделать ваши сервисы более отказоустойчивыми, масштабируемыми и производительными. В последнем разделе этой главы мы рассмотрим фоновые сервисы с длительным периодом работы.

Реализация фоновых сервисов с длительным временем работы

Как правило, фоновые сервисы этого типа используются для таких операций, как:

- выполнение задания по расписанию;
- обработка сообщений в очереди;
- выполнение ресурсоемких операций, например моделирование ИИ, машинное обучение или обработка видео/изображений.

Ранее в операционной системе Windows для того, чтобы код работал в фоновом режиме, необходимо было создать службу *Windows*. Например, ядро БД SQL Server реализовано как служба Windows. С переходом на кросс-платформенность .NET появилась необходимость в соответствующем решении для запуска кода в фоновом режиме.

Фоновые сервисы часто не имеют пользовательского интерфейса, хотя могут предоставлять его для настройки и управления службой.

Создание Worker Service

Давайте создадим проект службы рабочих процессов (Worker Service) и посмотрим, как можно разместить фоновый сервис с длительным временем работы.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта Worker Service / `worker`;
   - в поле Solution name (Имя решения) укажите значение `Chapter09`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Background.Workers`;

- снимите флажок Enable Docker (Включить поддержку контейнера);
- снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
- снимите флажок Enable Native AOT publishing (Включить публикацию Native AOT).

2. Откройте файл созданного проекта и обратите внимание, что используется .NET SDK — пакет `Microsoft.NET.Sdk.Worker`. Внесите в файл следующие изменения:
   - включите строгую обработку предупреждений;
   - подключите пакет для RabbitMQ;
   - добавьте ссылки на проекты моделей сущностей и очередей:

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <UserSecretsId>dotnet-Northwind.Background.Workers-66434cdf-0fdd-4993-a399-
    ➥ ec9581b4b914</UserSecretsId>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0" />
    <PackageReference Include="RabbitMQ.Client" Version="6.7.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include=
      "..\..\Chapter03\Northwind.Common.EntityModels.SqlServer\Northwind.
      ➥ Common.EntityModels.SqlServer.csproj" />
    <ProjectReference Include=
      "..\Northwind.Queue.Models\Northwind.Queue.Models.csproj" />
  </ItemGroup>

</Project>
```

3. В терминале выполните сборку проекта `Northwind.Background.Workers`, введя команду `dotnet build`.
4. В файле `Program.cs` код инициализации проекта напоминает код из ASP.NET Core, а Worker Service при создании сначала регистрируется, а затем запускается хостом:

```
using Northwind.Background.Workers;

var builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<Worker>();

var host = builder.Build();
host.Run();
```

5. В файле `Worker.cs` класс `Worker` наследуется от класса `BackgroundService` и реализует метод `ExecuteAsync`, который выполняет цикл до запроса на отмену, регистрируя текущие дату и время, а затем делает паузу на одну секунду:

```
namespace Northwind.Background.Workers
{
  public class Worker : BackgroundService
  {
    private readonly ILogger<Worker> _logger;

    public Worker(ILogger<Worker> logger)
    {
      _logger = logger;
    }

    protected override async Task ExecuteAsync(
      CancellationToken stoppingToken)
    {
      while (!stoppingToken.IsCancellationRequested)
      {
        _logger.LogInformation("Worker running at: {time}",
          DateTimeOffset.Now);

        await Task.Delay(1000, stoppingToken);
      }
    }
  }
}
```

6. Запустите проект без отладки. Обратите внимание, что текущее время выводится каждую секунду. Чтобы остановить службу рабочего процесса, нажмите Ctrl+C:

```
info: Northwind.Queue.Worker.Worker[0]
      Worker running at: 06/12/2023 08:25:02 +01:00
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: C:\apps-services-net8\Chapter09\
      Northwind.Queue.Worker
info: Northwind.Queue.Worker.Worker[0]
      Worker running at: 06/12/2023 08:25:03 +01:00
info: Northwind.Queue.Worker.Worker[0]
      Worker running at: 06/12/2023 08:25:04 +01:00
info: Northwind.Queue.Worker.Worker[0]
      Worker running at: 06/12/2023 08:25:05 +01:00
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
```

Обработка сообщений в очереди с помощью Worker Service

Теперь мы можем реализовать обработку сообщений, поставленных в очередь RabbitMQ.

1. Переименуйте файл `Worker.cs` в `QueueWorker.cs`, а класс `Worker` — в `QueueWorker`.
2. В файле `Program.cs` измените имя класса зарегистрированного сервиса с `Worker` на `QueueWorker`:

```
builder.Services.AddHostedService<QueueWorker>();
```

3. В файле `QueueWorker.cs` подключите необходимые пространства имен для работы с очередями RabbitMQ и реализуйте обработчик сообщений:

```
using Northwind.Queue.Models; // Для класса ProductQueueMessage
using RabbitMQ.Client; // Для класса ConnectionFactory
using RabbitMQ.Client.Events; // Для класса EventingBasicConsumer
using System.Text.Json; // Для класса JsonSerializer

namespace Northwind.Background.Workers;

public class QueueWorker : BackgroundService
{
  private readonly ILogger<QueueWorker> _logger;

  // Объекты RabbitMQ
  private const string queueNameAndRoutingKey = "product";
  private readonly ConnectionFactory _factory;
  private readonly IConnection _connection;
  private readonly IModel _channel;
  private readonly EventingBasicConsumer _consumer;

  public QueueWorker(ILogger<QueueWorker> logger)
  {
    _logger = logger;

    _factory = new() { HostName = "localhost" };
    _connection = _factory.CreateConnection();
    _channel = _connection.CreateModel();
    _consumer = new(_channel);

    _channel.QueueDeclare(queue: queueNameAndRoutingKey, durable: false,
      exclusive: false, autoDelete: false, arguments: null);

    _consumer = new(_channel);

    _consumer.Received += (model, args) =>
    {
      byte[] body = args.Body.ToArray();
```

```
        ProductQueueMessage? message = JsonSerializer
          .Deserialize<ProductQueueMessage>(body);

        if (message is not null)
        {
          _logger.LogInformation($"Received product. Id: {
            message.Product.ProductId}, Name: {message.Product.ProductName},
              Message: {message.Text}");
        }
        else
        {
          _logger.LogInformation("Received unknown: {0}.",
            args.Body.ToArray());
        }
      };

      // Запуск потребления сообщений по мере их поступления в очередь
      _channel.BasicConsume(queue: queueNameAndRoutingKey,
        autoAck: true, consumer: _consumer);
    }

    protected override async Task ExecuteAsync(
      CancellationToken stoppingToken)
    {
      while (!stoppingToken.IsCancellationRequested)
      {
        _logger.LogInformation("Worker running at: {time}",
        DateTimeOffset.Now);

        await Task.Delay(3000, stoppingToken);
      }
    }
  }
```

4. Запустите контейнер RabbitMQ с помощью следующей команды:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672
rabbitmq:3.12-management
```

5. Дождитесь сообщения о готовности к подключению клиентов на порт 5672:

```
2023-06-12 08:50:16.591574+00:00 [info] <0.599.0> Ready to start client
connection listeners
2023-06-12 08:50:16.593090+00:00 [info] <0.744.0> started TCP listener on
[::]:5672
```

6. Не закрывайте терминал.
7. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.

8. Запустите проект `Northwind.WebApi.Service` без отладки, чтобы иметь возможность запрашивать товары из БД Northwind.
9. Запустите проект `Northwind.WebApi.Client.Mvc` без отладки, чтобы отправлять сообщения в очередь RabbitMQ.
10. На сайте MVC нажмите кнопку `Send a message`, а затем введите сообщение `apples` и идентификатор товара `1`.
11. Повторите процедуру с товарами `bananas` и `2`, а также `cherries` и `3`.
12. Запустите проект `Northwind.Background.Workers` без отладки и убедитесь, что три сообщения обрабатываются из очереди:

```
info: Northwind.Background.Workers.QueueWorker[0]
      Queue product is waiting for messages.
info: Northwind.Background.Workers.QueueWorker[0]
      Worker running at: 06/12/2023 09:58:59 +01:00
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: C:\apps-services-net8\Chapter09\
      Northwind.Queue.Worker
info: Northwind.Background.Workers.QueueWorker[0]
      Received product. Id: 1, Name: Chai, Message: apples
info: Northwind.Background.Workers.QueueWorker[0]
      Received product. Id: 2, Name: Chang, Message: bananas
info: Northwind.Background.Workers.QueueWorker [0]
      Received product. Id: 3, Name: Aniseed Syrup, Message: cherries
```

Выполнение кода по расписанию

Другое распространенное применение фоновых сервисов — выполнение задач по таймеру. Для реализации таймеров можно использовать класс `System.Threading.Timer`, который запускает метод `DoWork`.

Добавим еще один сервис в проект.

1. В проекте `Northwind.Background.Workers` создайте класс `TimerWorker.cs`.
2. Реализуйте класс `TimerWorker`, добавив необходимую логику для работы с таймером:

```
namespace Northwind.Background.Workers;

public class TimerWorker : IHostedService, IAsyncDisposable
{
  private readonly ILogger<TimerWorker> _logger;
```

```
  private int _executionCount = 0;
  private Timer? _timer;
  private int _seconds = 5;

  public TimerWorker(ILogger<TimerWorker> logger)
  {
    _logger = logger;
  }

  private void DoWork(object? state)
  {
    int count = Interlocked.Increment(ref _executionCount);

    _logger.LogInformation(
      "{0} is working, execution count: {1:#,0}",
      nameof(TimerWorker), count);
  }

  public Task StartAsync(CancellationToken cancellationToken)
  {
    _logger.LogInformation("{0} is running.", nameof(TimerWorker));

    _timer = new Timer(callback: DoWork, state: null,
      dueTime: TimeSpan.Zero,
      period: TimeSpan.FromSeconds(_seconds));

    return Task.CompletedTask;
  }

  public Task StopAsync(CancellationToken cancellationToken)
  {
    _logger.LogInformation("{0} is stopping.", nameof(TimerWorker));

    _timer?.Change(dueTime: Timeout.Infinite, period: 0);

    return Task.CompletedTask;
  }

  public async ValueTask DisposeAsync()
  {
    if (_timer is IAsyncDisposable asyncTimer)
    {
      await asyncTimer.DisposeAsync();
    }

    _timer = null;
  }
}
```

3. В файле `Program.cs` добавьте код для регистрации нового сервиса с таймером:

```
builder.Services.AddHostedService<TimerWorker>();
```

4. Запустите проект `Northwind.Background.Workers` без отладки и обратите внимание, как инициализируются оба рабочих процесса:

```
info: Northwind.Background.Workers.QueueWorker[0]
      Worker running at: 06/12/2023 12:58:25 +01:00
info: Northwind.Background.Workers.TimerWorker[0]
      TimerWorker is running.
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: C:\apps-services-net8\Chapter09\
      Northwind.Background.Workers
```

5. Не останавливайте фоновые рабочие процессы как минимум 10 секунд и обратите внимание, что рабочий процесс обработки очереди выводит сообщения каждую секунду, а рабочий процесс с таймером — каждые 5 секунд:

```
info: Northwind.Background.Workers.TimerWorker[0]
      TimerWorker is working, execution count: 1
info: Northwind.Background.Workers.QueueWorker[0]
      Worker running at: 06/12/2023 12:58:26 +01:00
info: Northwind.Background.Workers.QueueWorker[0]
      Worker running at: 06/12/2023 12:58:27 +01:00
info: Northwind.Background.Workers.QueueWorker[0]
      Worker running at: 06/12/2023 12:58:28 +01:00
info: Northwind.Background.Workers.QueueWorker[0]
      Worker running at: 06/12/2023 12:58:29 +01:00
info: Northwind.Background.Workers.TimerWorker[0]
      TimerWorker is working, execution count: 2
```

6. Для остановки процессов нажмите Ctrl+C, и убедитесь, что процессы завершены корректно:

```
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: Northwind.Background.Workers.TimerWorker[0]
      TimerWorker is stopping.
```

Для большей гибкости в управлении фоновыми процессами можно вместо выполнения задачи через определенные интервалы (например, 5 секунд) настроить систему так, чтобы она ежесекундно проверяла запланированные задачи. Время выполнения задач можно запланировать на определенный момент. Вместо разработки такой инфраструктуры вручную проще использовать стороннюю библиотеку, такую как *Hangfire*, которая предоставляет удобный механизм для планирования задач.

Создание сайта с функционалом Hangfire

Библиотека Hangfire с открытым исходным кодом поддерживает коммерческое использование и предлагает несколько моделей работы.

- *«Добавил в очередь и забыл»* — задачи, которые выполняются однократно сразу после добавления в очередь.
- *Отложенные* — задачи, которые выполняются в определенное время в будущем.
- *Повторяющиеся* — задачи, выполняющиеся по регулярному расписанию (например, по CRON).
- *Продолжающие* — задачи, которые выполняются после завершения родительского задания.
- *Пакетные* — транзакционные задачи (доступны только в платной версии).

В Hangfire используется постоянное хранилище данных, которое можно настроить на работу с различными системами:

- SQL Server;
- Redis;
- оперативная память;
- хранилище, разработанное сообществом.

Для начала работы с Hangfire создадим проект ASP.NET Core.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта ASP.NET Core Empty (Пустой шаблон ASP.NET Core) / `web`;
   - в поле Solution name (Имя решения) укажите значение `Chapter09`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Background.Hangfire`;
   - установите флажок Configure for HTTPS (Настроить для HTTPS);
   - снимите флажок Enable Docker (Включить поддержку контейнера);
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня).
2. В файле проекта добавьте строгую обработку предупреждений и подключите пакеты для работы с Hangfire и SQL Server:

```
<ItemGroup>
  <PackageReference Include="Hangfire.Core" Version="1.8.6" />
  <PackageReference Include="Hangfire.SqlServer" Version="1.8.6" />
  <PackageReference Include="Hangfire.AspNetCore" Version="1.8.6" />
  <PackageReference Include="Microsoft.Data.SqlClient" Version="5.1.2" />
</ItemGroup>
```

3. Выполните сборку проекта для восстановления зависимостей.
4. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5095 для протокола HTTPS и 5096 — для HTTP:

```
"profiles": {
  ...
  "https": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "launchUrl": "swagger",
    "applicationUrl": "https://localhost:5095;http://localhost:5096",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
```

5. В файле `appsettings.Development.json` укажите уровень Hangfire-журналирования `Information`, как показано в следующем JSON-коде:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Hangfire": "Information"
    }
  }
}
```

6. Создайте БД SQL Server с именем `Northwind.HangfireDb`.
   - В Visual Studio 2022 выберите команду меню View ▸ Server Explorer (Вид ▸ Обозреватель серверов), чтобы открыть одноименную панель. На панели Server Explorer (Обозреватель серверов) щелкните правой кнопкой мыши на пункте Data Connections (Подключения к данным) и выберите в контекстном меню команду Create New SQL Server database (Создать новую базу данных SQL Server). В открывшемся диалоговом окне укажите параметры подключения и имя БД, а затем нажмите кнопку OK.
   - В Visual Studio Code подключитесь к SQL Server, щелкните правой кнопкой мыши и выберите пункт New Query (Новый запрос). Настройте параметры подключения, а затем в окне запроса выполните следующую SQL-команду:

```
USE master
GO
CREATE DATABASE [Northwind.HangfireDb]
GO
```

7. Удалите содержимое файла `Program.cs` и добавьте код для настройки Hangfire на работу с SQL Server и включения инструмента Hangfire Dashboard:

```
using Microsoft.Data.SqlClient; // Для класса SqlConnectionStringBuilder
using Hangfire; // Для класса GlobalConfiguration

SqlConnectionStringBuilder connection = new();

connection.InitialCatalog = "Northwind.HangfireDb";
connection.MultipleActiveResultSets = true;
connection.Encrypt = true;
connection.TrustServerCertificate = true;
connection.ConnectTimeout = 5; // По умолчанию 30 секунд
connection.DataSource = "."; // Для локального SQL Server

// Чтобы использовать интегрированную в Windows систему
// проверки подлинности
connection.IntegratedSecurity = true;

/*
// Для проверки подлинности SQL Server
builder.UserID = "sa";
builder.Password = "123456";
builder.PersistSecurityInfo = false;
*/

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHangfire(config => config
  .SetDataCompatibilityLevel(CompatibilityLevel.Version_180)
  .UseSimpleAssemblyNameTypeSerializer()
  .UseRecommendedSerializerSettings()
  .UseSqlServerStorage(connection.ConnectionString));

builder.Services.AddHangfireServer();

var app = builder.Build();

app.UseHangfireDashboard();

app.MapGet("/", () =>
  "Navigate to /hangfire to see the Hangfire Dashboard.");

app.MapHangfireDashboard();

app.Run();
```

8. Запустите проект `Northwind.Background.Hangfire` без отладки и обратите внимание на сообщения, выводимые в консоль:

```
info: Hangfire.SqlServer.SqlServerObjectsInstaller[0]
      Start installing Hangfire SQL objects...
info: Hangfire.SqlServer.SqlServerObjectsInstaller[0]
      Hangfire SQL objects installed.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5095
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5096
info: Hangfire.BackgroundJobServer[0]
      Starting Hangfire Server using job storage:
      'SQL Server: .@Northwind.HangfireDb'
info: Hangfire.BackgroundJobServer[0]
      Using the following options for SQL Server job storage: Queue
      poll interval: 00:00:00.
info: Hangfire.BackgroundJobServer[0]
      Using the following options for Hangfire Server:
          Worker count: 20
          Listening queues: 'default'
          Shutdown timeout: 00:00:15
          Schedule polling interval: 00:00:15
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: C:\apps-services-net8\Chapter09\
      Northwind.Background.Hangfire
info: Hangfire.Server.BackgroundServerProcess[0]
      Server desktop-j1pqhr7:14120:c8ea792b successfully announced
      in 140.4628 ms
info: Hangfire.Server.BackgroundServerProcess[0]
      Server desktop-j1pqhr7:14120:c8ea792b is starting the registered
      dispatchers: ServerWatchdog, ServerJobCancellationWatcher,
      ExpirationManager, CountersAggregator, SqlServerHeartbeatProcess,
      Worker, DelayedJobScheduler, RecurringJobScheduler...
info: Hangfire.Server.BackgroundServerProcess[0]
      Server desktop-j1pqhr7:14120:c8ea792b all the dispatchers started
```

В браузере отобразится обычное текстовое сообщение.

9. В адресной строке браузера добавьте к URL-адресу значение `/hangfire`. Откроется пользовательский интерфейс инструмента *Hangfire Dashboard*, показанный на рис. 9.7.

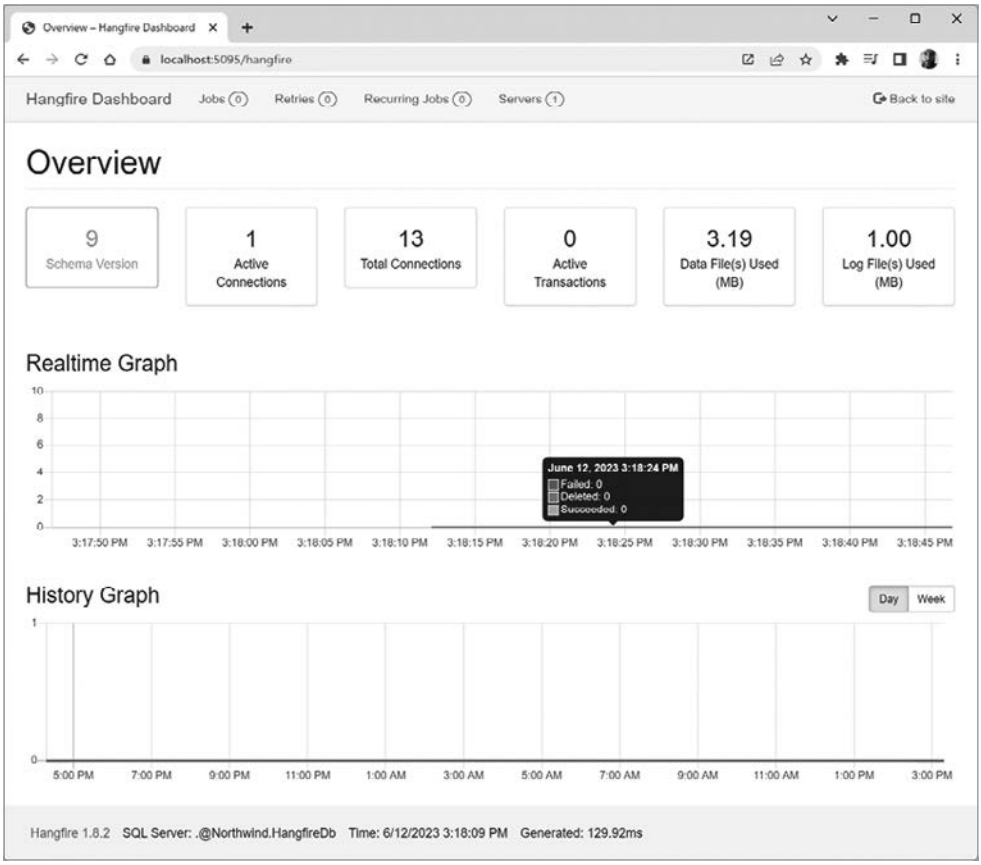


Рис. 9.7. Интерфейс Hangfire Dashboard

10. Закройте окно браузера и завершите работу сервера нажатием Ctrl+C. Прочитайте сообщения:

```
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: Hangfire.Server.BackgroundServerProcess[0]
      Server desktop-j1pqhr7:14120:c8ea792b caught stopping signal...
info: Hangfire.Server.BackgroundServerProcess[0]
      Server desktop-j1pqhr7:14120:c8ea792b All dispatchers stopped
info: Hangfire.Server.BackgroundServerProcess[0]
      Server desktop-j1pqhr7:14120:c8ea792b successfully reported itself
      as stopped in 2.8874 ms
info: Hangfire.Server.BackgroundServerProcess[0]
      Server desktop-j1pqhr7:14120:c8ea792b has been stopped
      in total 19.6204 ms
```

Планирование задач с помощью Hangfire

Далее мы запланируем на клиенте задание (например, вывод через некоторое время сообщения в консоль), отправив POST-запрос на веб-сервис.

1. В проекте Northwind.Background.Hangfire создайте файл класса WriteMessageJobDetail.cs.
2. В созданном файле определите класс, описывающий параметры запланированного задания:

```
namespace Northwind.Background.Models;

public class WriteMessageJobDetail
{
  public string? Message { get; set; }
  public int Seconds { get; set; }
}
```

3. В папке проекта Northwind.Background.Hangfire создайте файл класса Program.Methods.cs.
4. В этом файле дополните разделяемый класс Program методом для вывода сообщения в консоль зеленым текстом:

```
using static System.Console;

partial class Program
{
  public static void WriteMessage(string? message)
  {
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = ConsoleColor.Green;
    WriteLine(message);
    ForegroundColor = previousColor;
  }
}
```

5. В файле Program.cs импортируйте необходимые пространства имен для работы с задачей:

```
using Northwind.Background.Models; // Для класса WriteMessageJobDetail
using Microsoft.AspNetCore.Mvc;    // Для атрибута [FromBody]
```

6. В этом же файле Program.cs после GET-запроса сопоставьте POST-запрос с маршрутом /schedulejob, извлеките из запроса параметры задачи и используйте их для планирования (в секундах) фонового задания с помощью Hangfire:

```
app.MapPost("/schedulejob", ([FromBody] WriteMessageJobDetail job) =>
  {
    BackgroundJob.Schedule(
```

```
    methodCall: () => WriteMessage(job.Message),
    enqueueAt: DateTimeOffset.UtcNow +
      TimeSpan.FromSeconds(job.Seconds));
});
```

7. Запустите проект `Northwind.Background.Hangfire` без отладки.
8. В терминале убедитесь, что диспетчеры работают:

```
info: Hangfire.Server.BackgroundServerProcess[0]
      Server desktop-j1pqhr7:13916:9f1851b5 all the dispatchers started
```

9. В браузере перейдите по адресу `/hangfire`, чтобы открыть интерфейс Hangfire Dashboard.
10. В редакторе кода создайте файл `hangfire-schedule-job.http` в папке `HttpRequests`
11. Добавьте в него запрос для выполнения `POST`-запроса к Hangfire:

```
### Настройка переменной для базового адреса веб-сервиса Hangfire
@base_address = https://localhost:5095/

POST {{base_address}}schedulejob
Content-Type: application/json
{
  "message": "Hangfire is awesome!",
  "seconds": 30
}
```

12. Отправьте запрос и обратите внимание на успешный ответ:

```
HTTP/1.1 200 OK
Content-Length: 0
Connection: close
Date: Mon, 12 Jun 2023 16:04:20 GMT
Server: Kestrel
Alt-Svc: h3=":5095"; ma=86400
```

13. В Hangfire Dashboard в браузере перейдите в раздел `Scheduled` (Запланированные) и убедитесь, что запланированное задание отображается (рис. 9.8).
14. Подождите 30 секунд, затем в левой части интерфейса откройте раздел Succeeded (Успешно выполненные) с сообщением об успешном выполнении задачи (рис. 9.9).
15. В терминале отследите запись сообщения в консоль:

```
Hangfire is awesome!
```

16. Закройте браузер и выключите сервер.

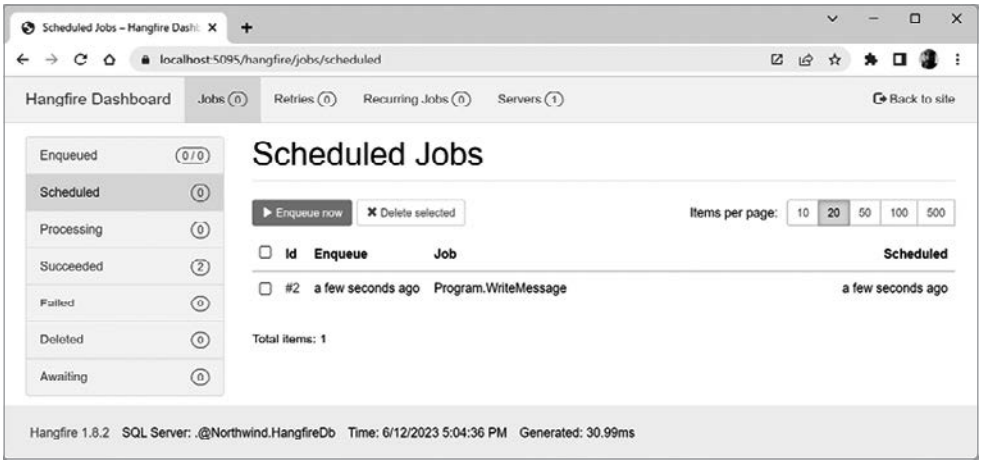


Рис. 9.8. Запланированные задачи в Hangfire Dashboard

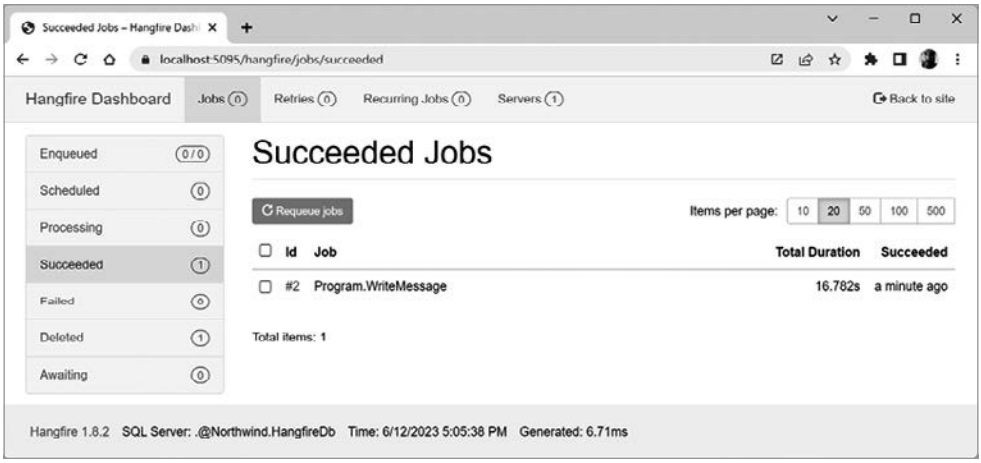


Рис. 9.9. Успешно выполненные задачи в Hangfire Dashboard

В копилку знаний: подробнее о Hangfire можно прочитать на официальном сайте по адресу https://www.hangfire.io/.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 9.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. Какова разница в скорости считывания 1 Мбайт данных с SSD и из оперативной памяти по результатам исследования Джеффа Дина?
2. В чем разница между абсолютными и скользящими сроками хранения?
3. В каких единицах измеряется параметр `Size` кэша?
4. В следующем операторе вы пробуете получить информацию о кэшировании в оперативной памяти, но параметр `stats` возвращает `null`. Как решить эту проблему?

```
MemoryCacheStatistics? stats = _memoryCache.GetCurrentStatistics();
```

5. Какие типы данных могут храниться: а) в кэше оперативной памяти и б) в распределенном кэше?
6. Чем различаются шаблоны Retry и Circuit Breaker?
7. В случае прямого обмена RabbitMQ какой ключ маршрутизации должен использоваться для очереди с именем `product`?
8. Чем отличается веерный обмен от обмена через топики?
9. На каком порте RabbitMQ прослушивает соединения по умолчанию?
10. При наследовании от класса `BackgroundService` какой метод необходимо переопределить, чтобы хост мог автоматически запустить ваш сервис?

Упражнение 9.2. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-9---caching-queuing-and-resilient-background-services.

Упражнение 9.3. Замените модель Distributed Memory Cache другой реализацией распределенного кэша

В этой главе для изучения основ работы с распределенным кэшем мы использовали реализацию Distributed Memory Cache.

Для выполнения дополнительного упражнения зарегистрируйте учетную запись Azure (если вы этого еще не сделали), создайте ресурс Azure Cache for Redis (Кэш Azure для Redis) и измените конфигурацию проекта веб-сервиса, чтобы использовать этот ресурс.

В файле проекта `Northwind.WebApi.Service` нужно сослаться на пакет Redis, закомментировать модель Distributed Memory Cache, а затем вызвать метод расширения для регистрации Redis в качестве реализации распределенного кэша:

```
// builder.Services.AddDistributedMemoryCache();

builder.Services.AddStackExchangeRedisCache(options =>
{
  options.Configuration = builder.Configuration
    .GetConnectionString("MyRedisConStr");

  options.InstanceName = "SampleInstance";
});
```

Дополнительную информацию можно найти по следующим ссылкам.

- Кэш Azure для Redis: https://azure.microsoft.com/ru-ru/products/cache/.
- Пакет Redis на сайте NuGet: https://www.nuget.org/packages/Microsoft.Extensions.Caching.StackExchangeRedis.
- Redis с .NET: https://docs.redis.com/latest/rs/references/client_references/client_csharp.

Упражнение 9.4. Замените Hangfire на Quartz.NET

Quartz.NET — библиотека, аналогичная Hangfire. Ознакомьтесь с официальной документацией (https://www.quartz-scheduler.net/), а затем создайте проект `Northwind.Background.Quartz`, который реализует ту же функциональность, что и `Northwind.Background.Hangfire`.

Упражнение 9.5. Знакомство с шаблоном Reliable Web App

Шаблон надежного веб-приложения (Reliable Web App, RWA) представляет собой набор лучших приемов для переноса локальных веб-проектов в облако. Шаблон включает эталонную реализацию и рекомендации по использованию облачных сервисов Azure для модернизации критически важных рабочих нагрузок. Вы научитесь современным методам проектирования, разработки и эксплуатации, обеспечивая надежность, безопасность, высокую производительность и экономическую эффективность своих проектов.

Документация: https://learn.microsoft.com/ru-ru/azure/architecture/web-apps/guides/enterprise-app-patterns/reliable-web-app/dotnet/guidance.

Подборка видеороликов о шаблоне Reliable Web App для .NET: https://www.youtube.com/playlist?list=PLI7iePan8aH54gIDJquV61dE3ENyaDi3Q.

Резюме

Из этой главы вы узнали:

- об архитектуре сервисов и влиянии различных компонентов системы на производительность;
- как кэшировать данные с использованием оперативной памяти и распределенным методом;
- как управлять HTTP-кэшированием для клиентов и посредников;
- как реализовать отказоустойчивость с помощью библиотеки Polly;
- как реализовать очереди с помощью брокера RabbitMQ;
- как создавать фоновые сервисы с длительным периодом работы с помощью класса `BackgroundService` и библиотеки Hangfire.

В следующей главе мы рассмотрим Azure Functions для реализации наносервисов (бессерверных сервисов).

10 Создание бессерверных наносервисов с помощью Azure Functions

В этой главе вы познакомитесь с Azure Functions. Это решение можно настроить для выполнения исключительно на стороне сервера. Его функции запускаются при определенных действиях, таких как поступление сообщения в очередь, загрузка файла в Azure Storage или срабатывание по расписанию.

Платформа Azure Functions

Azure Functions — событийная (управляемая событиями) платформа для бессерверных вычислений. Функции можно разрабатывать и отлаживать локально, а затем развертывать в облаке Microsoft Azure. Поддерживаются разные языки программирования, а не только C# и .NET. Доступны расширения для программ Visual Studio 2022 и Visual Studio Code, а также консольный инструмент.

Но что значит «бессерверный сервис»? Речь про отсутствие *постоянно работающего* сервера. Такой сервис бóльшую часть времени не задействован или тратит минимальные ресурсы, динамически масштабируясь при необходимости, позволяя снизить затраты.

Например, в организациях часто есть задачи, которые выполняются по расписанию — ежечасно, ежемесячно или при наступлении определенных условий. Допустим, компания формирует платежные ведомости в конце месяца. Функция, преобразующая суммы зарплат в текст, вполне может работать как бессерверный наносервис.

Другой пример — система управления контентом (CMS), где редакторы загружают изображения, требующие обработки: создания миниатюр, сжатия и других операций. Эти задачи могут выполняться функцией Azure при загрузке файла в хранилище BLOB-объектов Azure.

Платформа Azure Functions предоставляет не только отдельные функции. С помощью *устойчивых функций (durable functions)* можно создавать сложные решения, учитывающие состояние, рабочие процессы и события.

В этой книге устойчивые функции не рассматриваются. Подробнее об их реализации с использованием C# и .NET можно узнать по адресу https://learn.microsoft.com/ru-ru/azure/azure-functions/durable/durable-functions-overview.

На платформе Azure Functions используется модель программирования, основанная на триггерах и привязках. Благодаря этому наносервис может реагировать на события и взаимодействовать с другими сервисами, например с хранилищами данных.

Триггеры и привязки Azure Functions

Триггеры и *привязки* — ключевые концепции технологии Azure Functions.

Триггер — это событие, инициирующее запуск функции. У каждой функции может быть только один триггер. Рассмотрим основные типы триггеров.

- *HTTP* — срабатывает при входящем HTTP-запросе (обычно `GET` или `POST`).
- *Azure SQL* — реагирует на изменения в таблице SQL.
- *Azure Cosmos DB* — использует канал изменений для отслеживания вставок и обновлений.
- *Таймер* — запускается по расписанию. Повторных попыток выполнения при сбое нет — функция срабатывает только в следующий раз по графику.
- *SignalR* — реагирует на сообщения сервиса Azure SignalR.
- *Очередь* и *RabbitMQ* — срабатывают при поступлении сообщения в очередь.
- *Хранилище BLOB-объектов* — активируется при добавлении или изменении BLOB-файла.
- *Сетка событий* и *центр событий* — запускаются при наступлении заданного события.

Привязки определяют входные и выходные данные функции. Одна функция может содержать одну, несколько привязок или не иметь их вовсе. Основные типы привязок следующие.

- *Azure SQL* — чтение и запись в таблицу БД SQL Server.
- *Хранилище BLOB-объектов* — работа с файлами в BLOB-хранилище.
- *Azure Cosmos DB* — взаимодействие с документами в облачном хранилище.
- *SignalR* — прием и выполнение удаленных вызовов методов.
- *HTTP* — выполнение HTTP-запросов и обработка ответов.
- *Очередь* и *RabbitMQ* — запись и чтение сообщений в очереди.

- *SendGrid* — отправка электронных писем.
- *Twilio* — отправка SMS.
- *Центр IoT* — взаимодействие с подключенными устройствами.

Полный список поддерживаемых триггеров и привязок доступен по адресу https://learn.microsoft.com/ru-ru/azure/azure-functions/functions-triggers-bindings.

Настройка триггеров и привязок зависит от языка программирования. В C# и Java используются атрибуты для параметров методов, в остальных языках — конфигурационный файл `function.json`.

Выражения NCRONTAB

В триггере *«Таймер выражения NCRONTAB»* применяются для определения частоты срабатывания. Часовой пояс по умолчанию — *Всемирное координированное время (Coordinated Universal Time, UTC)*. Хотя его можно изменить, рекомендуется использовать именно UTC по причинам, которые мы обсуждали в главе 7.

Если система развернута с планом Azure App Service, в качестве альтернативы можно использовать объект `TimeSpan`, но выражения NCRONTAB обеспечивают большую гибкость.

Формат NCRONTAB включает пять или шесть компонентов (если заданы секунды):

```
* * * * * *
- - - - - -
| | | | | |
| | | | | +-------- день недели (0-6) (воскресенье=0)
| | | | +--------- месяц (1-12)
| | | +---------- день месяца (1-31)
| | +----------- час (0-23)
| +------------- минуты (0-59)
+-------------- секунды (0-59)
```

Символ `*` означает все допустимые значения для соответствующего столбца (указаны в скобках). Можно указывать диапазоны (`-`) и шаг (`/`). Примеры:

- `0` обозначает фиксированное значение. Например, для часов — ровно в полночь;
- `0,6,12,18` указывает на перечисленные значения. Например, срабатывание триггера в полночь, 6 утра, полдень и 6 вечера;
- `3-7` задает диапазон включительно. Например, запуск в 3, 4, 5, 6 и 7 часов;
- `4/3` означает, что отсчет начинается с 4, затем выполняется с шагом 3. Например, триггер будет срабатывать в 4, 7, 10, 13, 16, 19 и 22 часа.

Дополнительные примеры приведены в табл. 10.1.

Таблица 10.1. Примеры выражений NCRONTAB

Выражение	Описание
`0 5 * * * *`	Раз в час на 5-й минуте каждого часа
`0 0,10,30,40 * * * *`	Четырежды в час — на 0, 10, 30 и 40-й минутах каждого часа
`* * */2 * * *`	Каждые 2 часа
`0,15 * * * * *`	В 0 и 15 секунд каждой минуты
`0/15 * * * * *`	В 0, 15, 30 и 45 секунд каждой минуты, то есть с шагом 15 секунд
`0-15 * * * * *`	В первые 15 секунд каждой минуты (от 0 до 15 секунд)
`0 30 9-16 * * *`	Восемь раз в день — в 09:30, 10:30 и далее до 16:30
`0 */5 * * * *`	Каждые 5 минут начиная с 0-й минуты каждого часа
`0 0 */4 * * *`	Раз в 4 часа — в 0 минут каждого 4-го часа суток
`0 30 9 * * *`	Ежедневно в 09:30
`0 30 9 * * 1-5`	В будние дни в 09:30
`0 30 9 * * * Mon-Fri`	В рабочие дни (понедельник — пятница) в 09:30
`0 30 9 * Jan Mon`	Каждый понедельник января в 09:30

Теперь создадим консольное приложение для проверки знаний о выражениях NCRONTAB.

1. В редакторе кода создайте консольное приложение `NCrontab.Console` и поместите его в решение `Chapter10`.
2. В файле созданного проекта добавьте строгую обработку предупреждений, статически импортируйте класс `System.Console` на глобальном уровне и подключите пакет `NCrontab.Signed`:

```
<ItemGroup>
  <PackageReference Include="NCrontab.Signed" Version="3.3.3" />
</ItemGroup>
```

Библиотека NCRONTAB предназначена только для синтаксического анализа выражений и не является планировщиком. Подробнее можно узнать в GitHub-репозитории https://github.com/atifaziz/NCrontab.

3. Выполните сборку проекта `NCrontab.Console` для восстановления зависимостей.
4. Удалите содержимое файла `Program.cs`. Определите в нем диапазон дат на 2023 год, добавьте краткое описание синтаксиса NCRONTAB, постройте расписание NCRONTAB и выведите 40 ближайших вхождений за 2023 год:

```
using NCrontab; // Для класса CrontabSchedule и т. д.

DateTime start = new(year: 2023, month: 1, day: 1);
DateTime end = start.AddYears(1);

WriteLine($"Start at:   {start:ddd, dd MMM yyyy HH:mm:ss}");
WriteLine($"End at:     {end:ddd, dd MMM yyyy HH:mm:ss}");
WriteLine();

string sec = "0,30";
string min = "*";
string hour = "*";
string dayOfMonth = "*";
string month = "*";
string dayOfWeek = "*";

string expression = string.Format(
  "{0,-3} {1,-3} {2,-3} {3,-3} {4,-3} {5,-3}",
  sec, min, hour, dayOfMonth, month, dayOfWeek);

WriteLine($"Expression: {expression}");
WriteLine(@"            \ / \ / \ / \ / \ / \ /");
WriteLine($"             -   -   -   -   -   -");
WriteLine($"             |   |   |   |   |   |");
WriteLine($"             |   |   |   |   |   +--- day of week (0-6) (Sunday=0)");
WriteLine($"             |   |   |   |   +------- month (1-12)");
WriteLine($"             |   |   |   +----------- day of month (1-31)");
WriteLine($"             |   |   +--------------- hour (0-23)");
WriteLine($"             |   +------------------- min (0-59)");
WriteLine($"             +----------------------- sec (0-59)");
WriteLine();

CrontabSchedule schedule = CrontabSchedule.Parse(expression,
  new CrontabSchedule.ParseOptions { IncludingSeconds = true });

IEnumerable<DateTime> occurrences = schedule.GetNextOccurrences(start, end);

// Вывод первых 40 вхождений
foreach (DateTime occurrence in occurrences.Take(40))
{
  WriteLine($"{occurrence:ddd, dd MMM yyyy HH:mm:ss}");
}
```

Особенности реализации

- По умолчанию в качестве временного интервала используется весь 2023 год.
- Стандартное выражение `0,30 * * * *` означает, что события происходят каждые 30 секунд. Символы * ссылаются на каждый час каждого дня недели каждого месяца.
- Для форматирования вывода синтаксической справки используется значение `-3`, чтобы ширина каждого компонента была равна трем символам. Можно реализовать динамическое форматирование, чем вы займетесь в дополнительном упражнении.
- Поскольку выражение включает секунды, их нужно передавать в качестве дополнительного параметра.
- После определения расписание вызывает метод `GetNextOccurrences`, который возвращает последовательность вычисленных вхождений.
- Выводится только 40 первых вхождений — этого достаточно для понимания работы выражений.

5. Запустите консольное приложение без отладки и обратите внимание, что события происходят каждые 30 секунд (ниже показан фрагмент):

```
Start at:   Sun, 01 Jan 2023 00:00:00
End at:     Mon, 01 Jan 2024 00:00:00

Expression: 0,30 *   *   *   *   *
            \ / \ / \ / \ / \ / \ /
             -   -   -   -   -   -
             |   |   |   |   |   |
             |   |   |   |   |   +--- day of week (0-6) (Sunday=0)
             |   |   |   |   +------- month (1-12)
             |   |   |   +----------- day of month (1-31)
             |   |   +--------------- hour (0-23)
             |   +------------------- min (0-59)
             +----------------------- sec (0-59)

Sun, 01 Jan 2023 00:00:30
Sun, 01 Jan 2023 00:01:00
Sun, 01 Jan 2023 00:01:30
...
Sun, 01 Jan 2023 00:19:30
Sun, 01 Jan 2023 00:20:00
```

Хотя время начала — Sun, 01 Jan 2023 00:00:00, оно не включено в список, так как не является «следующим» вхождением.

6. Закройте консольное приложение.

7. В файле `Program.cs` измените выражение и протестируйте примеры из таблицы или придумайте собственные. Попробуйте применить выражение `0 0 */4 * * *` и обратите внимание, что оно выдаст следующий результат (показан фрагмент):

```
Start at:   Sun, 01 Jan 2023 00:00:00
End at:     Mon, 01 Jan 2024 00:00:00

Expression: 0   0   */4  *   *   *
            \ / \ / \ / \ / \ / \ /
             -   -   -   -   -   -
             |   |   |   |   |   |
             |   |   |   |   |   +--- day of week (0-6) (Sunday=0)
             |   |   |   |   +------- month (1-12)
             |   |   |   +----------- day of month (1-31)
             |   |   +--------------- hour (0-23)
             |   +------------------- min (0-59)
             +----------------------- sec (0-59)

Sun, 01 Jan 2023 04:00:00
Sun, 01 Jan 2023 08:00:00
Sun, 01 Jan 2023 12:00:00
Sun, 01 Jan 2023 16:00:00
Sun, 01 Jan 2023 20:00:00
Mon, 02 Jan 2023 00:00:00
Mon, 02 Jan 2023 04:00:00
Mon, 02 Jan 2023 08:00:00
Mon, 02 Jan 2023 12:00:00
Mon, 02 Jan 2023 16:00:00
Mon, 02 Jan 2023 20:00:00
Tue, 03 Jan 2023 00:00:00
...
Sat, 07 Jan 2023 12:00:00
Sat, 07 Jan 2023 16:00:00
```

Хотя время начала — Sun, 01 Jan 2023 00:00:00, оно не включено в список, так как не является «следующим» вхождением. Воскресенье имеет пять вхождений, а начиная с понедельника — шесть в день.

Версии и языки Azure Functions

Версия 4 среды выполнения Azure Functions — единственная актуальная. Все предыдущие версии устарели.

Дельный совет: Microsoft рекомендует использовать версию 4 для всех языков. Версии 1–3 переведены в режим поддержки и не рекомендуются.

Поддерживаемые языки и платформы в Azure Functions v4 (фрагмент):

- *C#, F#*: .NET 8, .NET 7, .NET 6 и .NET Framework 4.8. Обратите внимание, что .NET 6 и .NET 7 (а в будущем и .NET 9) поддерживаются только в изолированной модели рабочих процессов, поскольку это релизы *Standard-Term Support (STS)*. .NET 6 относится к LTS, но уже неактуален, так как был заменен новым LTS-релизом. .NET 8 поддерживается как в изолированной, так и в рабочей модели, поскольку это релиз *Long-Term Support (LTS)*.
- *JavaScript*: Node 14, 16 и 18. (TypeScript через транспиляцию).
- *Java* 8, 11 и 17.
- *PowerShell* 7.2.
- *Python* 3.7, 3.8, 3.9 и 3.10.

В копилку знаний: полный список поддерживаемых языков опубликован по адресу https://learn.microsoft.com/ru-ru/azure/azure-functions/functions-versions.

В книге рассматривается сервис Azure Functions с использованием C# и .NET 8, поскольку эта версия поддерживает изолированную и внутрипроцессную модели и имеет долгосрочную поддержку (LTS).

Подкованные пользователи могут зарегистрировать пользовательский обработчик и использовать любой язык для реализации функции Azure. Подробнее о пользовательских обработчиках Azure Functions можно прочитать по адресу https://learn.microsoft.com/ru-ru/azure/azure-functions/functions-custom-handlers.

Модели рабочих процессов Azure Functions

Azure Functions поддерживает две модели.

- *В процессе* (In-process). В этой модели функция реализуется в библиотеке классов, которая выполняется в том же процессе, что и хост. Поэтому должен использоваться актуальный LTS-релиз .NET. На момент выхода книги — .NET 8. Следующий LTS-релиз, .NET 10, ожидается осенью 2025 года, но, начиная с версии .NET 8, поддержка хостинга в процессе будет прекращена. Версии после .NET 8 будут поддерживать только модель изолированного рабочего процесса.
- *Изолированная* (Isolated). Функция реализуется в консольном приложении, запускаемом в отдельном процессе. Поэтому функция может работать на любой поддерживаемой версии .NET, а также предоставляет полный контроль над точкой входа `Main` и возможностью использовать дополнительные компоненты, такие как промежуточное ПО для вызовов. С релизом версии .NET 9 эта модель станет единственной поддерживаемой.

Подробнее об этом решении можно прочитать по адресу https://techcommunity.microsoft.com/t5/apps-on-azure-blog/net-on-azure-functions-august-2023-roadmap-update/ba-p/3910098.

Передовой опыт: для новых проектов рекомендуется изолированная модель.

Тарифные планы Azure Functions

После локального тестирования необходимо развернуть проект Azure Functions на хостинге Azure. Доступны следующие три тарифных плана.

- *Потребление*. Экземпляры хостов добавляются и удаляются динамически в зависимости от активности, поэтому этот план близок к бессерверному. Он автоматически масштабируется в периоды высокой нагрузки, а затраты начисляются только за время работы функций. Можно также настроить срок завершения функций, чтобы они не выполнялись бесконечно.
- *Премиум*. Этот план поддерживает эластичное масштабирование, прогретые экземпляры для предотвращения холодных стартов, неограниченную продолжительность выполнения, многоядерные экземпляры (до четырех ядер) и предусматривает более предсказуемые затраты. Стоимость зависит от времени работы ядра и выделенной памяти. Один экземпляр должен быть всегда активен, поэтому некая минимальная арендная плата в месяц будет взиматься обязательно, даже если функции не выполняются.
- *Выделенный*. Работает в облачном эквиваленте серверной фермы. Хостинг предоставляется Azure App Service, отвечающей за выделенные серверные ресурсы. Тарифные планы Azure App Service включают категории «Базовая», «Общая», «Премиальная» и «Изолированная». Этот вариант подойдет, если уже оформлен тарифный план App Service для других проектов, таких как сайты или gRPC-сервисы, OData, GraphQL. Стоимость зависит от выбранного плана App Service. Доступно неограниченное количество функций Azure и прочих веб-приложений.

Внимание! Будьте внимательны с категориями «Премиум» и «Выделенный» Azure App Service. Следует тщательно выбирать наиболее подходящий тарифный план, соответствующий вашему плану хостинга Azure Functions. Так, в категории «Премиум» следует выбрать тарифный план Elastic Premium («Гибкий премиум»), например EP1. При выборе выделенного плана App Service типа P1V1 ваша система не будет гибко масштабироваться!

Подробнее о планах Azure Functions вы можете прочитать по адресу https://learn.microsoft.com/ru-ru/azure/azure-functions/functions-scale.

Требования к хранилищу Azure Storage

Для работы с Azure Functions требуется учетная запись Azure Storage. Она используется для хранения данных, необходимых для функционирования привязок и триггеров. Эти сервисы могут также использоваться в коде.

- *Файлы Azure* — хранят и запускают код функциональных приложений в тарифных планах «Потребление» и «Премиум».
- *Хранилище BLOB-объектов Azure* — хранит состояние привязок и функциональных ключей.
- *Хранилище очередей Azure* — используется для обработки отказов и повторных попыток в некоторых триггерах.
- *Хранилище таблиц Azure* — центры задач в устойчивых функциях используют хранилища BLOB-объектов, очередей и таблиц Azure.

Локальное тестирование с помощью Azurite

Azurite — это инструмент с открытым исходным кодом для тестирования Azure Functions и работы с хранилищами BLOB, Queue и Table Storage на локальной машине. Он поддерживает платформы Windows, Linux и macOS и служит заменой устаревшему эмулятору Azure Storage Emulator.

Что нужно для установки Azurite?

- В программе Visual Studio 2022 инструмент Azurite включен в комплект поставки.
- В Visual Studio Code необходимо установить расширение `Azurite`.
- В JetBrains Rider нужно установить плагин Azure Toolkit for Rider, который содержит Azurite.
- Для установки через терминал потребуется Node.js версии 8 или выше. После этого выполните команду:

```
npm install -g azurite
```

После завершения локального тестирования функций Azure можно переключиться на использование учетной записи Azure Storage в облаке.

Дополнительную информацию об Azurite можно найти по адресу https://learn.microsoft.com/ru-ru/azure/storage/common/storage-use-azurite.

Уровни авторизации Azure Functions

В Azure Functions доступно три уровня авторизации, которые определяют необходимость использования ключа API:

- Anonymous (Анонимный): ключ API не требуется;
- Function (Функциональный): требуется ключ функционального уровня;
- Admin (Администратор): для доступа необходим мастер-ключ;

Ключи API можно получить на портале Azure.

Поддержка внедрения зависимостей в Azure Functions

В Azure Functions внедрение зависимостей основано на стандартных механизмах .NET, но с особенностями реализации в зависимости от выбранной модели рабочих процессов.

Для регистрации зависимостей создайте класс, наследующийся от `FunctionsStartup`, и переопределите его метод `Configure`. Примените атрибут `[FunctionsStartup]`, чтобы указать имя класса, отвечающего за запуск. Затем добавьте сервисы в экземпляр интерфейса `IFunctionsHostBuilder`, переданный методу. Пример реализации будет приведен в практическом задании далее в этой главе.

Класс, реализующий функцию Azure Functions, обычно статичен и содержит статический метод. Поскольку статические классы не могут быть созданы с помощью конструктора, для внедрения зависимостей следует использовать классы экземпляров, как для внедряемых сервисов, так и для самого класса функции. Это будет рассмотрено в практическом задании.

Установка Azure Functions Core Tools

Набор инструментов *Azure Functions Core Tools* предоставляет основную среду выполнения и шаблоны для создания функций, поддерживающих локальную разработку в операционных системах Windows, macOS и Linux в любых редакторах кода.

Инструментарий Azure Functions Core Tools входит в состав рабочей нагрузки Azure development для Visual Studio 2022, поэтому, возможно, этот инструмент уже установлен у вас.

Для установки последней версии *Azure Functions Core Tools* нужно перейти по адресу https://www.npmjs.com/package/azure-functions-core-tools.

На этой странице приведены инструкции по установке с помощью различных инструментов:

- *Microsoft Software Installer (MSI)* и `winget` для Windows;
- Homebrew для macOS;
- `npm` для любой операционной системы;
- а также для популярных дистрибутивов Linux.

В JetBrains Rider для установки Azure Functions Core Tools используйте встроенный функционал.

Создание проекта Azure Functions

Настало время создать проект Azure Functions. Функции можно добавить прямо в облаке на портале Azure, но удобнее сначала разработать и протестировать их локально. После успешного тестирования их можно развернуть в облаке.

У каждого редактора кода свои особенности работы с проектами Azure Functions, поэтому разберем их по очереди, начиная с Visual Studio 2022.

В Visual Studio 2022

Если вы пользуетесь Visual Studio 2022, для создания проекта Azure Functions нужно выполнить следующие шаги.

1. В Visual Studio 2022 создайте проект со следующими настройками:
   - выберите шаблон проекта Azure Functions (Функции Azure);
   - в поле Solution name (Имя решения) укажите значение `Chapter10`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Azure-Functions.Service`;
   - в раскрывающемся списке Functions worker (Рабочая роль функций) выберите пункт .NET 8.0 Isolated (Long Term Support) (.NET 8.0 Изолированный (долгосрочная поддержка));
   - в раскрывающемся списке Function (Функция) выберите пункт Http trigger (HTTP-триггер);
   - установите флажок Use Azurite for runtime storage account (Использование Azurite для учетной записи хранения среды выполнения);
   - снимите флажок Enable Docker (Включить поддержку контейнера);
   - в раскрывающемся списке Authorization level (Уровень авторизации): Anonymous (Анонимный).

2. Нажмите кнопку Create (Создать).
3. Затем присвойте параметру Startup Project (Запускаемый проект) значение Current selection (Текущий вариант).

В Visual Studio Code

Если вы пользуетесь программой Visual Studio Code, для создания проекта Azure Functions нужно выполнить следующие шаги.

1. В Visual Studio Code перейдите на вкладку Extensions (Расширения) и найдите расширение Azure Functions (`ms-azuretools.vscode-azurefunctions`). Оно зависит от двух расширений — Azure Account (`ms-vscode.azure-account`) и Azure Resources (`ms-azuretools.vscode-azureresourcegroups`), поэтому они тоже будут установлены. Нажмите кнопку Install (Установить), чтобы загрузить расширение.
2. Создайте папку `Northwind.AzureFunctions.Service`. (Если ранее вы работали с таким же проектом в программе Visual Studio 2022, создайте каталог `Chapter10-vscode`, а в ней — указанную папку.)
3. В программе Visual Studio Code откройте папку `Northwind.AzureFunctions.Service`.
4. Выберите команду меню View ▸ Command Palette (Вид ▸ Command Palette), введите `azure f`, а затем в списке команд Azure Functions выберите пункт Azure Functions: Create New Project (Azure Functions: создать новый проект).
5. Выберите папку `Northwind.AzureFunctions.Service`, в которой будет расположен создаваемый проект.
6. В ответ на подсказки выберите следующее:
   - Select a language (Выберите язык): C#;
   - Select a .NET runtime (Выберите среду выполнения .NET): .NET 8 Isolated LTS (.NET 8.0 Изолированный LTS);
   - Select a template for your project's first function (Выберите шаблон для первой функции вашего проекта): HTTP trigger;
   - Provide a function name (Укажите имя функции): `NumbersToWordsFunction`;
   - Provide a namespace (Предоставьте пространство имен): `Northwind.AzureFunctions.Service`;
   - Select the authorization level (Выберите уровень авторизации): Anonymous (Анонимный);
   - Select how you would like to open your project (Выберите способ открытия проекта): Open in current window (Открыть в текущем окне).

7. Выберите команду меню Terminal ▸ New Terminal (Терминал ▸ Новый терминал).
8. В терминале выполните сборку проекта:

```
dotnet build
```

В терминале с помощью инструмента func

Если вы пользуетесь другим редактором кода или терминалом, для создания проекта Azure Functions нужно выполнить следующие шаги.

1. Создайте каталог `Chapter10-cli` с вложенной папкой `Northwind.Azure-Functions.Service`.
2. В терминале откройте папку `Northwind.AzureFunctions.Service` и создайте новый проект Azure Functions на языке C#:

```
func init --csharp
```

3. В той же папке создайте функцию Azure Functions, вызываемую анонимно HTTP-триггером:

```
func new --name NumbersToWordsFunction --template "HTTP trigger" --authlevel
"anonymous"
```

4. По желанию функцию можно запустить локально:

```
func start
```

Если команда func недоступна в терминале, установите пакет Azure Functions Core Tools с помощью менеджера Chocolatey, воспользовавшись инструкцией на сайте https://community.chocolatey.org/packages/azure-functions-core-tools.

Обзор проекта Azure Functions

Прежде чем писать функцию, рассмотрим структуру проекта Azure Functions.

1. Откройте файл проекта и обратите внимание на версию Azure Functions, а также ссылки на пакеты, необходимые для обработки HTTP-запросов:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <AzureFunctionsVersion>v4</AzureFunctionsVersion>
    <OutputType>Exe</OutputType>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
```

```
  <ItemGroup>
    <PackageReference Version="1.19.0"
      Include="Microsoft.Azure.Functions.Worker" />
    <PackageReference Version="3.0.13"
      Include="Microsoft.Azure.Functions.Worker.Extensions.Http" />
    <PackageReference Version="1.14.0"
      Include="Microsoft.Azure.Functions.Worker.Sdk" />
  </ItemGroup>

  <ItemGroup>
    <None Update="host.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
    <None Update="local.settings.json">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
      <CopyToPublishDirectory>Never</CopyToPublishDirectory>
    </None>
  </ItemGroup>

  <ItemGroup>
    <Using Include="System.Threading.ExecutionContext"
      Alias="ExecutionContext" />
  </ItemGroup>
</Project>
```

2. В файле `host.json` включено журналирование в сервисе Application Insights, но исключены значения типа `Request`:

```
{
  "version": "2.0",
  "logging": {
    "applicationInsights": {
      "samplingSettings": {
        "isEnabled": true,
        "excludedTypes": "Request"
      },
      "enableLiveMetricsFilters": true
    }
  }
}
```

Application Insights — сервис мониторинга и журналирования Azure. В этой главе он не рассматривается.

3. В файле `local.settings.json` указано, что во время локальной разработки проекта используются локальное хранилище и изолированная рабочая модель:

```
{
  "IsEncrypted": false,
  "Values": {
```

```
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated"
  }
}
```

4. Если параметр `AzureWebJobsStorage` отсутствует или пуст, добавьте его в Visual Studio Code, присвойте параметру `UseDevelopmentStorage` значение `true` и сохраните изменения.

Параметр FUNCTIONS_WORKER_RUNTIME определяет используемый в проекте язык. Значение dotnet соответствует библиотеке классов .NET, а dotnet-isolated — консольному приложению .NET. Другие возможные варианты: java, node, powershell, python.

5. В папке `Properties` откройте файл `launchSettings.json`. Вы увидите произвольный номер порта, назначенный веб-сервису:

```
{
  "profiles": {
    "Northwind.AzureFunctions.Service": {
      "commandName": "Project",
      "commandLineArgs": "--port 7274",
      "launchBrowser": false
    }
  }
}
```

6. Измените номер порта на `5101` и сохраните изменения в файле.

Реализация функции

Создадим функцию, преобразующую числа в слова, воспользовавшись пакетом `Humanizer`.

1. В файле проекта подключите пакет `Humanizer`:

```
<PackageReference Include="Humanizer" Version="2.14.1" />
```

2. Выполните сборку проекта для восстановления зависимостей.

   В Visual Studio 2022 откройте проект `Northwind.AzureFunctions.Service` и переименуйте файл `Function1.cs` в `NumbersToWordsFunction.cs`.

3. Измените содержимое `NumbersToWordsFunction.cs`, реализовав преобразование чисел в слова:

```
using Humanizer; // Для метода расширения ToWords
using Microsoft.Azure.Functions.Worker; // Для атрибута [HttpTrigger]
using Microsoft.Azure.Functions.Worker.Http; // Для класса HttpResponseData
using Microsoft.Extensions.Logging; // Для интерфейса ILogger
```

```
namespace Northwind.AzureFunctions.Service;

public class NumbersToWordsFunction
{
  private readonly ILogger _logger;

  public NumbersToWordsFunction(ILoggerFactory loggerFactory)
  {
    _logger = loggerFactory.CreateLogger<NumbersToWordsFunction>();
  }

  [Function(nameof(NumbersToWordsFunction))]
  public HttpResponseData Run(
    [HttpTrigger(AuthorizationLevel.Anonymous,
      "get", "post", Route = null)] HttpRequestData req)
  {
    _logger.LogInformation("C# HTTP trigger function processed a request.");

    string? amount = req.Query["amount"];

    HttpResponseData response;

    if (long.TryParse(amount, out long number))
    {
      response = req.CreateResponse(System.Net.HttpStatusCode.OK);
      response.WriteString(number.ToWords());
    }
    else
    {
      response = req.CreateResponse(System.Net.HttpStatusCode.BadRequest);
      response.WriteString($"Failed to parse: {amount}");
    }

    return response;
  }
}
```

Если во время выполнения возникает показанное ниже исключение:

```
[2024-02-19T22:04:57.896Z] Function 'NumbersToWordsFunction', Invocation id
'1c7e2c25-32c6-4027-b30b-f29216597cfa': An exception was thrown by the invocation.
[2024-02-19T22:04:57.897Z] Result: Function 'NumbersToWordsFunction', Invocation id
'1c7e2c25-32c6-4027-b30b-f29216597cfa': An exception was thrown by the invocation.
Exception: System.InvalidOperationException: Synchronous operations are disallowed.
Call WriteAsync or set AllowSynchronousIO to true instead.
```

добавьте в файл `Program.cs` следующий код, чтобы устранить проблему:

```
services.Configure<KestrelServerOptions>(options =>
{
    options.AllowSynchronousIO = true;
});
```

Тестирование функции

Теперь проверим работу функции в локальной среде.

1. Запустите проект `Northwind.AzureFunctions.Service`:
   - в Visual Studio Code откройте панель Run and Debug (Запуск и отладка), выберите параметр Attach to .NET Functions (Прикрепление к функциям .NET), а затем нажмите кнопку Run (Запуск);
   - в операционной системе Windows, если появится запрос брандмауэра Windows Defender, нажмите кнопку Allow access (Разрешить доступ).
2. Вы увидите отчет о размещении функции, сгенерированный инструментом Azure Functions Core Tools (см. вывод ниже и рис. 10.1):

```
Azure Functions Core Tools
Core Tools Version:       4.0.5390 Commit hash: N/A (64-bit)
Function Runtime Version: 4.25.3.21264

[2023-10-01T11:42:05.319Z] Found C:\apps-services-net8\Chapter10\
Northwind.AzureFunctions.Service\Northwind.AzureFunctions.Service.csproj.
Using for user secrets file configuration.

Functions:
        NumbersToWordsFunction: [GET,POST] http://localhost:5101/api/
NumbersToWordsFunction
For detailed output, run func with --verbose flag.
[2023-06-05T11:42:14.265Z] Host lock lease acquired by instance
ID '000000000000000000000000011150C3D'.
```

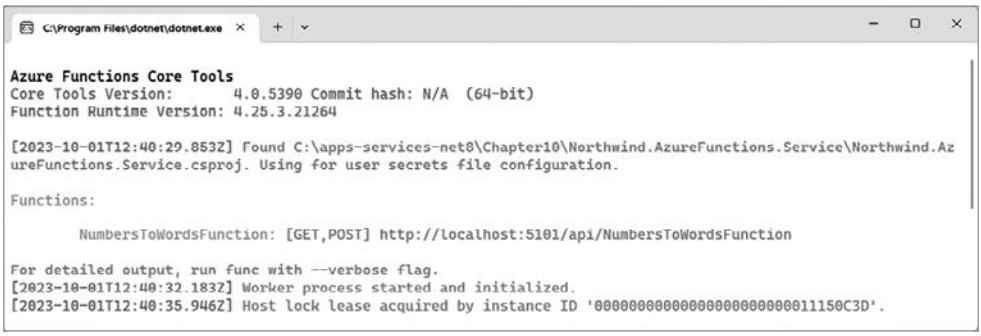


Рис. 10.1. Инструментарий Azure Functions Core Tools с отчетом о размещении функции

Если сообщение Host lock lease не появилось сразу, подождите несколько минут.

3. Скопируйте URL-адрес функции.
4. Запустите браузер Chrome.
5. Вставьте URL-адрес в адресную строку, добавьте запрос `?amount=123456` и убедитесь, что в браузере отобразился ответ `one hundred and twenty-three thousand four hundred and fifty-six` (рис. 10.2).

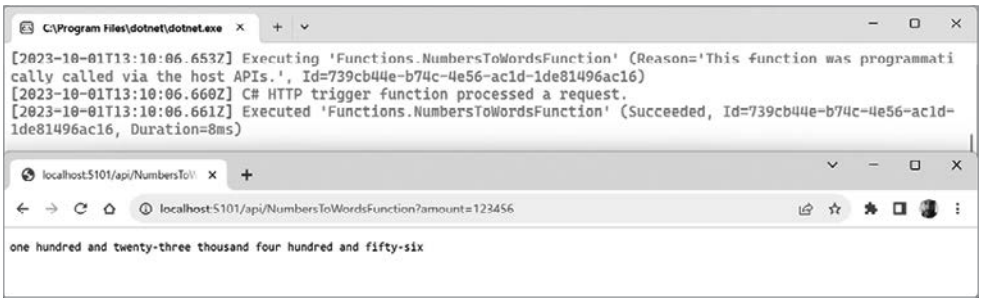


Рис. 10.2. Успешный вызов локальной функции Azure

6. В терминале появится подтверждение успешного вызова функции:

```
[2023-101-01T11:32:51.574Z] Executing 'NumbersToWordsFunction' (Reason='This
function was programmatically called via the host APIs.', Id=234d3122-ff3d-
4896-94b3-db3c8b5013d8)
[2023-10-01T11:32:51.603Z] C# HTTP trigger function processed a request.
[2023-10-01T11:32:51.629Z] Executed 'NumbersToWordsFunction' (Succeeded,
Id=234d3122-ff3d-4896-94b3-db3c8b5013d8, Duration=96ms)
```

7. Проверьте обработку некорректных данных:
   - вызовите функцию без параметра `amount` или передайте в запросе `?amount=apples`;
   - убедитесь, что возвращается код ответа `400` с сообщением `Failed to parse: apples`.
8. Закройте браузер Chrome и завершите работу веб-сервера (или отладку в Visual Studio Code).

Реакция на триггеры таймера и ресурсов

Теперь, рассмотрев функцию Azure Functions, отвечающую на HTTP-запрос, создадим несколько функций, реагирующих на другие типы триггеров.

В Azure Functions встроена поддержка HTTP-триггеров и таймера. Поддержка других привязок реализуется в виде пакетов расширения.

Реализация функции с триггером в форме таймера

Создадим функцию, которая будет запускаться каждый час и запрашивать страницу сайта Amazon.com с восьмым изданием книги *C# 12 and .NET 8 — Modern Cross-Platform Development Fundamentals*, чтобы отслеживать ее рейтинг среди бестселлеров в США.

Функция должна выполнять HTTP-запросы `GET`, поэтому необходимо внедрить фабрику HTTP-клиентов. Для этого добавим несколько ссылок на пакет и создадим класс запуска.

1. В файле проекта `Northwind.AzureFunctions.Service` подключите пакеты для работы с расширениями и таймерами Azure Functions:

```
<PackageReference Include="Microsoft.Azure.Functions.Extensions"
  Version="1.1.0" />
<PackageReference Version="4.2.0"
  Include="Microsoft.Azure.Functions.Worker.Extensions.Timer" />
```

2. Выполните сборку проекта `Northwind.AzureFunctions.Service` для восстановления зависимостей.
3. В файле `Program.cs` импортируйте пространство имен для работы с внедрением зависимостей и заголовками HTTP:

```
using Microsoft.Extensions.DependencyInjection; // Для метода AddHttpClient()
using System.Net.Http.Headers; // Для класса MediaTypeWithQualityHeaderValue
```

4. В том же файле добавьте код для настройки фабрики HTTP-клиентов, чтобы запросы к Amazon выполнялись так, будто их отправляет браузер Chrome:

```
var host = new HostBuilder()
  .ConfigureFunctionsWorkerDefaults()
  .ConfigureServices(services =>
  {
    services.AddHttpClient(name: "Amazon",
      configureClient: options =>
    {
      options.BaseAddress = new Uri("https://www.amazon.com");

      // Имитация браузера Chrome с американским английским
      options.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("text/html"));
      options.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/xhtml+xml"));
      options.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/xml", 0.9));
      options.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("image/avif"));
      options.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("image/webp"));
```

```
      options.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("image/apng"));
      options.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("*/*", 0.8));

      options.DefaultRequestHeaders.AcceptLanguage.Add(
        new StringWithQualityHeaderValue("en-US"));

      options.DefaultRequestHeaders.AcceptLanguage.Add(
        new StringWithQualityHeaderValue("en",0.8));
      options.DefaultRequestHeaders.UserAgent.Add(
        new(productName: "Chrome", productVersion: "114.0.5735.91"));
    });
  })
  .Build();

host.Run();
```

На 27 февраля 2025 года версия Chrome была 133.0.6943.142. Номер основной версии обычно увеличивается ежемесячно, поэтому в ноябре 2025 года он, скорее всего, стал 142, а в ноябре 2026 года — 154.

5. Создайте файл класса `ScrapeAmazonFunction.cs`.
6. Удалите содержимое файла функции и реализуйте код, который запрашивает страницу книги на сайте Amazon и обрабатывает ответ, сжатый в формате GZIP, извлекая рейтинг книги в списке бестселлеров:

```
using Microsoft.Azure.Functions.Worker; // Для атрибута [Function]
using Microsoft.Extensions.Logging; // Для интерфейса ILogger
using System.IO.Compression; // Для класса GZipStream и параметра
CompressionMode

namespace Northwind.AzureFunctions.Service;

public class ScrapeAmazonFunction
{
  private const string relativePath =
    "12-NET-Cross-Platform-Development-Fundamentals/dp/1837635870/";

  private readonly IHttpClientFactory _clientFactory;
  private readonly ILogger _logger;

  public ScrapeAmazonFunction(IHttpClientFactory clientFactory,
    ILoggerFactory loggerFactory)
  {
    _clientFactory = clientFactory;
    _logger = loggerFactory.CreateLogger<ScrapeAmazonFunction>();
  }
```

```
[Function(nameof(ScrapeAmazonFunction))]
public async Task Run( // Каждый час
  [TimerTrigger("0 0 * * * *")] TimerInfo timer)
{
  _logger.LogInformation($"C# Timer trigger function executed at {
    DateTime.UtcNow}.");

  _logger.LogInformation(
    $"C# Timer trigger function next three occurrences at: {
      timer.ScheduleStatus?.Next}.");

  HttpClient client = _clientFactory.CreateClient("Amazon");
  HttpResponseMessage response = await client.GetAsync(relativePath);

  _logger.LogInformation(
    $"Request: GET {client.BaseAddress}{relativePath}");

  if (response.IsSuccessStatusCode)
  {
    _logger.LogInformation("Successful HTTP request.");

    // Считывание содержимого GZIP-потока и помещение данных в строку
    Stream stream = await response.Content.ReadAsStreamAsync();
    GZipStream gzipStream = new(stream, CompressionMode.Decompress);
    StreamReader reader = new(gzipStream);
    string page = reader.ReadToEnd();

    // Извлечение рейтинга лучших продавцов
    int posBsr = page.IndexOf("Best Sellers Rank");
    string bsrSection = page.Substring(posBsr, 45);

    // Строка bsrSection будет иметь вид:
    //    "Best Sellers Rank: </span> #22,258 in Books ("

    // Получение позиции # с последующим пробелом
    int posHash = bsrSection.IndexOf("#") + 1;
    int posSpaceAfterHash = bsrSection.IndexOf(" ", posHash);

    // Получение номера из рейтинга бестселлеров (BSR) в виде текста
    string bsr = bsrSection.Substring(
      posHash, posSpaceAfterHash - posHash);

    bsr = bsr.Replace(",", null); // Удаление запятых

    // Синтаксический анализ текста с переводом в числа
    if (int.TryParse(bsr, out int bestSellersRank))
    {
      _logger.LogInformation(
        $"Best Sellers Rank #{bestSellersRank:N0}.");
    }
    else
    {
```

```
          _logger.LogError(
            $"Failed to extract BSR number from: {bsrSection}.");
        }
      }
      else
      {
        _logger.LogError("Bad HTTP request.");
      }
    }
  }
```

Проверка функции

Информацию о функции можно получить, отправив следующий HTTP-запрос `GET`:

`http://locahost:5101/admin/functions/<имя_функции>`

Теперь протестируем функцию таймера в локальной среде разработки.

1. Запустите проект `Northwind.AzureFunctions.Service`.
   - Убедитесь, что в программе Visual Studio Code установлено расширение Azurite и запущены службы Azurite.
   - Перейдите на панель Run and Debug (Запуск и отладка), выберите параметр Attach to .NET Functions (Прикрепление к функциям .NET), а затем нажмите кнопку Run (Запуск).
2. Убедитесь, что в терминале отображены две функции (ниже показан фрагмент):

```
Functions:

        NumbersToWordsFunction: [GET,POST] http://localhost:5101api/
NumbersToWordsFunction
        ScrapeAmazonFunction: timerTrigger
For detailed output, run func with --verbose flag.
```

3. В папке `HttpRequests` создайте файл `azurefunctions-scrapeamazon.http`.
4. Удалите его содержимое и добавьте код, определяющий глобальную переменную и два запроса к локально запущенному Azure Functions:

```
### Настройка переменной для базового адреса Azure Functions
@base_address = http://localhost:5101/

### Получение информации о функции NumbersToWordsFunction
GET {{base_address}}admin/functions/NumbersToWordsFunction

### Получение информации о функции ScrapeAmazonFunction
GET {{base_address}}admin/functions/ScrapeAmazonFunction
```

5. Отправьте первый запрос и проверьте, что в ответ получен JSON-документ с информацией о функции `NumbersToWordsFunction`:

```
HTTP/1.1 200 OK
Content-Length: 918
Connection: close
Content-Type: application/json; charset=utf-8
Date: Mon, 05 Jun 2023 13:32:11 GMT
Server: Kestrel

{
  "name": "NumbersToWordsFunction",
  "script_root_path_href": null,
  "script_href":
    "http://localhost:5101/admin/vfs/bin/
     Northwind.AzureFunctions.Service.dll",
  "config_href": null,
  "test_data_href": null,
  "href": "http://localhost:5101/admin/functions/NumbersToWordsFunction",
  "invoke_url_template": "http://localhost:5101/api/numberstowordsfunction",
  "language": "dotnet-isolated",
  "config": {
    "name": "NumbersToWordsFunction",
    "entryPoint": "Northwind.AzureFunctions.Service.NumbersToWordsFunction.Run",
    "scriptFile": "Northwind.AzureFunctions.Service.dll",
    "language": "dotnet-isolated",
    "functionDirectory": null,
    "bindings": [
      {
        "name": "req",
        "direction": "In",
        "type": "httpTrigger",
        "authLevel": "Anonymous",
        "methods": [
          "get",
          "post"
        ],
        "properties": {}
      },
      {
        "name": "$return",
        "type": "http",
        "direction": "Out"
      }
    ]
  },
  "files": null,
  "test_data": null,
  "isDisabled": false,
  "isDirect": true,
  "isProxy": false
}
```

6. Отправьте второй запрос и проверьте, что в ответ получен JSON-документ с данными о функции `ScrapeAmazonFunction`. В нем важны сведения о типе привязки и расписании (ниже показан фрагмент):

```
"bindings": [
  {
    "name": "timer",
    "direction": "In",
    "type": "timerTrigger",
    "schedule": "0 0 * * * *",
    "properties": {}
  }
],
```

7. Добавьте третий запрос, `POST`, который запустит функцию таймера вручную, без ожидания события расписания. В теле запроса должен содержаться пустой JSON-документ, а сам запрос должен будет отправиться на конечную точку `admin`:

```
### выполнение запроса к функции с таймером вручную
POST {{base_address}}admin/functions/ScrapeAmazonFunction
Content-Type: application/json

{}
```

8. Отправьте этот запрос и убедитесь, что он успешно принят:

```
HTTP/1.1 202 Accepted
```

9. Удалите символы `{}` в теле запроса, отправьте его снова. Обратите внимание, что возникла ошибка. Она означает, что тело запроса должно содержать пустой JSON-документ:

```
HTTP/1.1 400 Bad Request
```

10. Верните символы `{}` в тело запроса.
11. В терминале сервиса Azure Functions проверьте, что функция запустилась после вызова. Отобразятся время запуска (например, 13:49) и следующее запланированное срабатывание (14:00):

```
[2023-10-01T13:49:53.939Z] Executing 'Functions.ScrapeAmazonFunction'
(Reason='This function was programmatically called via the host APIs.',
Id=1df349a1-79c5-4b52-a7f1-d0f8f0d5cd9c)
[2023-10-01T13:49:54.095Z] C# Timer trigger function executed at 01/10/2023
13:49:54.
[2023-10-01T13:49:54.095Z] C# Timer trigger function next occurrence at:
01/10/2023 14:00:00.
[2023-10-01T13:49:54.105Z] Start processing HTTP request GET https://www.
amazon.com/12-NET-Cross-Platform-Development-Fundamentals/dp/1837635870/
```

```
[2023-10-01T13:49:54.106Z] Sending HTTP request GET https://www.amazon.com/12-
NET-Cross-Platform-Development-Fundamentals/dp/1837635870/
[2023-10-01T13:49:54.520Z] Received HTTP response after 407.4353ms - OK
[2023-10-01T13:49:54.521Z] End processing HTTP request after 420.1273ms -OK
[2023-10-01T13:49:56.212Z] Successful HTTP request.
[2023-10-01T13:49:56.212Z] Request: GET https://www.amazon.com/12-NETCross-
Platform-Development-Fundamentals/dp/1837635870/
[2023-10-01T13:49:56.251Z] Best Sellers Rank #384,269.
[2023-10-01T13:49:56.275Z] Executed 'Functions.ScrapeAmazonFunction'
(Succeeded, Id=1df349a1-79c5-4b52-a7f1-d0f8f0d5cd9c, Duration=2362ms)
```

12. Можно дождаться следующего события в расписании и убедиться, что триггер сработает автоматически:

```
[2023-10-01T14:00:00.023Z] Executing 'Functions.ScrapeAmazonFunction'
(Reason='Timer fired at 2023-10-01T15:00:00.0220351+01:00', Id=aa9f7495-6066-
4b0a-ba81-42582d677321)
[2023-10-01T14:00:00.027Z] C# Timer trigger function executed at 01/10/2023
14:00:00.
[2023-10-01T14:00:00.028Z] Start processing HTTP request GET https://www.
amazon.com/12-NET-Cross-Platform-Development-Fundamentals/dp/1837635870/
[2023-10-01T14:00:00.027Z] C# Timer trigger function next occurrence at:
01/10/2023 15:00:00.
[2023-10-01T14:00:00.028Z] Sending HTTP request GET https://www.amazon.com/12-
NET-Cross-Platform-Development-Fundamentals/dp/1837635870/
[2023-10-01T14:00:00.337Z] Received HTTP response after 305.1877ms - OK
[2023-10-01T14:00:00.339Z] End processing HTTP request after 305.5222ms -OK
[2023-10-01T14:00:01.899Z] Successful HTTP request.
[2023-10-01T14:00:01.899Z] Request: GET https://www.amazon.com/12-NETCross-
Platform-Development-Fundamentals/dp/1837635870/
[2023-10-01T14:00:01.931Z] Best Sellers Rank #387,339.
[2023-10-01T14:00:01.931Z] Executed 'Functions.ScrapeAmazonFunction'
(Succeeded, Id=aa9f7495-6066-4b0a-ba81-42582d677321, Duration=1909ms)
```

13. Если остановить сервис, подождать более часа, а затем снова запустить, функция выполнится немедленно, так как время ее срабатывания уже наступило:

```
[2023-10-01T16:19:31.369Z] Trigger Details: UnscheduledInvocationReason:
IsPastDue, OriginalSchedule: 2023-10-01T15:00:00.0000000+01:00
```

14. Отключите сервис Azure Functions.

Реализация функции для работы с очередями и BLOB-объектами

Функция с HTTP-триггером сразу отправляет ответ на простой `GET`-запрос, а теперь создадим функцию, которая будет добавлять сообщения в очередь Azure, указывая, что нужно сгенерировать изображение и загрузить его в хранилище BLOB-объектов. Затем его можно вывести для просмотра.

При локальном запуске сервиса сгенерированное изображение BLOB-объекта будет сохраняться в файловой системе, чтобы можно было проверить результат. Для этого добавим переменную окружения в локальные настройки.

Для создания изображения понадобится рукописный шрифт. Подойдет, к примеру, шрифт Caveat, доступный на сайте Google Fonts по адресу https://fonts.google.com/specimen/Caveat.

Приступим.

1. Скачайте шрифт по ссылке выше, распакуйте ZIP-файл и скопируйте файлы в папку `fonts`, показанную на примере Visual Studio 2022 на рис. 10.3.

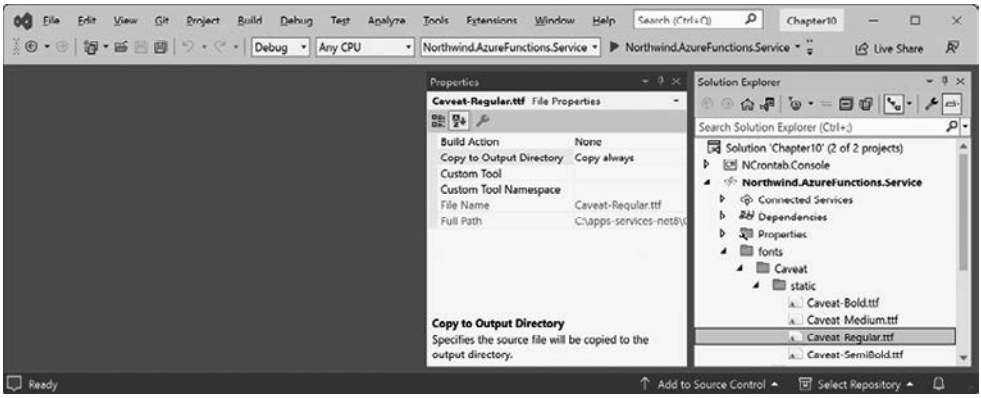


Рис. 10.3. Папка fonts с файлами шрифтов Caveat в Visual Studio 2022

2. Выберите файл шрифта `Caveat-Regular.ttf`.
3. В раскрывающемся списке Copy to Output Directory (Копировать в выходной каталог) на панели Properties (Свойства) выберите пункт Copy always (Всегда копировать) (см. рис. 10.3). Соответствующая запись будет добавлена в файл проекта:

```
<ItemGroup>
  <None Update="fonts\Caveat\static\Caveat-Regular.ttf">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  <None Update="host.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
  <None Update="local.settings.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    <CopyToPublishDirectory>Never</CopyToPublishDirectory>
  </None>
</ItemGroup>
```

В программе Visual Studio Code вручную добавьте выделенный код в файл проекта.

4. В файле проекта `Northwind.AzureFunctions.Service` добавьте ссылки на пакеты для работы с очередями и хранилищем BLOB-объектов Azure, а также библиотеку `ImageSharp` для рисования:

```
<PackageReference Version="5.2.0" Include=
  "Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues" />
<PackageReference Version="6.2.0" Include=
  "Microsoft.Azure.Functions.Worker.Extensions.Storage.Blobs" />
<PackageReference Include="SixLabors.ImageSharp" Version="3.0.2" />
<PackageReference Include="SixLabors.ImageSharp.Drawing" Version="2.0.0" />
```

5. Выполните сборку проекта для восстановления зависимостей.
6. В папке проекта `Northwind.AzureFunctions.Service` создайте файл класса `NumbersToChecksFunction.cs`.
7. В созданном файле добавьте код, регистрирующий функцию в выходной привязке для хранилища очередей Azure. Она будет записывать данные в именованную очередь и возвращать в нее сумму в виде слов:

```
using Humanizer; // Для метода расширения ToWords
using Microsoft.Azure.Functions.Worker; // Для атрибута [Function] и т. д.
using Microsoft.Azure.Functions.Worker.Http; // Для структуры HttpRequestData
using Microsoft.Extensions.Logging; // Для интерфейса ILogger

namespace Northwind.AzureFunctions.Service;

public class NumbersToChecksFunction
{
  private readonly ILogger _logger;

  public NumbersToChecksFunction(ILoggerFactory loggerFactory)
  {
    _logger = loggerFactory.CreateLogger<NumbersToChecksFunction>();
  }

  [Function(nameof(NumbersToChecksFunction))]
  [QueueOutput("checksQueue")] // Возвращаемое значение записывается
                               // в эту очередь
  public string Run(
    [HttpTrigger(AuthorizationLevel.Anonymous,
      "get", "post", Route = null)] HttpRequestData request)
  {
    _logger.LogInformation("C# HTTP trigger function processed a request.");

    string? amount = request.Query["amount"];

    if (long.TryParse(amount, out long number))
    {
      return number.ToWords();
    }
    else
    {
```

```
      return $"Failed to parse: {amount}";
    }
  }
}
```

8. В файле `local.settings.json` добавьте переменную окружения `IS_LOCAL` со значением `true`:

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated",
    "IS_LOCAL": true
  }
}
```

9. Создайте файл класса `CheckGeneratorFunction.cs`.
10. Удалите его содержимое и добавьте следующий код:

```
using Microsoft.Azure.Functions.Worker; // Для атрибута [Function] и т. д.
using Microsoft.Extensions.Logging; // Для интерфейса ILogger
using SixLabors.Fonts; // Для класса Font
using SixLabors.ImageSharp.Drawing; // Для интерфейса IPath
using SixLabors.ImageSharp.Drawing.Processing; // Для классов Brush, Pen
using SixLabors.ImageSharp; // Для Image
using SixLabors.ImageSharp.PixelFormats; // Для Rgba32
using SixLabors.ImageSharp.Processing; // Для Mutate

namespace Northwind.AzureFunctions.Service;

public class CheckGeneratorFunction
{
  private readonly ILogger _logger;

  public CheckGeneratorFunction(ILoggerFactory loggerFactory)
  {
    _logger = loggerFactory.CreateLogger<NumbersToWordsFunction>();
  }

  [Function(nameof(CheckGeneratorFunction))]
  [BlobOutput("checks-blob-container/check.png")]
  public byte[] Run(
    [QueueTrigger("checksQueue")] string message)
  {
    _logger.LogInformation("C# Queue trigger function executed.");
    _logger.LogInformation($"Body: {message}.");

    // Создание пустого изображения с белым фоном
    using (Image<Rgba32> image = new(width: 1200, height: 600,
      backgroundColor: new Rgba32(r: 255, g: 255, b: 255, a: 100)))
```

```
{
  // Загрузка файла шрифта и создание большого шрифта
  FontCollection collection = new();
  FontFamily family = collection.Add(
    @"fonts\Caveat\static\Caveat-Regular.ttf");

  Font font = family.CreateFont(72);

  string amount = message.Body.ToString();

  DrawingOptions options = new()
  {
    GraphicsOptions = new()
    {
      ColorBlendingMode = PixelColorBlendingMode.Multiply
    }
  };

  // Определение нескольких перьев и кистей

  Pen blackPen = Pens.Solid(Color.Black, 2);
  Pen blackThickPen = Pens.Solid(Color.Black, 8);
  Pen greenPen = Pens.Solid(Color.Green, 3);
  Brush redBrush = Brushes.Solid(Color.Red);
  Brush blueBrush = Brushes.Solid(Color.Blue);

  // Определение нескольких путей и их рисование

  IPath border = new RectangularPolygon(
    x: 50, y: 50, width: 1100, height: 500);

  image.Mutate(x => x.Draw(options, blackPen, border));

  IPath star = new Star(x: 150.0f, y: 150.0f,
    prongs: 5, innerRadii: 20.0f, outerRadii: 30.0f);

  image.Mutate(x => x.Fill(options, redBrush, star)
                     .Draw(options, greenPen, star));

  IPath line1 = new Polygon(new LinearLineSegment(
    new PointF(x: 100, y: 275), new PointF(x: 1050, y: 275)));

  image.Mutate(x => x.Draw(options, blackPen, line1));

  IPath line2 = new Polygon(new LinearLineSegment(
    new PointF(x: 100, y: 365), new PointF(x: 1050, y: 365)));

  image.Mutate(x => x.Draw(options, blackPen, line2));

  RichTextOptions textOptions = new(font)
  {
    Origin = new PointF(100, 200),
    WrappingLength = 1000,
```

```
        HorizontalAlignment = HorizontalAlignment.Left
      };

      image.Mutate(x => x.DrawText(
        textOptions, amount, blueBrush, blackPen));

      string blobName = $"{DateTime.UtcNow:yyyy-MM-dd-hh-mm-ss}.png";
      _logger.LogInformation($"Blob filename: {blobName}.");

      try
      {
        if (Environment.GetEnvironmentVariable("IS_LOCAL") == "true")
        {
          // Создание BLOB-объекта в локальной файловой системе

          string folder = $@"{System.Environment.CurrentDirectory}\ blobs";

          if (!Directory.Exists(folder))
          {
            Directory.CreateDirectory(folder);
          }
          _logger.LogInformation($"Blobs folder: {folder}");

          string blobPath = $@"{folder}\{blobName}";

          image.SaveAsPng(blobPath);
        }

        // Создание BLOB-объекта в хранилище с помощью потока памяти

        MemoryStream stream = new();

        image.SaveAsPng(stream);

        stream.Seek(0, SeekOrigin.Begin);

        return stream.ToArray();
      }
      catch (System.Exception ex)
      {
        _logger.LogError(ex.Message);
      }
      return Array.Empty<byte>();
    }
  }
}
```

Особенности реализации

- Параметр `[QueueTrigger("checksQueue")] string message` означает, что функция срабатывает при добавлении сообщения в очередь `checksQueue` и элемент очереди автоматически передается в параметр `message`.

- Библиотека ImageSharp используется для создания изображения размером 1200 × 600 пикселей.
- Имя BLOB-объекта формируется по текущей дате и времени в UTC, чтобы избежать дублирования (в реальной системе лучше использовать уникальный идентификатор GUID или нечто подобное).
- Если переменной окружения `IS_LOCAL` присвоено значение `true`, PNG-изображение сохраняется в подкаталоге `blobs` в локальной файловой системе.
- PNG-изображение сохраняется в поток памяти, а затем возвращается и в виде массива байтов загружается в контейнер BLOB-объектов с атрибутом `[BlobOutput("checks-blob-container/check.png")]`.

Тестирование функции

Теперь мы можем протестировать функцию, работающую с очередями и BLOB-объектами, в локальной среде разработки.

1. Запустите проект `Northwind.AzureFunctions.Service`: перейдите на панель Run and Debug (Запуск и отладка), выберите параметр Attach to .NET Functions (Прикрепление к функциям .NET), а затем нажмите кнопку Run (Запуск).
2. Убедитесь, что теперь работают четыре функции (ниже показан фрагмент):

```
Functions:

        NumbersToChecksFunction: [GET,POST]
          http://localhost:5101/api/NumbersToChecksFunction

        NumbersToWordsFunction: [GET,POST]
          http://localhost:5101/api/NumbersToWordsFunction

        CheckGeneratorFunction: queueTrigger

        ScrapeAmazonFunction: timerTrigger
```

3. В папке `HttpRequests` создайте файл `azurefunctions-numberstochecks.http`.
4. Удалите его содержимое и добавьте следующий код:

```
### Настройка переменной для базового адреса Azure Functions
@base_address = http://localhost:5101/

### Запуск функции NumbersToChecksFunction
GET {{base_address}}api/NumbersToChecksFunction?amount=123456
```

5. Отправьте запрос. В ответ поступит JSON-документ с информацией о функции `NumbersToWordsFunction`:

```
Response time: 2524 ms
Status code: OK (200)
Transfer-Encoding: chunked
Date: Mon, 05 Jun 2023 13:53:11 GMT
Server: Kestrel

Content-Type: text/plain; charset=utf-8
Content-Length: 64

-----------------------------------------------
Content:
one hundred and twenty-three thousand four hundred and fifty-six
```

6. В терминале проверьте, успешно ли прошел вызов функции — в очередь отправлено сообщение, вызвавшее функцию `CheckGeneratorFunction`:

```
[2023-06-05T13:53:12.175Z] Executing 'NumbersToWordsFunction' (Reason='This
function was programmatically called via the host APIs.', Id=b6a49d34-edbf-4c2a-
97f2-195f8d06cd13)
[2023-06-05T13:53:12.195Z] C# HTTP trigger function processed a request.
[2023-06-05T13:53:12.262Z] Executed 'NumbersToWordsFunction' (Succeeded,
Id=b6a49d34-edbf-4c2a-97f2-195f8d06cd13, Duration=104ms)
[2023-06-05T13:53:14.302Z] Executing 'CheckGeneratorFunction' (Reason='New queue
message detected on 'checksqueue'.', Id=2697ddc0-46dd-4c06-b960-fb5a443ec929)
[2023-06-05T13:53:14.305Z] Trigger Details: MessageId: 229e4961-bfaf-46da-bb17-
040ffc2bbf91, DequeueCount: 1, InsertedOn: 2023-06-05T13:53:12.000+00:00
[2023-06-05T13:53:14.313Z] C# Queue trigger function executed.
[2023-06-05T13:53:14.314Z] MessageId: 229e4961-bfaf-46da-bb17-040ffc2bbf91.
[2023-06-05T13:53:14.316Z] InsertedOn: 05/06/2023 13:53:12 +00:00.
[2023-06-05T13:53:14.317Z] ExpiresOn: 12/06/2023 13:53:12 +00:00.
[2023-06-05T13:53:14.318Z] Body: one hundred and twenty-three thousand four
hundred and fifty-six.
[2023-06-05T13:53:14.845Z] Blob name: 2023-06-05-01-53-14.png.
[2023-06-05T13:53:14.848Z] Blobs folder: C:\apps-services-net8\Chapter10\
Northwind.AzureFunctions.Service\bin\Debug\net8.0\blobs
[2023-06-05T13:53:15.057Z] Blob sequence number: 0.
[2023-06-05T13:53:15.060Z] Executed 'CheckGeneratorFunction' (Succeeded,
Id=2697ddc0-46dd-4c06-b960-fb5a443ec929, Duration=776ms)
[2023-06-05T13:53:20.979Z] Host lock lease acquired by instance ID
'000000000000000000000000011150C3D'.
```

7. Убедитесь, что в папке `Northwind.AzureFunctions.Service\bin\Debug\net8.0\blobs` создано локальное изображение, показанное на рис. 10.4.

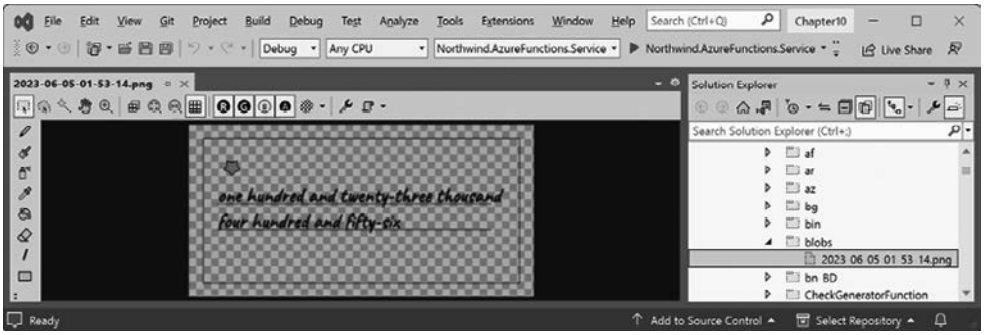


Рис. 10.4. Изображение, созданное в папке blobs проекта

8. Созданное изображение, открытое в программе Paint, показано на рис. 10.5.

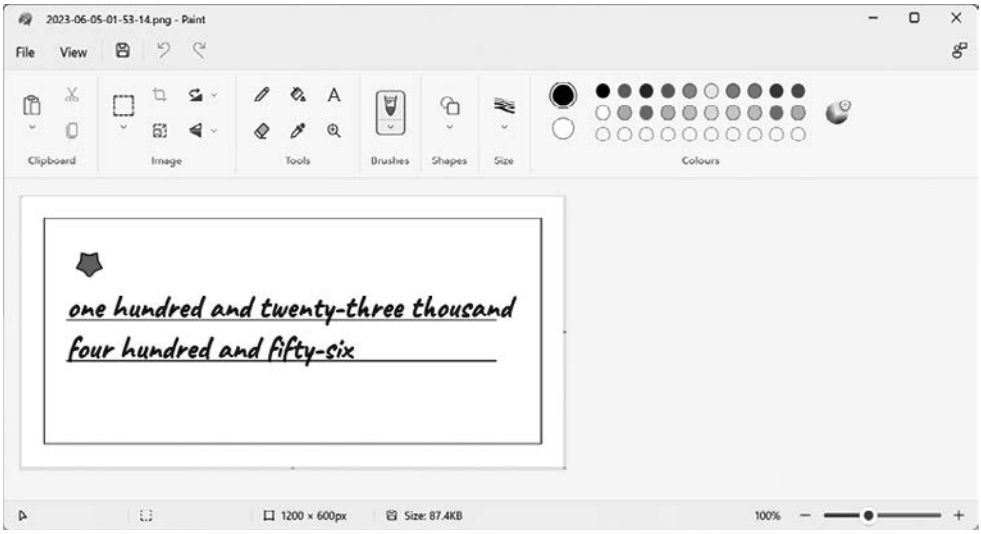


Рис. 10.5. Изображение, открытое в программе Windows Paint

Публикация проекта Azure Functions в облаке

Теперь создадим функциональное приложение, связанные с ним ресурсы в подписке Azure, развернем функции в облаке и запустим их.

Если у вас еще нет учетной записи Azure, ее можно бесплатно зарегистрировать по адресу https://azure.microsoft.com/ru-ru/free/.

Публикация в Visual Studio 2022

В Visual Studio 2022 реализован специальный графический интерфейс для публикации проектов в Azure.

1. На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши на проекте `Northwind.AzureFunctions.Service` и выберите в контекстном меню пункт Publish (Опубликовать).
2. В открывшемся диалоговом окне выберите вариант Azure и нажмите кнопку Next (Далее).
3. Выберите вариант Azure Function App (Linux) и нажмите кнопку Next (Далее).
4. Войдите в систему и введите свои учетные данные Azure.
5. Выберите подписку, например Pay-As-You-Go.
6. В разделе Function Instance (Экземпляр функции) нажмите кнопку + Create New (+ Создать новую).
7. Появится диалоговое окно, показанное на рис. 10.6. Заполните его следующим образом.
   - В поле Name (Имя) укажите глобально уникальное значение. По умолчанию включает название проекта и текущие дату и время.
   - В раскрывающемся списке Subscription name (Подписка) выберите действующую подписку.
   - В раскрывающемся списке Resource group (Группа ресурсов) выберите существующую группу или создайте новую (впоследствии будет удобно удалить группу с ресурсами одним щелчком), например `apps-services-book`.
   - В раскрывающемся списке Plan Type (Тип тарифного плана) выберите пункт Consumption (Потребление) — оплата только используемых ресурсов.
   - В раскрывающемся списке Location (Расположение) выберите ближайший центр обработки данных, например UK South (Юг Великобритании).
   - В строке параметра Azure Storage (Хранилище Azure) щелкните кнопкой мыши на ссылке New (Новая) и создайте учетную запись `northwindazure-functions` (или с другим уникальным именем) в ближайшем центре обработки данных. Выберите тип записи Standard — Locally Redundant Storage (Стандарт — локальное резервируемое хранилище).
   - В раскрывающемся списке Application Insights: выберите пункт None (Нет).
8. Нажмите кнопку Create (Создать). Процесс создания приложения может занять пару минут.
9. В диалоговом окне Publish (Опубликовать) нажмите кнопку Finish (Готово), а затем кнопку Close (Закрыть).

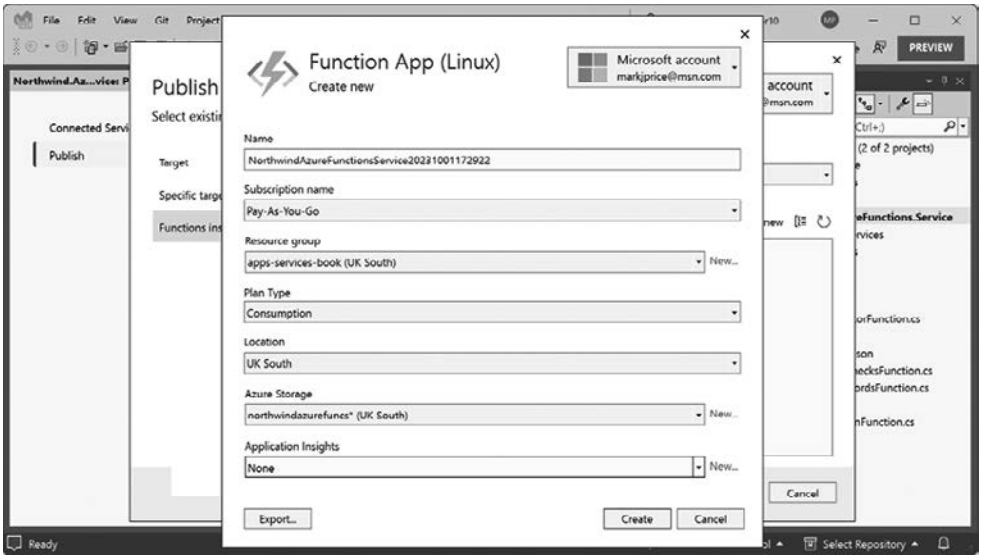


Рис. 10.6. Создание функционального приложения Azure

10. На панели Publish (Опубликовать), показанной на рис. 10.7, нажмите кнопку Publish (Опубликовать).

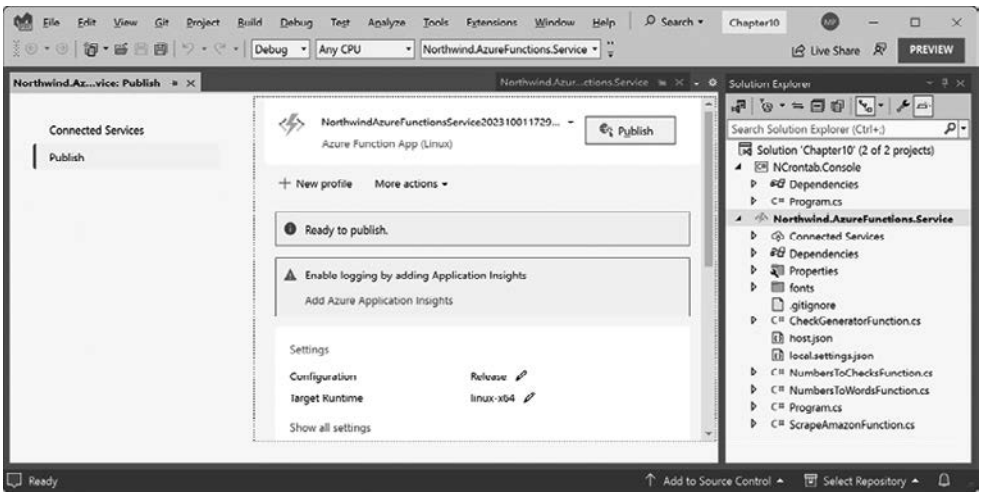


Рис. 10.7. Приложение Azure Function App готово к публикации в Visual Studio 2022

11. Проанализируйте вывод в консоли:

```
Build started...
1>------ Build started: Project: Northwind.AzureFunctions.Service,
Configuration: Release Any CPU ------
1>Northwind.AzureFunctions.Service -> C:\apps-services-net8\Chapter10\Northwind.
AzureFunctions.Service\bin\Release\net8.0\Northwind.AzureFunctions.Service.dll
2>------ Publish started: Project: Northwind.AzureFunctions.Service,
Configuration: Release Any CPU ------
2>Northwind.AzureFunctions.Service -> C:\apps-services-net8\Chapter10\Northwind.
AzureFunctions.Service\bin\Release\net8.0\Northwind.
AzureFunctions.Service.dll
2>Northwind.AzureFunctions.Service -> C:\apps-services-net8\Chapter10\
Northwind.AzureFunctions.Service\obj\Release\net8.0\PubTmp\Out\
2>Publishing C:\apps-services-net8\Chapter10\Northwind.
AzureFunctions.Service\obj\Release\net8.0\PubTmp\Northwind.
AzureFunctions.Service - 20230605152148071.zip to https://
northwindazurefunctionsservice20230605151137.scm.azurewebsites.net/api/
zipdeploy...
2>Zip Deployment succeeded.
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
==========
========== Build started at 3:21 PM and took 03:34.100 minutes ==========
========== Publish: 1 succeeded, 0 failed, 0 skipped ==========
========== Publish started at 3:21 PM and took 03:34.100 minutes
==========
Waiting for Function app to be ready...
Function app is ready
```

12. После успешной публикации на панели Publish (Опубликовать) нажмите кнопку Open site (Открыть сайт) и убедитесь, что ваш хост-сайт Azure Functions v4 готов.
13. Протестируйте функцию в браузере, добавив в адресную строку относительный URL-адрес `/api/NumbersToWordsFunction?amount=987654321` (рис. 10.8).

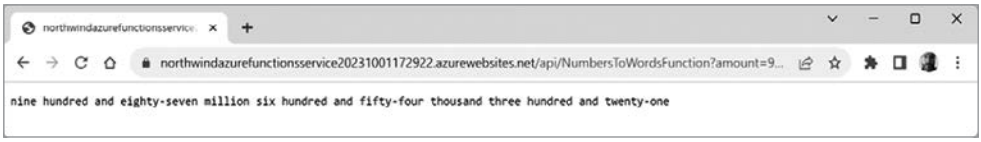


Рис. 10.8. Вызов функции, размещенной в облаке Azure

Публикация в Visual Studio Code

Подробнее о развертывании проектов с помощью Visual Studio Code можно узнать по адресу https://learn.microsoft.com/ru-ru/azure/azure-functions/functions-develop-vs-code#sign-in-to-azure.

После успешной публикации проекта Azure Functions в облаке важно правильно распределить ресурсы. Давайте рассмотрим, как удалять ненужные ресурсы Azure Functions во избежание напрасных затрат и для эффективного управления ресурсами.

Удаление ресурсов Azure Functions

Чтобы избежать ненужных затрат, важно правильно управлять ресурсами. Если функциональное приложение больше не требуется, его можно удалить вместе с зависимыми ресурсами.

1. Откройте браузер и перейдите на портал Azure по адресу https://portal.azure.com/.
2. На сайте Azure в разделе Overview (Обзор) вашего функционального приложения выберите пункт Resource Group (Группа ресурсов).
3. Выберите ресурсы, подлежащие удалению; например, Storage account (Учетная запись хранилища), Function App (Функциональное приложение) и App Service plan (План службы приложений).
4. Если требуется удалить всю группу, нажмите кнопку Delete resource group (Удалить группу ресурсов) и подтвердите действие. Можно также удалить ресурсы по отдельности.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 10.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. В чем разница между моделями `in-process` и `isolated` для Azure Functions?
2. Какой атрибут используется для срабатывания функции при поступлении сообщения в очередь?
3. Какой атрибут используется для поддержки очередью отправки сообщений?
4. Что означает показанное ниже NCRONTAB-выражение?

   ```
   0 0 */6 * 6 6
   ```

5. Как зарегистрировать зависимости для использования в функции?

Упражнение 10.2. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-10---building-serverless-nanoservices-using-azure-functions.

Резюме

Из этой главы вы узнали:

- основные концепции Azure Functions;
- как создавать бессерверные сервисы в Azure Functions;
- как реагировать на HTTP-триггеры, таймеры и очереди;
- как работать с привязками к хранилищам очередей и BLOB-объектов Azure;
- как развернуть проект Azure Functions в облаке.

В следующей главе вы познакомитесь с *SignalR* — технологией, позволяющей организовать связь между клиентом и сервером в реальном времени.

11 Трансляция сообщений в реальном времени с помощью SignalR

В этой главе рассматривается технология SignalR, позволяющая создавать сервисы для трансляции сообщений нескольким клиентам в реальном времени. Классический пример — групповая чат-комната, также возможны сценарии с уведомлениями и дашбордами, которым требуется мгновенное обновление данных, например котировок акций.

Знакомство с SignalR

Чтобы понять, какие задачи решает SignalR, стоит рассмотреть разработку веб-приложений без него. Всемирная паутина основана на протоколе HTTP, который более 30 лет успешно используется для сайтов и веб-сервисов. Однако он не предназначен для сценариев, требующих моментального обновления информации.

Развитие связи в реальном времени

Для оценки преимуществ технологии SignalR необходимо погрузиться в историю протокола HTTP и вклад компаний в его адаптацию для улучшения связи между клиентами и серверами в реальном времени.

На заре развития Всемирной паутины в 1990-х годах браузеры загружали обновленные данные с помощью полного HTTP-запроса `GET` к серверу. В 1999 году в Internet Explorer 5 появился компонент *XMLHttpRequest*, выполняющий асинхронные HTTP-вызовы в фоновом режиме. Это, наряду с появлением технологии *Dynamic HTML (DHTML)*, позволило обновлять отдельные части веб-страниц без полной перезагрузки.

Преимущества этой техники были очевидны, и вскоре все браузеры обладали аналогичными компонентами.

AJAX

Позже компания Google активно применяла эту технологию в смарт-сервисах Gmail и Google Maps. Спустя несколько лет это привело к популяризации подхода *асинхронного JavaScript и XML (Asynchronous JavaScript and XML, AJAX)*. Однако в AJAX для обмена данными по-прежнему используется протокол HTTP, вследствие чего возникают следующие ограничения.

- Во-первых, HTTP работает по модели «запрос — ответ», то есть сервер не может отправлять данные клиенту без запроса.
- Во-вторых, в HTTP-запросах и ответах, помимо самих сообщений, передаются заголовки, увеличивающие объем передаваемых данных.

WebSocket

Протокол WebSocket решает эти проблемы, обеспечивая двустороннюю (полнодуплексную) связь между клиентом и сервером через постоянное соединение. Это позволяет мгновенно передавать данные без необходимости отправки отдельных HTTP-запросов. Помимо прочего, минимальный размер сообщений ограничен 2 байтами, что положительно сказывается на объеме трафика. Протокол WebSocket поддерживает работу через стандартные порты HTTP (80 и 443), поэтому совместим с протоколом HTTP. При установлении прямого соединения используется HTTP-заголовок *Upgrade* для перехода от протокола HTTP к WebSocket.

Современные веб-приложения направлены на предоставление актуальной информации в реальном времени. Классический пример — групповая чат-комната, а также приложения с котировками акций и игры. WebSocket — отличная технология динамического обновления веб-страниц сервером в реальном времени. Несмотря на ее эффективность, есть ограничения в плане поддержки клиентами. Проверить совместимость можно на сайте https://caniuse.com/websockets.

Как правильно, WebSocket или WebSockets? «В 2011 году сетевой протокол WebSocket задокументирован Инженерным советом Интернета в спецификации RFC 6455. Для использования этого протокола веб-приложениями разработан API-стандарт WebSockets» (цитата из Википедии, https://en.wikipedia.org/wiki/WebSocket).

SignalR

ASP.NET Core SignalR — библиотека с открытым исходным кодом, упрощающая добавление функций реального времени в C#-приложения. Она абстрагирует различные технологии связи, автоматически выбирая наиболее подходящую.

Если браузер поддерживает WebSocket, SignalR использует его. В противном случае применяется альтернативный механизм, например AJAX-долгий опрос (long polling). Это позволяет избежать необходимости адаптировать код под разные технологии. Поэтому разработчику не обязательно понимать и реализовывать вручную каждую технологию.

SignalR — это API для *удаленного вызова процедур (Remote Procedure Call, RPC)*, позволяющее вызывать JavaScript-функции на клиенте из .NET-кода на сервере. Библиотека поддерживает конвейеризацию, диспетчеризацию сообщений и имеет два встроенных протокола — JSON и бинарный (MessagePack).

На стороне сервера SignalR может работать в системах Windows, macOS и Linux. Со стороны клиентов поддерживаются следующие платформы.

- JavaScript-клиенты для современных браузеров, в том числе Chrome, Firefox, Safari и Edge.
- .NET-клиенты, в том числе Blazor, .NET MAUI и Xamarin для мобильных приложений на Android и iOS.
- Java (версии 8 и новее).

Сервис Azure SignalR

Ранее я упоминал, что разделение хостинга SignalR и веб-проекта с JavaScript-библиотекой в качестве клиента повышает производительность. Это связано с тем, что сервису SignalR потенциально приходится обрабатывать множество одновременных подключений и оперативно на них реагировать.

С хостингом SignalR вам становятся доступны преимущества облачного *сервиса Azure SignalR*:

- глобальное покрытие и надежная сеть передачи данных;
- масштабируемость до миллионов подключений;
- соответствие соглашениям SLA;
- высокий уровень безопасности.

Дополнительные сведения о сервисе Azure SignalR вы найдете по адресу https://learn.microsoft.com/ru-ru/azure/azure-signalr/signalr-overview.

Разработка сигнатур методов

При проектировании методов в SignalR рекомендуется передавать сообщения в виде одного объекта вместо нескольких отдельных параметров. SignalR не накладывает жестких ограничений на сигнатуры методов, но такой подход помогает избежать проблем в будущем.

К примеру, вместо передачи нескольких строковых значений (или иного типа) определите тип с несколькими свойствами для единого параметра `Message`:

```
// Нежелательный способ: Метод RPC с несколькими параметрами
public void SendMessage(string to, string body)

// Рекомендуемый способ: один параметр, использующий сложный тип
public class Message
{
  public string To { get; set; }
  public string Body { get; set; }
}

public void SendMessage(Message message)
```

Преимущество такого подхода в возможности вносить изменения в будущем, например добавлять третье свойство — `title`. В нежелательном примере, если в дальнейшем потребуется добавить заголовок `title`, придется дописывать новый строковый параметр. При этом все существующие клиенты, вызывающие метод без параметра `title`, получат ошибки. Код из рекомендуемого способа не нарушит сигнатуру метода, поэтому существующие клиенты смогут продолжать вызывать его, как и до изменения. Если клиент не передает `title`, на сервере оно просто будет `null`. Можно обработать это значение, например установив заголовок сообщения по умолчанию.

SignalR сериализует параметры метода в JSON, поэтому все вложенные объекты доступны в JavaScript-клиентах.

Теперь, разобравшись с основными принципами SignalR и лучшими приемами разработки методов, рассмотрим, как создать сервис живого общения.

Создание сервиса живого общения в SignalR

В ASP.NET Core серверная библиотека SignalR включена в состав фреймворка, но клиентская библиотека JavaScript в проект не добавляется автоматически. SignalR поддерживает различные платформы, и веб-страница с JavaScript — лишь один из возможных клиентов.

Для загрузки клиентской библиотеки воспользуемся инструментом *Library Manager CLI*. Мы будем получать файлы из сети доставки содержимого *unpkg (Content Delivery Network, CDN)*, выдающей пакеты из репозитория Node.js.

Добавим серверный хаб SignalR и клиентский JavaScript в проект MVC, чтобы реализовать функцию чата. Пользователи смогут отправлять сообщения:

- всем, кто в данный момент находится на сайте;
- динамически создаваемым группам;
- определенному пользователю.

Дельный совет: в рабочем проекте центр SignalR лучше вынести в отдельный веб-проект. Это упростит масштабирование и не создаст излишнюю нагрузку на основной сайт. Живое общение требует большого количества соединений, что может повлиять на производительность всего веб-приложения.

Определение общих моделей

Определим две модели, которые можно использовать как на сервере, так и в клиентских .NET-приложениях, работающих с сервисом чата.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта Class Library (Библиотека классов);
   - в поле Solution name (Имя решения) укажите значение `Chapter11`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Common`.
2. В проекте `Northwind.Common` переименуйте файл `Class1.cs` в `UserModel.cs`.
3. Удалите содержимое переименованного файла и создайте в нем модель пользователя с полями:
   - имя пользователя;
   - уникальный идентификатор соединения;
   - список групп, в которые входит пользователь:

```
namespace Northwind.Chat.Models;

public class UserModel
{
  public string Name { get; set; } = null!;
  public string ConnectionId { get; set; } = null!;
  public string? Groups { get; set; } // список, разделенный запятыми
}
```

Дельный совет: в реальном приложении для свойства Groups лучше использовать коллекцию строк, но в данном случае сосредоточимся на изучении SignalR. Поэтому в интерфейсе будем использовать простое текстовое поле.

4. В папке проекта `Northwind.Common` создайте файл класса `MessageModel.cs`. Удалите его содержимое и определите в нем модель сообщения с полями:
   - `To` — кому отправляется сообщение;
   - `From` — от кого отправлено;
   - `Body` — текст сообщения:

```
namespace Northwind.Chat.Models;

public class MessageModel
{
```

```
    public string From { get; set; } = null!;
    public string To { get; set; } = null!;
    public string? Body { get; set; }
  }
```

Включение центра SignalR на сервере

Далее мы включим центр SignalR на стороне сервера, создав для этого проект MVC ASP.NET Core.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта ASP.NET Core Web App (Model-View-Controller) (Веб-приложение ASP.NET Core (Модель — представление — контроллер)) / `mvc`;
   - в поле Solution name (Имя решения) укажите значение `Chapter11`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.SignalR.Service.Client.Mvc`;
   - в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
   - установите флажок Configure for HTTPS (Настроить для HTTPS);
   - снимите флажок Enable Docker (Включить поддержку контейнера);
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня).
2. В файле созданного проекта добавьте строгую обработку предупреждений и ссылку на проект `Northwind.Common`:

```
<ItemGroup>
  <ProjectReference Include="..\Northwind.Common\Northwind.Common.csproj" />
</ItemGroup>
```

3. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5111 для протокола HTTPS и 5112 — для HTTP:

```
"https": {
  "commandName": "Project",
  "dotnetRunMessages": true,
  "launchBrowser": true,
  "applicationUrl": "https://localhost:5111;http://localhost:5112",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
}
```

4. В проекте `Northwind.SignalR.Service.Client.Mvc` создайте файл класса `ChatHub.cs`.

5. Измените содержимое созданного файла, унаследовав класс от Hub, и добавив два метода, доступных для вызова клиентом:

```
using Microsoft.AspNetCore.SignalR; // Для объекта Hub
using Northwind.Chat.Models; // Для классов UserModel, MessageModel

namespace Northwind.SignalR.Service.Hubs;

public class ChatHub : Hub
{
  // Для обработки каждого метода создается экземпляр класса
  // ChatHub, поэтому мы должны хранить имена пользователей,
  // идентификаторы соединений и групп в статическом поле
  private static Dictionary<string, UserModel> Users = new();

  public async Task Register(UserModel newUser)
  {
    UserModel user;
    string action = "registered as a new user";

  // Попытка извлечь сохраненного пользователя,
  // совпавшего с новым пользователем
  if (Users.ContainsKey(newUser.Name))
  {
    user = Users[newUser.Name];

    // Удаление всех существующих регистраций групп
    if (user.Groups is not null)
    {
      foreach (string group in user.Groups.Split(','))
      {
        await Groups.RemoveFromGroupAsync(user.ConnectionId, group);
      }
    }
    user.Groups = newUser.Groups;

    // Идентификатор соединения мог измениться, если браузер
    // обновился, поэтому обновите значение ConnectionId
    user.ConnectionId = Context.ConnectionId;

    action = "updated your registered user";
  }
  else
  {
    if (string.IsNullOrEmpty(newUser.Name))
    {
      // Назначение GUID для имени, если пользователь анонимный
      newUser.Name = Guid.NewGuid().ToString();
    }
    newUser.ConnectionId = Context.ConnectionId;
    Users.Add(key: newUser.Name, value: newUser);
```

```
    user = newUser;
  }

  if (user.Groups is not null)
  {
    // Пользователь не должен состоять ни в каких группах,
    // но если состоит, регистрируется с помощью объекта Hub

    foreach (string group in user.Groups.Split(','))
    {
      await Groups.AddToGroupAsync(user.ConnectionId, group);
    }
  }

  // Отправка зарегистрировавшемуся пользователю сообщения об успехе

  MessageModel message = new()
  {
    From = "SignalR Hub", To = user.Name,
    Body = string.Format(
      "You have successfully {0} with connection ID {1}.",
      arg0: action, arg1: user.ConnectionId)
    };

    IClientProxy proxy = Clients.Client(user.ConnectionId);
    await proxy.SendAsync("ReceiveMessage", message);
  }

  public async Task SendMessage(MessageModel message)
  {
    IClientProxy proxy;

    if (string.IsNullOrEmpty(message.To))
    {
      message.To = "Everyone";
      proxy = Clients.All;
      await proxy.SendAsync("ReceiveMessage", message);
      return;
    }

    // Разделение свойства To на список имен пользователей и групп
    string[] userAndGroupList = message.To.Split(',');

    // Каждый элемент может быть именем пользователя или группы
    foreach (string userOrGroup in userAndGroupList)
    {
      if (Users.ContainsKey(userOrGroup))
      {
        // Если элемент находится в разделе Users (Пользователи),
        // то этому пользователю отправляется сообщение путем поиска
        // идентификатора соединения в словаре
        message.To = $"User: {Users[userOrGroup].Name}";
```

```
        proxy = Clients.Client(Users[userOrGroup].ConnectionId);
      }
      else // Предположим, что элемент — это имя группы,
           // которой необходимо отправить сообщение
      {
        message.To = $"Group: {userOrGroup}";
        proxy = Clients.Group(userOrGroup);
      }
      await proxy.SendAsync("ReceiveMessage", message);
    }
  }
}
```

Особенности реализации

- В класс `ChatHub` добавлено приватное поле для хранения зарегистрированных пользователей. Это словарь, где уникальным ключом является имя пользователя.
- В классе `ChatHub` есть два метода, доступных для вызова клиентом: `Register` и `SendMessage`.
- Метод `Register` принимает единственный параметр, `UserModel`, в котором содержатся имя, идентификатор соединения и список групп. Эти данные сохраняются в статическом словаре, чтобы в будущем можно было отправлять сообщения конкретному пользователю. После регистрации или обновления данных клиент получает подтверждение об успешной операции.
- Метод `SendMessage` принимает объект `MessageModel`. Поведение метода зависит от значения свойства `To`:
  - если свойство `To` пустое, сообщение отправляется всем пользователям через прокси `Clients.All`;
  - если указано несколько получателей, строка `To` разбивается по запятой на массив, и для каждого элемента проверяется, зарегистрирован ли он в словаре `Users`:
    - если совпадение найдено, сообщение с помощью метода `Client` через прокси отправляется конкретному пользователю;
    - если совпадений нет, предполагается, что это имя группы, и сообщение с помощью метода `Group` через прокси отправляется всем ее участникам;
  - в завершение сообщение через прокси отправляется асинхронно.

6. В файле `Program.cs` импортируйте пространство имен, в котором находится центр SignalR:

```
using Northwind.SignalR.Service.Hubs; // Для класса ChatHub
```

7. В разделе конфигурации сервисов добавьте поддержку SignalR:

```
builder.Services.AddSignalR();
```

8. В разделе настройки HTTP-конвейера перед определением маршрутов добавьте оператор для сопоставления относительного URL-пути `/chat` с центром SignalR:

```
app.MapHub<ChatHub>("/chat");
```

После этих шагов сервис живого общения будет готов к работе.

Разработка веб-клиента на JavaScript в SignalR

Далее добавим на веб-страницу клиентскую JavaScript-библиотеку SignalR.

1. Откройте в терминале проект/папку `Northwind.SignalR.Service.Client.Mvc`.
2. Установите консольный инструмент Library Manager CLI:

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli
```

Этот инструмент уже может быть установлен. Чтобы обновить его до последней версии, выполните команду dotnet tool update -g Microsoft.Web.LibraryManager.Cli.

3. Добавьте в проект библиотеки `signalr.js` и `signalr.min.js` из сети `unpkg`:

```
libman install @microsoft/signalr@latest -p unpkg -d wwwroot/js/signalr --files
dist/browser/signalr.js --files dist/browser/signalr.min.js
```

Никогда не вставляйте в командную строку/терминал длинные команды, скопированные из PDF-файлов. Предварительно проверяйте их в простом текстовом редакторе — удаляйте лишние символы новой строки (переносы) и т. д. С целью упростить использование длинных команд я опубликовал их на сайте https://github.com/markjprice/apps-services-net8/blob/main/docs/command-lines.md.

4. Операция должна завершиться успешно:

```
wwwroot/js/signalr/dist/browser/signalr.js written to disk
wwwroot/js/signalr/dist/browser/signalr.min.js written to disk
Installed library "@microsoft/signalr@latest" to "wwwroot/js/signalr"
```

В графическом интерфейсе Visual Studio 2022 тоже можно добавлять клиентские JavaScript-библиотеки. Щелкните правой кнопкой мыши на проекте и выберите в контекстном меню команду Add ▸ Client Side Libraries (Добавить ▸ Клиентская библиотека).

Добавление на сайт страницы чата

Добавим функционал чата на главную страницу сайта.

1. Открыв расположенный в папке `Views/Home` файл `Index.cshtml`, введите в нем следующий код:

```
@using Northwind.Chat.Models
@{
  ViewData["Title"] = "SignalR Chat";
}
<div class="container">
  <h1>@ViewData["Title"]</h1>
  <hr />
  <div class="row">
    <div class="col">
      <h2>Register User</h2>
      <div class="mb-3">
        <label for="myName" class="form-label">My name</label>
        <input type="text" class="form-control"
          id="myName" value="Alice" required />
      </div>
      <div class="mb-3">
        <label for="myGroups" class="form-label">My groups</label>
        <input type="text" class="form-control"
          id="myGroups" value="Sales,IT" />
      </div>
      <div class="mb-3">
        <input type="button" class="form-control"
          id="registerButton" value="Register User" />
      </div>
    </div>
    <div class="col">
      <h2>Send Message</h2>
      <div class="mb-3">
        <label for="from" class="form-label">From</label>
        <input type="text" class="form-control"
          id="from" value="Alice" readonly />
      </div>
      <div class="mb-3">
        <label for="to" class="form-label">To</label>
        <input type="text" class="form-control" id="to" />
      </div>
      <div class="mb-3">
        <label for="body" class="form-label">Body</label>
        <input type="text" class="form-control" id="body" />
      </div>
      <div class="mb-3">
        <input type="button" class="form-control"
          id="sendButton" value="Send Message" />
      </div>
```

```
    </div>
  </div>
  <div class="row">
    <div class="col">
      <hr />
      <h2>Messages received</h2>
      <ul id="messages"></ul>
    </div>
  </div>
</div>
<script src="~/js/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

Обратите внимание на следующее.

- На странице есть три ключевых раздела.
  - Register User — регистрация пользователя.
    - Поля ввода:
      - имя пользователя;
      - группы, разделенные запятыми.
    - Кнопка Register User, которая отправляет данные для регистрации.
  - Send Message — отправка сообщений.
    - Поля ввода:
      - имя отправителя;
      - получатели (имена пользователей или группы через запятую);
      - тело сообщения.
    - Кнопка Send Message, которая отправляет сообщение.
  - Messages received — просмотр полученных сообщений. Элемент списка, куда динамически добавляются входящие сообщения.
  - А также два элемента `script` — для подключения клиентской JavaScript-библиотеки SignalR и для JavaScript-реализации клиента чата.

2. В каталоге `wwwroot/js` создайте JS-файл `chat.js` со следующим содержимым:

```
"use strict";

var connection = new signalR.HubConnectionBuilder()
  .withUrl("/chat").build();

document.getElementById("registerButton").disabled = true;
document.getElementById("sendButton").disabled = true;

document.getElementById("myName").addEventListener("input",
  function () {
    document.getElementById("from").value =
```

```
    document.getElementById("myName").value;
  }
);

connection.start().then(function () {
  document.getElementById("registerButton").disabled = false;
  document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
  return console.error(err.toString());
});

connection.on("ReceiveMessage", function (received) {
  var li = document.createElement("li");
  document.getElementById("messages").appendChild(li);

  li.textContent =
    // В этой строке должны использоваться знаки грависа `
    // для поддержки интерполированной строки.
    // Если использовать одинарные кавычки ('), код не сработает!
    `To ${received.to}, From ${received.from}: ${received.body}`;
});

document.getElementById("registerButton").addEventListener("click",
  function (event) {
    var registermodel = {
      name: document.getElementById("myName").value,
      groups: document.getElementById("myGroups").value
    };
    connection.invoke("Register", registermodel).catch(function (err) {
      return console.error(err.toString());
    });
    event.preventDefault();
  });

document.getElementById("sendButton").addEventListener("click",
  function (event) {
    var messagemodel = {
      from: document.getElementById("from").value,
      to: document.getElementById("to").value,
      body: document.getElementById("body").value
    };
    connection.invoke("SendMessage", messagemodel).catch(function (err) {
      return console.error(err.toString());
    });
    event.preventDefault();
});
```

Особенности реализации

- Подключение к серверу конструируется через центр SignalR на сервере с относительным URL-путем `/chat`.
- Кнопки Register User и Send Message отключены до установления соединения.

- Поле My name синхронизируется обработчиком событий ввода с полем From.
- При вызове метода `ReceiveMessage` на сервере полученное сообщение добавляется в маркированный список `messages`. В элементе списка содержатся подробные данные сообщения, такие как поля `from`, `to` и `body`. Мы определили две модели на языке C#. Обратите внимание, что в JavaScript используется регистр camelCase, а в C# — PascalCase.

Сообщения форматируются с помощью интерполяции строк JavaScript. Для этого строки следует заключать в символы `, а динамические заполнители — в фигурные скобки со знаком доллара (${}).

- Кнопка Register User содержит обработчик события `click`, который создает объект регистрации (имя + группы), а затем вызывает метод `Register` на сервере.
- Кнопка Send Message содержит обработчик события `click`, создающий объект сообщения с параметрами `from`, `to` и `body`, а затем вызывает метод `SendMessage` на стороне сервера.

Тестирование функции чата

Попробуем обменяться сообщениями в чате между несколькими пользователями.

1. Запустите сайт проекта `Northwind.SignalR.Service.Client.Mvc` в режиме `https`.
   - В программе Visual Studio 2022: выберите профиль https на панели инструментов, а затем запустите проект `Northwind.SignalR.Service.Client.Mvc` без отладки.
   - В терминале Visual Studio Code выполните команду:

```
dotnet run --launch-profile https
```

   - Если появится запрос брандмауэра Windows Defender с блокировкой доступа, нажмите кнопку Allow access (Разрешить доступ).
2. Запустите браузер Chrome и перейдите на сайт `https://localhost:5111/`.
3. Обратите внимание, что имя `Alice` и название группы `Sales,IT` уже введены в соответствующие поля. Нажмите кнопку Register User. Вы увидите ответ центра SignalR, показанный на рис. 11.1.
4. Откройте новое окно (не вкладку) браузера Chrome или запустите другой браузер, например Firefox или Edge.
5. Перейдите на сайт `https://localhost:5111/`.
6. Введите имя `Bob` и название группы `Sales` в соответствующие поля, а затем нажмите кнопку Register User.

Рис. 11.1. Регистрация пользователя в чате

7. Снова откройте новое окно браузера Chrome или запустите другой браузер.
8. Перейдите на сайт `https://localhost:5111/`.
9. Введите имя `Charlie` и название группы `IT` в соответствующие поля, а затем нажмите кнопку Register User.
10. Расположите окна браузера так, чтобы видеть все три одновременно.

Для удобной организации окон можно воспользоваться диспетчером FancyZones из коллекции утилит PowerToys. См. https://learn.microsoft.com/ru-ru/windows/powertoys/.

11. В окне браузера пользователя `Alice`:
    - в поле To введите значение `Sales`;
    - в поле Body введите значение `Sell more!`.
12. Нажмите кнопку Send Message.
13. Пользователи `Alice` и `Bob` получат сообщение (рис. 11.2).
14. В окне браузера пользователя `Bob` введите следующее:
    - в поле To введите значение `IT`;
    - в поле Body введите значение `Fix more bugs!`.
15. Нажмите кнопку Send Message.
16. Пользователи с именами `Alice` и `Charlie` получат сообщение (рис. 11.3).

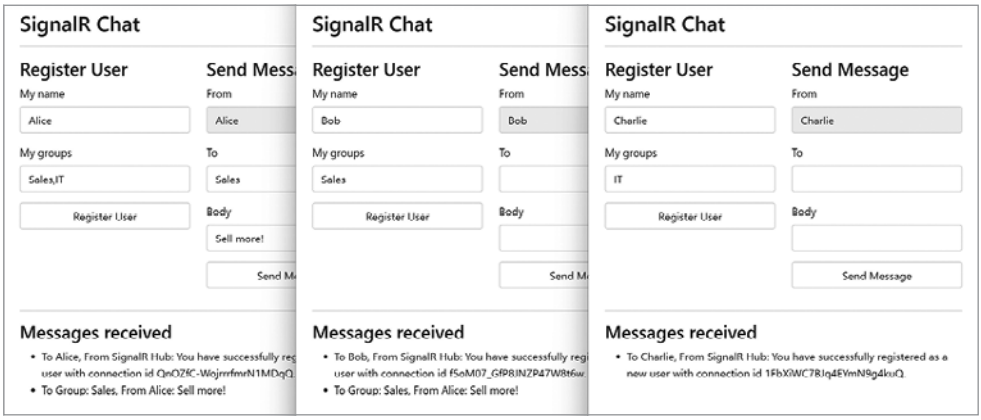


Рис. 11.2. Пользователь Alice отправляет сообщение в группу Sales

Рис. 11.3. Пользователь Bob отправляет сообщение группе IT

17. В окне браузера пользователя `Alice` введите следующее:
    - в поле To введите значение `Bob`;
    - в поле Body введите значение `Bonjour Bob!`.
18. Нажмите кнопку Send Message.
19. Сообщение получит только пользователь `Bob`.
20. В окне браузера пользователя `Charlie` введите следующее:
    - поле To оставьте пустым;
    - в поле Body введите значение `Everybody dance now!`.
21. Нажмите кнопку Send Message.

22. Сообщение получат все пользователи (рис. 11.4).

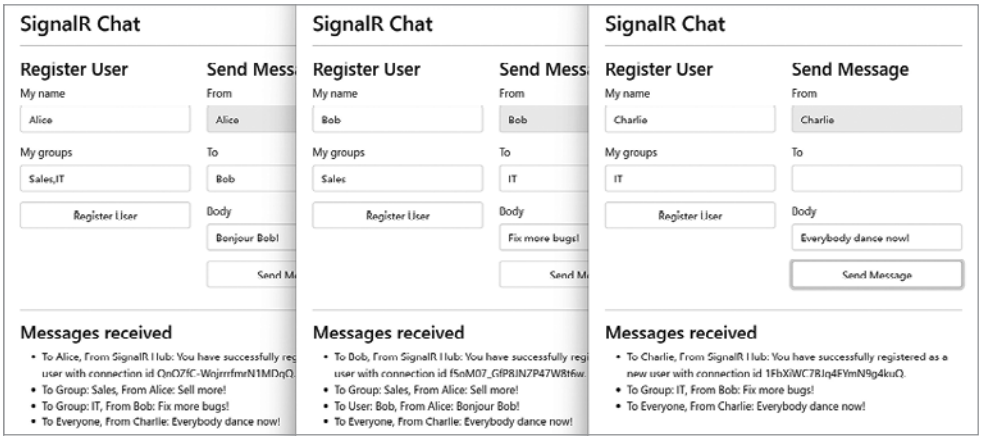


Рис. 11.4. Пользователь Charlie отправляет сообщение всем

23. В окне браузера пользователя `Charlie` введите следующее:
    - в поле To введите значение `HR,Alice`;
    - в поле Body введите значение `Is anyone in HR listening?`.
24. Нажмите кнопку Send Message.
25. Обратите внимание, что пользователь `Alice` получит сообщение, отправленное непосредственно ей, но, поскольку группы HR не существует, адресованное ее участникам сообщение не получит никто (рис. 11.5).
26. Закройте браузеры и завершите работу веб-сервера.

Рис. 11.5. Пользователь Charlie посылает сообщение пользователю Alice и несуществующей группе

Создание консольного клиента на .NET

Ранее мы рассмотрели сервис .NET с центром SignalR и клиент JavaScript, взаимодействующий с другими клиентами через этот центр. Теперь создадим клиент SignalR на .NET.

Создание .NET-клиента для SignalR

Мы воспользуемся консольным приложением, хотя для любого проекта .NET потребуются подключение тех же пакетов и реализация аналогичного кода.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта Console Application (Консольное приложение) / `console`;
   - в поле Solution name (Имя решения) укажите значение `Chapter11`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.SignalR.Client.Console`.
2. Подключите пакет клиента ASP.NET Core SignalR, добавьте ссылку на проект `Northwind.Common`, включите строгую обработку предупреждений и статически импортируйте класс `System.Console` на глобальном уровне:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.SignalR.Client" Version="8.0.0" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include="..\Northwind.Common\Northwind.Common.csproj" />
</ItemGroup>
```

3. Выполните сборку проекта, чтобы восстановить зависимости и пересобрать проекты, на которые он ссылается.
4. Удалите содержимое файла `Program.cs`, импортируйте в нем пространство имен для работы с SignalR в качестве клиента и моделями чата. Добавьте код для регистрации пользователя (в том числе в группе), подключения к центру SignalR и обработки сообщений:

```
using Microsoft.AspNetCore.SignalR.Client; // Для объекта HubConnection
using Northwind.Chat.Models; // Для классов UserModel, MessageModel

Write("Enter a username (required): ");
string? username = ReadLine();

if (string.IsNullOrEmpty(username))
{
  WriteLine("You must enter a username to register with chat!");
  return;
}
```

```
Write("Enter your groups (optional): ");
string? groups = ReadLine();

HubConnection hubConnection = new HubConnectionBuilder()
  .WithUrl("https://localhost:5111/chat")
  .Build();

hubConnection.On<MessageModel>("ReceiveMessage", message =>
{
  WriteLine($"To {message.To}, From {message.From}: {message.Body}");
});

await hubConnection.StartAsync();

WriteLine("Successfully started.");

UserModel registration = new()
{
  Name = username,
  Groups = groups
};
await hubConnection.InvokeAsync("Register", registration);

WriteLine("Successfully registered.");
WriteLine("Listening... (press ENTER to stop.)");

ReadLine();
```

Тестирование консольного клиента .NET

Запустим сервис SignalR и вызовем его из консольного приложения.

1. Запустите сайт проекта `Northwind.SignalR.Service.Client.Mvc` в режиме `https` без отладки.
2. Откройте браузер Chrome и перейдите на сайт `https://localhost:5111/`.
3. Нажмите кнопку Register User.
4. Запустите проект `Northwind.SignalR.Client.Console`.
5. Введите имя пользователя и название группы, например `Sales,Admins`.
6. Разместите окна браузера и консоли так, чтобы они были видны одновременно.
7. В браузере пользователя `Alice` введите следующее:
   - в поле To введите значение `Sales`;
   - в поле Body введите значение `Go team!`.
8. Нажмите кнопку Send Message и убедитесь, что сообщение получили пользователь `Alice` и консольный клиент (рис. 11.6).

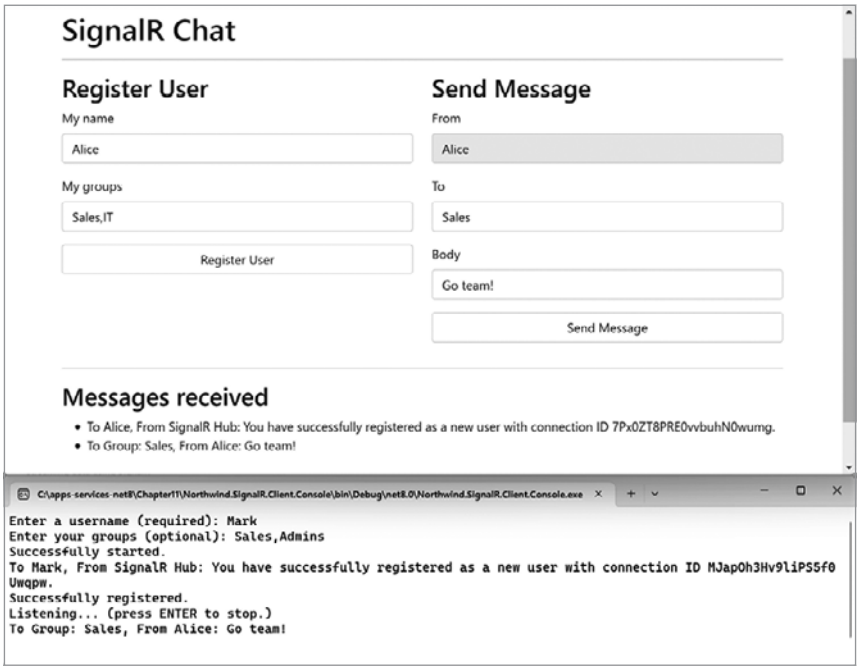


Рис. 11.6. Пользователь Alice отправляет сообщение в группу Sales, в том числе консольному клиенту

9. В консольном приложении нажмите клавишу Enter, чтобы завершить прослушивание.
10. Закройте Chrome и завершите работу веб-сервера.

Потоковая передача данных с помощью SignalR

Итак, SignalR способен передавать структурированные сообщения одному или нескольким клиентам. Отличный способ для работы с небольшими объемами структурированных и полноценных данных. Но как быть с данными, поступающими фрагментарно на протяжении длительного времени?

В таких случаях можно использовать потоки. SignalR поддерживает работу между сервисом и клиентом в обе стороны (загрузка данных из потока/в поток).

В версии C# 8 и новее метод центра для потоковой загрузки данных должен возвращать параметр `IAsyncEnumerable<T>`, а для потоковой передачи данных — принимать параметр `IAsyncEnumerable<T>`. В более старых версиях языка используются параметры `ChannelReader<T>` и `ChannelReader<T>` соответственно.

Определение центра для потоковой передачи

В рамках практикума добавим несколько потоковых методов.

1. В проекте `Northwind.Common` создайте файл `StockPrice.cs` и определите в нем модель данных для цен акций:

```
namespace Northwind.SignalR.Streams;

public record StockPrice(string Stock, double Price);
```

2. Выполните сборку проекта `Northwind.SignalR.Service.Client.Mvc`, чтобы обновить ссылки на проекты.
3. В этом проекте создайте файл класса `StockPriceHub.cs` и добавьте в него центр с двумя потоковыми методами:

```
using Microsoft.AspNetCore.SignalR; // Для класса Hub
using System.Runtime.CompilerServices; // Для атрибута [EnumeratorCancellation]
using Northwind.SignalR.Streams; // Для типа StockPrice

namespace Northwind.SignalR.Service.Hubs;

public class StockPriceHub : Hub
{
  public async IAsyncEnumerable<StockPrice> GetStockPriceUpdates(
    string stock,
    [EnumeratorCancellation] CancellationToken cancellationToken)
  {

    double currentPrice = 267.10; // Имитация начальной цены

    for (int i = 0; i < 10; i++)
    {
      // Токен отмены регулярно проверяется, чтобы сервер остановил
      // производство элементов, если клиент отсоединяется
      cancellationToken.ThrowIfCancellationRequested();

      // Увеличение или уменьшение текущей цены на случайную величину.
      // Компилятору не нужны лишние скобки, но для восприятия человеком
      // будет лучше, если их добавить
      currentPrice += (Random.Shared.NextDouble() * 10.0) - 5.0;

      StockPrice stockPrice = new(stock, currentPrice);

      Console.WriteLine("[{0}] {1} at {2:C}",
        DateTime.UtcNow, stockPrice.Stock, stockPrice.Price);

      yield return stockPrice;
```

```
        await Task.Delay(4000, cancellationToken); // миллисекунды
      }
    }

    public async Task UploadStocks(IAsyncEnumerable<string> stocks)
    {
      await foreach (string stock in stocks)
      {
        Console.WriteLine($"Receiving {stock} from client...");
      }
    }
  }
```

4. В файле `Program.cs` проекта `Northwind.SignalR.Service.Client.Mvc` зарегистрируйте новый центр котировок после регистрации чата:

```
app.MapHub<StockPriceHub>("/stockprice");
```

Создание .NET-клиента для потоковой передачи

Теперь мы создадим клиент для загрузки/выгрузки данных из центра SignalR и обратно.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта Console Application (Консольное приложение) / `console`;
   - в поле Solution name (Имя решения) укажите значение `Chapter11`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.SignalR.Client.Console.Streams`.
2. В файле созданного проекта подключите строгую обработку предупреждений, пакет клиента ASP.NET Core SignalR, ссылку на проект `Northwind.Common`, а также статически импортируйте класс `System.Console` на глобальном уровне:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.SignalR.Client"
Version="8.0.0" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include="..\Northwind.Common\Northwind.Common.csproj" />
</ItemGroup>
```

3. В проекте `Northwind.SignalR.Client.Console.Streams` создайте файл класса `Program.Methods.cs`, удалите его содержимое, а затем в разделяемом классе

Program добавьте статические методы для генерации десяти случайных четырехбуквенных кодов акций:

```
// Определяется в пустом пространстве имен по умолчанию для слияния
// с автоматически сгенерированным разделяемым классом Program

partial class Program
{
  static async IAsyncEnumerable<string> GetStocksAsync()
  {
    for (int i = 0; i < 10; i++)
    {
      // Возврат случайного четырехбуквенного кода акции
      yield return $"{AtoZ()}{AtoZ()}{AtoZ()}{AtoZ()}";

      await Task.Delay(TimeSpan.FromSeconds(3));
    }
  }

  static string AtoZ()
  {
    return char.ConvertFromUtf32(Random.Shared.Next(65, 91));
  }
}
```

4. Удалите содержимое файла Program.cs в папке проекта Northwind.SignalR.Client.Console.Streams. Импортируйте пространство имен клиента SignalR, добавьте код для приема пользовательского ввода (кодов акций), подключения к центру, обработки потоков цен на акции и отправки асинхронных данных в центр:

```
using Microsoft.AspNetCore.SignalR.Client; // Для объекта HubConnection
using Northwind.SignalR.Streams; // Для типа StockPrice

Write("Enter a stock (press Enter for MSFT): ");
string? stock = ReadLine();

if (string.IsNullOrEmpty(stock))
{
  stock = "MSFT";
}

HubConnection hubConnection = new HubConnectionBuilder()
  .WithUrl("https://localhost:5111/stockprice")
  .Build();

await hubConnection.StartAsync();

try
{
  CancellationTokenSource cts = new();
```

```
    IAsyncEnumerable<StockPrice> stockPrices =
      hubConnection.StreamAsync<StockPrice>(
        "GetStockPriceUpdates", stock, cts.Token);

    await foreach (StockPrice sp in stockPrices)
    {
      WriteLine($"{sp.Stock} is now {sp.Price:C}.");

      Write("Do you want to cancel (y/n)? ");
      ConsoleKey key = ReadKey().Key;
      if (key == ConsoleKey.Y)
      {
        cts.Cancel();
      }
      WriteLine();
    }
  }
  catch (Exception ex)
  {
    WriteLine($"{ex.GetType()} says {ex.Message}");
  }
  WriteLine();

  WriteLine("Streaming download completed.");

  await hubConnection.SendAsync("UploadStocks", GetStocksAsync());

  WriteLine("Uploading stocks to service... (press ENTER to stop.)");
  ReadLine();

  WriteLine("Ending console app.");
```

Проверка потокового сервиса и клиента

Итак, мы готовы протестировать передачу потоковых данных.

1. Запустите сайт проекта `Northwind.SignalR.Service.Client.Mvc` в режиме `https` без отладки.
2. Запустите консольное приложение `Northwind.SignalR.Client.Console.Streams` без отладки.
3. Разместите окна консоли с сайтом MVC ASP.NET Core и клиентским приложением так, чтобы видеть их оба.
4. В окне с клиентским приложением нажмите клавишу Enter, чтобы отобразить котировку акций компании Microsoft:

```
Enter a stock (press Enter for MSFT):
MSFT is now £265.00.
Do you want to cancel (y/n)?
```

5. В окне консоли с сайтом подождите около 10 секунд — появятся сгенерированные цены акций, но они пока не отправлены клиенту:

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5131
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5132
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: C:\apps-services-net7\Chapter13\Northwind.
SignalR.Service.Client.Mvc
[12/09/2022 17:52:26] MSFT at £265.00
[12/09/2022 17:52:30] MSFT at £260.78
[12/09/2022 17:52:34] MSFT at £264.86
[12/09/2022 17:52:38] MSFT at £262.10
```

6. В окне с клиентским приложением нажмите клавишу N, чтобы обновить котировки. Продолжайте нажимать клавишу N, пока котировки не будут отправлены из центра SignalR клиенту. Затем нажмите клавишу Y, чтобы сервис SignalR остановился, а клиент начал загрузку котировок:

```
MSFT is now £260.78.
Do you want to cancel (y/n)? n
MSFT is now £264.86.
Do you want to cancel (y/n)? n
MSFT is now £262.10.
Do you want to cancel (y/n)? y
System.Threading.Tasks.TaskCanceledException says A task was canceled.

Streaming download completed.
Uploading stocks to service... (press ENTER to stop.)
```

7. В окне консоли с сайтом убедитесь, что получены случайные коды акций:

```
Receiving PJON from client...
Receiving VWJD from client...
Receiving HMOJ from client...
Receiving QQMQ from client...
```

8. Закройте оба окна консоли.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 11.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. Какие протоколы передачи используются в SignalR? А какой из них по умолчанию?
2. Каковы принципы проектирования сигнатур методов RPC?
3. Как загрузить JavaScript-библиотеку SignalR?
4. Что произойдет при отправке сообщения SignalR клиенту с несуществующим идентификатором соединения?
5. Каковы преимущества отделения сервиса SignalR от других компонентов ASP.NET Core?

Упражнение 11.2. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-11---broadcasting-real-time-communication-using-signalr.

Резюме

В этой главе мы рассмотрели:

- историю развития технологий, предшествовавших SignalR;
- принципы работы SignalR;
- реализацию чата на основе SignalR, включая создание центра, клиента JavaScript и консольного .NET-клиента;
- потоковую загрузку и передачу данных через SignalR.

В следующей главе речь пойдет о GraphQL — стандарте, который позволяет клиенту управлять данными, возвращаемыми сервисом.

12 Объединение источников данных с помощью GraphQL

В этой главе вы познакомитесь с GraphQL — технологией сервисов, предлагающей современный подход к объединению данных из разных источников и предоставляющей стандартный механизм их запроса.

Основы GraphQL

В главе 8 рассматривалось создание веб-API с сопоставлением конечных точек запросов с лямбда-выражениями или методами, возвращающими ответ. Формат ответов и параметры находятся под контролем сервиса, поэтому клиент не может запрашивать только необходимые ему данные или выбирать более эффективные форматы.

Такая технология, как OData, содержит встроенный язык запросов, позволяющий клиенту управлять данными, которые он хочет получить. Однако этот протокол привязан к стандарту HTTP, например к использованию строк запроса, а его подход считается устаревшим.

GraphQL — это более современная и гибкая технология для объединения и представления данных в виде сервиса. Как и OData, GraphQL определяет структуру данных и способ их запроса, но при этом клиент получает больше контроля.

GraphQL разработан в компании Facebook в 2012 году, а в 2015-м был открыт и передан под управление GraphQL Foundation.

Вот некоторые ключевые преимущества GraphQL перед OData.

- Не зависит от протокола HTTP. GraphQL не привязан к конкретному транспортному протоколу и может работать через WebSockets, TCP и другие варианты.
- Большое количество клиентских библиотек. GraphQL поддерживается на большем числе платформ, чем OData.
- Одна конечная точка. Обычно GraphQL-сервис имеет единственный URL-адрес, например `/graphql`.

Формат документа GraphQL-запроса

В GraphQL используется собственный формат документов для запросов, но без запятых между именами полей. Пример запроса для получения данных о товаре с идентификатором 23 показан ниже:

```
{
  product (productId: 23) {
    productId
    productName
    unitPrice
    supplier {
      companyName
      country
    }
  }
}
```

Официальный формат данных в документах GraphQL-запросов — application/graphql.

Запрашиваемые поля

Простейший GraphQL-запрос извлекает несколько полей сущности, например данные о клиентах:

```
# Ключевое слово query необязательно. Комментарии отмечаются префиксом #
query {
  customer {
    customerId
    companyName
    country
  }
}
```

Ответ приходит в формате JSON в виде массива объектов:

```
{
  "data": [
    {
      "customerId": "ALFKI",
      "companyName": "Alfreds Futterkiste",
      "country": "Germany"
    },
  ...
  ]
}
```

Фильтрация и аргументы

При работе с HTTP или REST API клиент может передавать параметры только в том случае, если они заранее определены в API. В GraphQL параметры можно задавать в любом месте запроса. Например, отбор заказов по дате и стране клиента:

```
query GetOrdersByDateAndCountry {
  order(orderDate: "23/04/1998") {
    orderId
    orderDate
    customer(country: "UK") {
      companyName
      country
    }
  }
}
```

Обратите внимание, что, хотя в GraphQL используется стиль camelCase для имен сущностей, свойств и параметров, для имен запросов следует использовать стиль PascalCase.

Можно передавать параметры динамически:

```
query GetOrdersByDateAndCountry($country: String, $orderDate: String) {
  order(orderDate: $orderDate) {
    orderId
    orderDate
    customer(country: $country) {
      companyName
      country
    }
  }
}
```

Подробнее о языке GraphQL-запросов см. на сайте по адресу https://graphql.org/learn/queries/.

Другие возможности GraphQL

Помимо стандартных запросов, GraphQL поддерживает:

- *мутации* — операции создания, обновления и удаления данных;
- *подписки* — механизм получения уведомлений об изменениях ресурсов, который особенно эффективен при использовании WebSockets.

Платформа ChilliCream GraphQL

Одна из самых известных библиотек для работы с GraphQL в .NET — *GraphQL.NET*. Однако настроить ее довольно сложно, а документация оставляет желать лучшего. Эта библиотека строго следует спецификации GraphQL без реализации упрощенных подходов для начинающих пользователей платформы .NET.

Ознакомиться с GraphQL.NET можно по адресу https://graphql-dotnet.github.io/.

В ходе поиска альтернатив я нашел решение компании ChilliCream, разработавшей целую экосистему для работы с GraphQL:

- *Hot Chocolate* — создание GraphQL-сервисов на .NET;
- *Strawberry Shake* — создание клиентов GraphQL для .NET;
- *Banana Cake Pop* — инструмент для выполнения запросов и исследования API GraphQL с помощью специальной среды разработки на основе `Monaco`;
- *Green Donut* — механизм кэширования и загрузки данных с высокой производительностью.

В отличие от некоторых других пакетов, используемых для поддержки GraphQL, пакеты ChilliCream ориентированы на простоту внедрения. В них используются стандартные классы POCO и соглашения вместо сложных схем и множества конфигурационных файлов. По своей концепции эта платформа напоминает инструменты Microsoft .NET — оптимальные дефолтные параметры и соглашения, а не масса шаблонного кода и конфигураций.

На сайте ChilliCream указано: *«Мы создаем конечную платформу GraphQL. Большая часть нашего кода открыта и останется таковой навсегда»*.

GitHub-репозиторий для Hot Chocolate расположен по адресу https://github.com/ChilliCream/hotchocolate.

Создание сервиса с поддержкой GraphQL

Проект GraphQL невозможно создать командой `dotnet new`, поэтому мы воспользуемся шаблоном ASP.NET Core Empty (Пустой шаблон ASP.NET Core). Хотя GraphQL не требует размещения на веб-сервере, так как не привязан к протоколу HTTP, это удобный вариант для начала работы. Далее добавим пакет для поддержки GraphQL.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта ASP.NET Core Empty (Пустой шаблон ASP.NET Core) / `web`;
   - в поле Solution name (Имя решения) укажите значение `Chapter12`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.GraphQL.Service`;
   - другие параметры в программе Visual Studio 2022:
     - установите флажок Configure for HTTPS (Настроить для HTTPS);
     - снимите флажок Enable Docker (Включить поддержку контейнера);
     - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня).

2. В файле проекта подключите пакет Hot Chocolate, предназначенный для работы с GraphQL в ASP.NET Core:

```
<ItemGroup>
  <PackageReference Include="HotChocolate.AspNetCore" Version="13.5.1" />
</ItemGroup>
```

На момент написания книги актуальной считалась предрелизная версия 13.6.0. Чтобы установить самую свежую предварительную версию, следует использовать параметр Version="13.6-*". Но я все же рекомендую перейти по ссылке https://www.nuget.org/packages/HotChocolate.AspNetCore/ и установить актуальную общедоступную (GA) версию.

3. В файле проекта включите строгую обработку предупреждений и отключите предупреждение `AD0001`:

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <Nullable>enable</Nullable>
  <ImplicitUsings>enable</ImplicitUsings>
  <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  <NoWarn>AD0001</NoWarn>
</PropertyGroup>
```

4. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5121 для протокола HTTPS и 5122 — для HTTP. Кроме того, присвойте параметру `launchUrl` значение `graphql`:

```
"https": {
  "commandName": "Project",
  "dotnetRunMessages": true,
  "launchBrowser": true,
  "launchUrl": "graphql",
  "applicationUrl": "https://localhost:5121;http://localhost:5122",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
  }
```

5. Выполните сборку проекта `Northwind.GraphQL.Service`.

Определение схемы GraphQL для проекта с приветствием

Первый шаг — определить, какие GraphQL-модели будут представлены в веб-сервисе. Для начала создадим простейший запрос `Hello World!`, который будет возвращать текст приветствия.

1. В проекте/папке `Northwind.GraphQL.Service` создайте файл класса `Query.cs`.
2. Измените его содержимое, добавив метод `GetGreeting`, возвращающий строку `"Hello, World!"`:

```
namespace Northwind.GraphQL.Service;

public class Query
{
  public string GetGreeting() => "Hello, World!";
}
```

3. В файле `Program.cs` импортируйте пространство имен, в котором определен класс `Query`:

```
using Northwind.GraphQL.Service; // Для класса Query
```

4. В разделе настройки сервисов после вызова метода `CreateBuilder` добавьте код для поддержки GraphQL на стороне сервера и зарегистрируйте тип запроса:

```
builder.Services
  .AddGraphQLServer()
  .AddQueryType<Query>();
```

5. Измените обработку `GET`-запросов, чтобы возвращался более информативный текст:

```
app.MapGet("/", () => "Navigate to: https://localhost:5121/graphql");
```

6. В разделе настройки HTTP-конвейера перед вызовом метода `Run` добавьте GraphQL как конечную точку:

```
app.MapGraphQL();
```

7. Запустите веб-проект `Northwind.GraphQL.Service` в режиме `https` без отладки.
   - В Visual Studio 2022 выберите профиль https, запустите проект без отладки и обратите внимание, что браузер откроется автоматически.
   - В Visual Studio Code выполните команду `dotnet run --launch-profile https`, запустите браузер Chrome вручную и перейдите по адресу https://localhost:5121/graphql.
8. Ознакомьтесь с интерфейсом инструмента Banana Cake Pop, а затем нажмите кнопку Create Document (Создать документ).
9. В правом верхнем углу страницы нажмите кнопку Connection Settings (Параметры подключения), показанную на рис. 12.1.

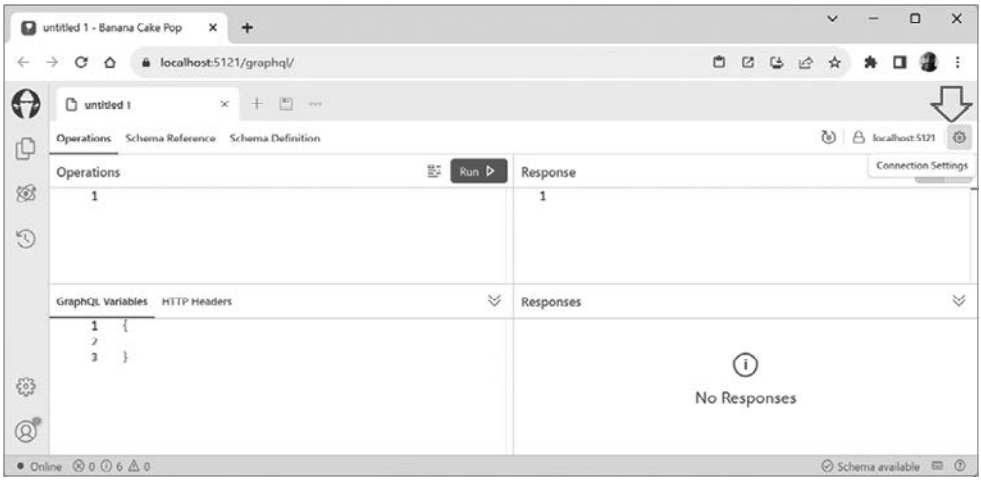


Рис. 12.1. Созданный документ BananaCakePop и кнопка настройки соединения

10. В появившемся окне убедитесь, что конечная точка в поле Schema Endpoint (Конечная точка схемы) указана верно, а затем нажмите кнопку Cancel (Отмена) (рис. 12.2).

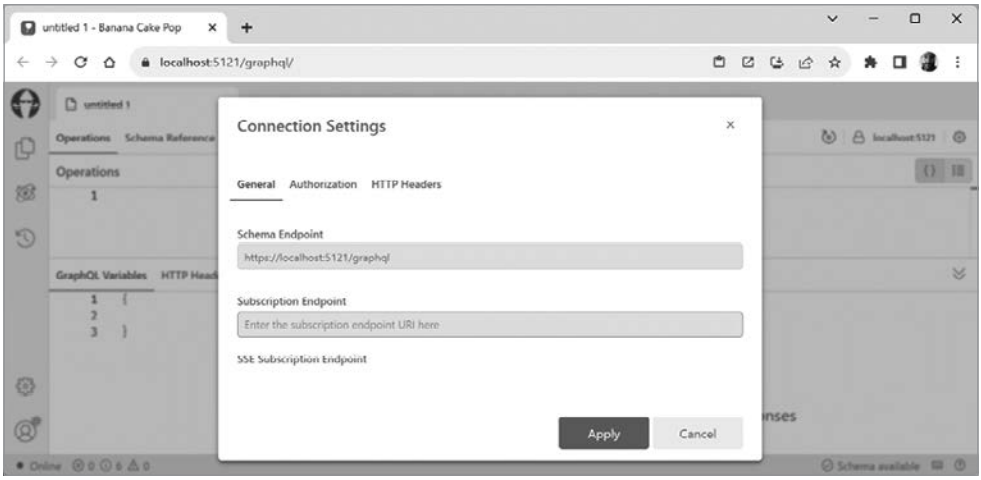


Рис. 12.2. Настройки подключения в интерфейсе BananaCakePop

11. В верхней части страницы с документом `untitled 1` перейдите на вкладку Schema Reference (Описание схемы).

12. На этой вкладке обратите внимание, что `Query` — это специальный тип, определяющий точку входа каждого GraphQL-запроса. В нем есть поле `greeting`, возвращающее значение типа `String!`. Восклицательный знак означает, что значение не может быть `null`.
13. Перейдите на вкладку Schema Definition (Определение схемы) и убедитесь, что в схеме присутствует только один объект — `Query`, содержащий поле `greeting` со строковым значением, не равным `null`:

```
type Query {
  greeting: String!
}
```

Составление и выполнение GraphQL-запросов

Теперь, когда схема известна, напишем и выполним запрос.

1. На странице с документом `untitled 1` инструмента Banana Cake Pop перейдите на вкладку Operations (Операции).
2. В левой части страницы введите символ `{`. Закрывающая фигурная скобка будет вставлена автоматически.
3. Введите `g`, после чего система предложит добавить поле `greeting` (рис. 12.3).
4. Нажмите клавишу Enter, чтобы принять предложение.

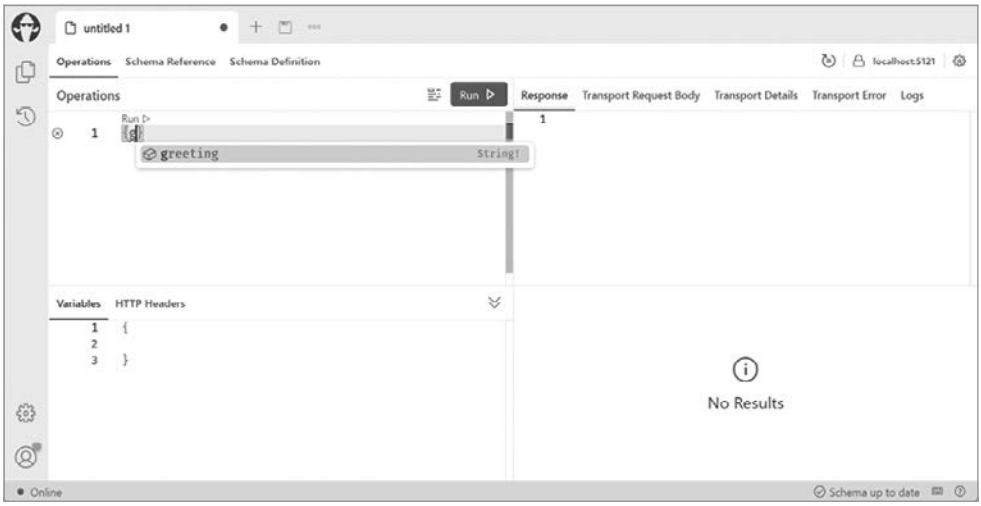


Рис. 12.3. Функция автодополнения — предложено поле greeting

5. Нажмите кнопку Run (Запуск) и проанализируйте результат:

```
{
  "data": {
    "greeting": "Hello, World!"
  }
}
```

6. Закройте браузер Chrome и завершите работу веб-сервера.

Именование GraphQL-запросов и операций

Итак, вы написали безымянный запрос, но можно создать и именованный:

```
query QueryNameGoesHere {
  greeting
}
```

Именованные запросы удобно использовать для идентификации запросов и их мониторинга, особенно в облачных сервисах Microsoft Azure с помощью инструмента Application Insights.

Оформление полей в GraphQL

Метод C#, который мы добавили в класс `Query`, назывался `GetGreeting`, но при запросе использовалось слово `greeting`. В GraphQL префикс `Get` в именах методов указывать не обязательно. Рассмотрим дополнительные примеры.

1. В файле `Query.cs` добавьте два метода без префикса `Get` (выделено в коде ниже):

```
namespace Northwind.GraphQL.Service;

public class Query
{
  public string GetGreeting() => "Hello, World!";
  public string Farewell() => "Ciao! Ciao!";
  public int RollTheDie() => Random.Shared.Next(1, 7);
}
```

2. Запустите проект `Northwind.GraphQL.Service` в режиме `https` без отладки.
3. Перейдите на вкладку Schema Definition (Определение схемы) и зафиксируйте обновленную схему:

```
type Query {
  greeting: String!
  farewell: String!
  rollTheDie: Int!
}
```

Обратите внимание, что, хотя в GraphQL-полях используется стиль camelCase, имена методов C# следует оформлять в стиле PascalCase.

4. Перейдите на вкладку Operations (Операции) и измените запрос, добавив поле `rollTheDie`:

```
query GetNumber {
  rollTheDie
}
```

5. Нажмите кнопку Run (Запуск) несколько раз. Обратите внимание, что ответы содержат случайные числа от 1 до 6, а история запросов и ответов сохраняется в текущей сессии браузера (рис. 12.4):

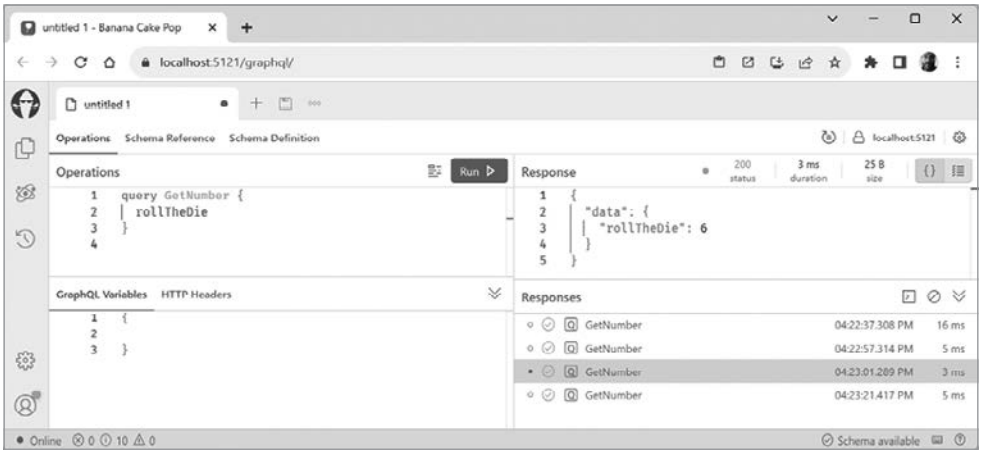


Рис. 12.4. Выполнение именованного запроса и история запросов и ответов

6. Закройте браузер Chrome и завершите работу веб-сервера.

Определение GraphQL-запросов для моделей EF Core

Теперь, когда базовый GraphQL-сервис успешно работает, давайте расширим его, добавив возможность запросов к БД Northwind.

Добавление поддержки EF Core

Для интеграции контекста БД EF Core с GraphQL-запросами потребуется добавить еще один пакет Hot Chocolate.

1. Добавьте пакет для интеграции Hot Chocolate с EF Core и укажите ссылку на проект контекста БД Northwind:

```
<ItemGroup>
  <PackageReference Include="HotChocolate.AspNetCore" Version="13.5.1" />
```

```
  <PackageReference Include="HotChocolate.Data.EntityFramework"
➥ Version="13.5.1" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.DataContext.
➥ SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
</ItemGroup>
```

Путь к проекту не должен содержать символы разрыва строки. Все пакеты Hot Chocolate должны быть с одинаковым номером версии.

2. В терминале соберите проект `Northwind.GraphQLService` с помощью команды `dotnet build`.

Если в Visual Studio 2022 проект расположен вне текущего решения, обязательно выполните его сборку в терминале, прежде чем компилировать командой в меню Build (Сборка).

3. В файле `Program.cs` импортируйте пространство имен для работы с моделью EF Core:

```
using Northwind.EntityModels; // Для метода AddNorthwindContext
```

4. Добавьте код для регистрации класса контекста БД Northwind:
   - после метода `CreateBuilder` зарегистрируйте контекст БД;
   - после добавления поддержки сервера GraphQL зарегистрируйте класс `NorthwindContext` для внедрения зависимостей:

```
builder.Services.AddNorthwindContext();

builder.Services
  .AddGraphQLServer()
  .RegisterDbContext<NorthwindContext>()
  .AddQueryType<Query>();
```

5. В файле `Query.cs` добавьте определения типа объектного графа. Определите несколько запросов для получения:
   - списка категорий;
   - отдельной категории;
   - товаров для категории;

- товаров с минимальной ценой за единицу;
- всех товаров:

```
using Microsoft.EntityFrameworkCore; // Для метода Include
using Northwind.EntityModels; // Для контекста NorthwindContext

namespace Northwind.GraphQL.Service;

public class Query
{
  public string GetGreeting() => "Hello, World!";
  public string Farewell() => "Ciao! Ciao!";
  public int RollTheDie() => Random.Shared.Next(1, 7);

  public IQueryable<Category> GetCategories(NorthwindContext db) =>
    db.Categories.Include(c => c.Products);

  public Category? GetCategory(NorthwindContext db, int categoryId)
  {
    Category? category = db.Categories.Find(categoryId);
    if (category == null) return null;
    db.Entry(category).Collection(c => c.Products).Load();
    return category;
  }

  public IQueryable<Product> GetProducts(NorthwindContext db) =>
    db.Products.Include(p => p.Category);

  public IQueryable<Product> GetProductsInCategory(
    NorthwindContext db, int categoryId) =>
      db.Products.Where(p => p.CategoryId == categoryId);

  public IQueryable<Product> GetProductsByUnitPrice(
    NorthwindContext db, decimal minimumUnitPrice) =>
      db.Products.Where(p => p.UnitPrice >= minimumUnitPrice);
}
```

GraphQL-запросы к БД Northwind

Теперь можно протестировать выполнение GraphQL-запросов к категориям и товарам.

1. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
2. Запустите проект `Northwind.GraphQL.Service` в режиме `https` без отладки.
3. В окне инструмента Banana Cake Pop нажмите кнопку +, чтобы открыть новую вкладку.

4. Перейдите на вкладку Schema Definition (Определение схемы) и изучите описание запроса и типа `Category` (фрагмент показан на рис. 12.5).

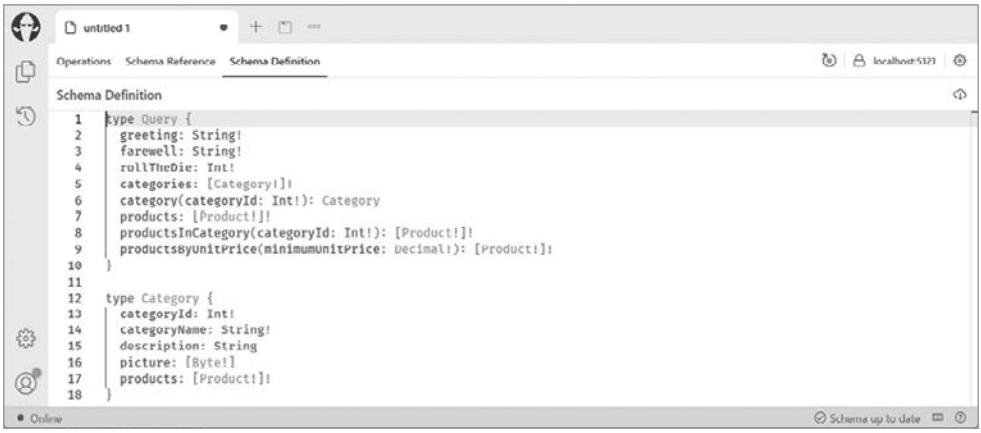


Рис. 12.5. Схема запроса к категориям и товарам в БД Northwind с помощью GraphQL

5. Ознакомьтесь с полными определениями схемы ниже:

```
type Query {
  greeting: String!
  farewell: String!
  rollTheDie: Int!
  categories: [Category!]!
  category(categoryId: Int!): Category
  products: [Product!]!
  productsInCategory(categoryId: Int!): [Product!]!
  productsByUnitPrice(minimumUnitPrice: Decimal!): [Product!]!
}

type Category {
  categoryId: Int!
  categoryName: String!
  description: String
  picture: [Byte!]
  products: [Product!]!
}

type Product {
  productId: Int!
  productName: String!
  supplierId: Int
  categoryId: Int
  quantityPerUnit: String
  unitPrice: Decimal
  unitsInStock: Short
  unitsOnOrder: Short
```

```
  reorderLevel: Short
  discontinued: Boolean!
  category: Category
  orderDetails: [OrderDetail!]!
  supplier: Supplier
}
```

6. Переключитесь на вкладку Operations (Операции) и напишите запрос для получения идентификатора, имени и описания всех категорий:

```
query AllCategories {
  categories {
    categoryId
    categoryName
    description
  }
}
```

7. Нажмите кнопку Run (Запуск) и изучите полученный ответ (рис. 12.6), см. вывод ниже (показан фрагмент):

```
{
  "data": {
    "categories": [
      {
        "categoryId": 1,
        "categoryName": "Beverages",
        "description": "Soft drinks, coffees, teas, beers, and ales"
      },
      {
        "categoryId": 2,
        "categoryName": "Condiments",
        "description": "Sweet and savory sauces, relishes, spreads,
         and seasonings"
      },
      ...
```

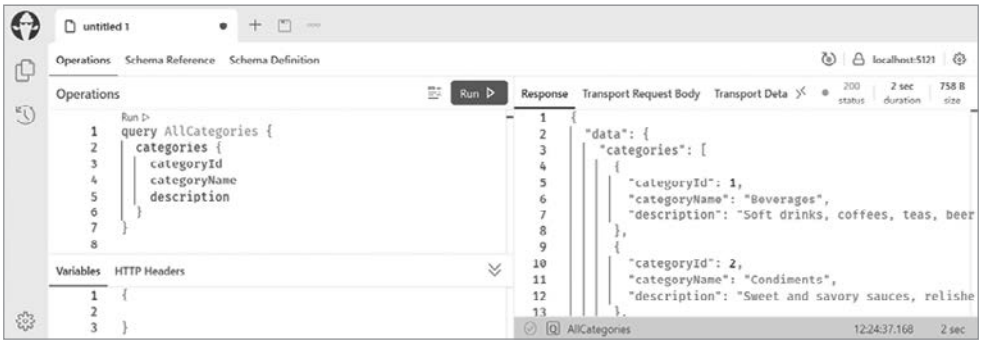


Рис. 12.6. Получение списка всех категорий

8. Нажмите кнопку +, чтобы открыть вкладку для нового документа (например, untitled 2), и напишите запрос для получения данных по категории 2, включая идентификатор, название и цену товаров:

```
query Condiments {
  category (categoryId: 2) {
    categoryId
    categoryName
    products {
      productId
      productName
      unitPrice
    }
  }
}
```

Убедитесь, что буква I в значении categoryId написана в верхнем регистре.

9. Нажмите кнопку Run (Запуск) и изучите ответ (ниже показан фрагмент):

```
{
  "data": {
    "category": {
      "categoryId": 2,
      "categoryName": "Condiments",
      "products": [
        {
          "productId": 3,
          "productName": "Aniseed Syrup",
          "unitPrice": 10
        },
        {
          "productId": 4,
          "productName": "Chef Anton's Cajun Seasoning",
          "unitPrice": 22
        },
        ...
```

10. В терминале GraphQL-сервиса обратите внимание на операторы SQL, выполненные для этого запроса:

```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (68ms) [Parameters=[@__p_0='?' (DbType = Int32)],
CommandType='Text', CommandTimeout='30']
```

```
      SELECT TOP(1) [c].[CategoryId], [c].[CategoryName], [c].[Description],
[c].[Picture]
      FROM [Categories] AS [c]
      WHERE [c].[CategoryId] = @__p_0
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (5ms) [Parameters=[@__p_0='?' (DbType = Int32)],
CommandType='Text', CommandTimeout='30']
      SELECT [p].[ProductId], [p].[CategoryId], [p].[Discontinued], [p].
[ProductName], [p].[QuantityPerUnit], [p].[ReorderLevel], [p].[SupplierId],
[p].[UnitPrice], [p].[UnitsInStock], [p].[UnitsOnOrder]
      FROM [Products] AS [p]
      WHERE [p].[CategoryId] = @__p_0
```

Хотя GraphQL-запрос запрашивал только значения ID, Name и UnitPrice, динамически сгенерированные запросы EF Core возвращали все свойства.

11. Нажмите кнопку +, чтобы добавить новую вкладку, и напишите запрос для получения идентификаторов, названий и количества единиц на складе товаров категории 1:

```
query BeverageProducts {
  productsInCategory (categoryId: 1) {
    productId
    productName
    unitsInStock
  }
}
```

12. Нажмите кнопку Run (Запуск) и изучите ответ (ниже показан фрагмент):

```
{
  "data": {
    "productsInCategory": [
      {
        "productId": 1,
        "productName": "Chai",
        "unitsInStock": 39
      },
      {
        "productId": 2,
        "productName": "Chang",
        "unitsInStock": 17
      },
      ...
```

13. Нажмите кнопку + для открытия новой вкладки и напишите запрос для получения идентификаторов, названий и количества единиц на складе товаров, а также названий их категорий:

```
query ProductsWithCategoryNames {
  products {
    productId
    productName
    category {
      categoryName
    }
    unitsInStock
  }
}
```

14. Нажмите кнопку Run (Запуск) и изучите ответ (ниже показан фрагмент):

```
{
  "data": {
    "products": [
      {
        "productId": 1,
        "productName": "Chai",
        "category": {
          "categoryName": "Beverages"
        },
        "unitsInStock": 39
      },
      {
        "productId": 2,
        "productName": "Chang",
        "category": {
          "categoryName": "Beverages"
        },
        "unitsInStock": 17
      },
      ...
```

15. Нажмите кнопку +, чтобы добавить новую вкладку, и напишите запрос для получения идентификатора и названия категории, выбора ее по идентификатору и добавления идентификаторов и названий входящих в нее товаров. Идентификатор категории задается через переменную:

```
query CategoryAndItsProducts($id: Int!){
  category(categoryId: $id) {
    categoryId
    categoryName
    products {
      productId
```

```
        productName
      }
    }
  }
```

16. В разделе Variables (Переменные) задайте значение переменной (пример показан на рис. 12.7 и в листинге ниже):

```
{
  "id": 1
}
```

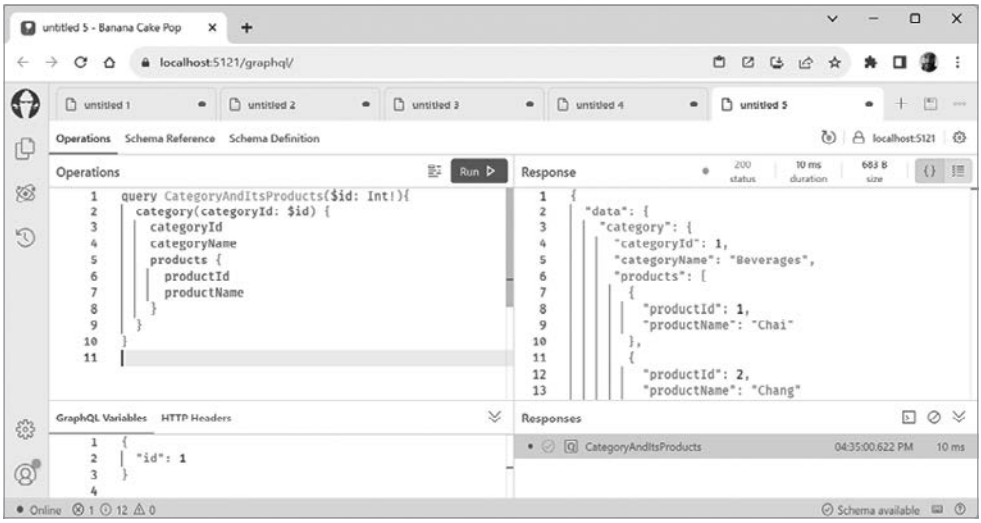


Рис. 12.7. Выполнение GraphQL-запроса с переменной

17. Нажмите кнопку Run (Запуск) и изучите ответ (ниже показан фрагмент):

```
{
  "data": {
    "category": {
      "categoryId": 1,
      "categoryName": "Beverages",
      "products": [
        {
          "productId": 1,
          "productName": "Chai"
        },
        {
          "productId": 2,
          "productName": "Chang"
        },
        ...
```

18. Нажмите кнопку +, чтобы добавить новую вкладку, и напишите запрос для получения идентификатора и названия категории, выбора ее по идентификатору и добавления идентификаторов и названий входящих в нее товаров. Идентификатор категории задается через переменную:

```
query ProductsWithMinimumPrice($unitPrice: Decimal!){
  productsByUnitPrice(minimumUnitPrice: $unitPrice) {
    productId
    productName
    unitPrice
  }
}
```

19. В разделе Variables (Переменные) задайте значение переменной:

```
{
  "unitPrice": 100
}
```

20. Нажмите кнопку Run (Запуск) и изучите ответ:

```
{
  "data": {
    "productsByUnitPrice": [
      {
        "productId": 29,
        "productName": "Thüringer Rostbratwurst",
        "unitPrice": 123.79
      },
      {
        "productId": 38,
        "productName": "Côte de Blaye",
        "unitPrice": 263.5
      }
    ]
  }
}
```

21. Закройте браузер Chrome и завершите работу веб-сервера.

Реализация пагинации

При запросе товаров через метод `GetProducts` возвращается полный список из 77 товаров. Добавим поддержку пагинации.

1. В файле `Query.cs` добавьте выражение для определения запроса, возвращающего товары с поддержкой пагинации. Реализация будет аналогична запросу без пагинации, но запрос помечен атрибутом `[UsePaging]`:

```
[UsePaging]
public IQueryable<Product> GetProductsWithPaging(NorthwindContext db) =>
  db.Products.Include(p => p.Category);
```

Дельный совет: атрибуты [UsePaging], [UseFiltering] и [UseSorting] нужно добавлять к методам запросов, возвращающим объект IQueryable<T>. Это позволит GraphQL динамически изменять LINQ-запрос перед его выполнением.

2. Запустите проект `Northwind.GraphQL.Service` в режиме `https` без отладки.
3. В окне инструмента Banana Cake Pop нажмите кнопку +, чтобы открыть новую вкладку.
4. Перейдите на вкладку Schema Definition (Определение схемы) и посмотрите, как описаны запросы `products` (без пагинации) и `productsWithPaging` (с пагинацией) страницы товаров:

```
products: [Product!]!
productsInCategory(categoryId: Int!): [Product!]!
productsByUnitPrice(minimumUnitPrice: Decimal!): [Product!]!
productsWithPaging(
  """
  Возвращает первые _n_ элементов из списка.
  """
  first: Int

  """
  Возвращает первые _n_ элементов из списка.
  """
  after: String

  """
  Возвращает последние _n_ элементов из списка.
  """
  last: Int

  """
  Возвращает элементы списка, находящиеся перед указанным курсором.
  """
  before: String
): ProductsWithPagingConnection
```

5. Откройте вкладку Operations (Операции) и напишите именованный запрос для получения первой страницы, содержащей десять товаров:

```
query FirstTenProducts {
  productsWithPaging(first: 10) {
    pageInfo {
```

```
      hasPreviousPage
      hasNextPage
      startCursor
      endCursor
    }
    nodes {
      productId
      productName
    }
  }
}
```

6. Нажмите кнопку `Run` (Запуск) и изучите ответ, в том числе раздел `pageInfo`. В нем будет указано, есть ли следующая страница, а также предоставлен курсор для навигации (пример: `MA==` — `OQ==`) (ниже показан фрагмент):

```
{
  "data": {
    "productsWithPaging": {
      "pageInfo": {
        "hasPreviousPage": false,
        "hasNextPage": true,
        "startCursor": "MA==",
        "endCursor": "OQ=="
      },
      "nodes": [
        {
          "productId": 1,
          "productName": "Chai"
        },
...
{
          "productId": 10,
          "productName": "Ikura"
        }
      ]
    }
  }
}
```

7. Нажмите кнопку `+`, чтобы открыть новую вкладку, и напишите запрос для получения второй страницы (десять товаров), задав курсор `after: "OQ=="`:

```
query SecondTenProducts {
  productsWithPaging(after: "OQ==") {
    pageInfo {
      hasPreviousPage
      hasNextPage
      startCursor
      endCursor
    }
```

```
      nodes {
        productId
        productName
      }
    }
  }
```

8. Нажмите кнопку Run (Запуск) и изучите ответ. В разделе `pageInfo` отобразится информация о наличии следующей страницы и новый курсор (в диапазоне `MTA=` — `MTk=`) (ниже показан фрагмент):

```
{
  "data": {
    "productsWithPaging": {
      "pageInfo": {
        "hasPreviousPage": true,
        "hasNextPage": true,
        "startCursor": "MTA=",
        "endCursor": "MTk="
      },
      "nodes": [
        {
          "productId": 11,
          "productName": "Queso Cabrales"
        },
...
{
          "productId": 20,
          "productName": "Sir Rodney's Marmalade"
        }
      ]
    }
  }
}
```

9. Закройте браузер Chrome и завершите работу веб-сервера.

Реализация фильтрации

Ранее в этой главе мы рассматривали предопределенные запросы с параметрами, например выборку товаров в категории по `categoryId`. Однако часто фильтры не известны заранее. Добавим в GraphQL-запросы поддержку динамической фильтрации.

1. В файле `Program.cs` добавьте вызов метода `AddFiltering()` сразу после вызова метода `AddGraphQLServer()`:

```
builder.Services
  .AddGraphQLServer()
  .AddFiltering()
  .RegisterDbContext<NorthwindContext>()
  .AddQueryType<Query>();
```

2. В файле `Query.cs` пометьте метод `GetProducts` атрибутом `[UseFiltering]`:

```
[UseFiltering]
public IQueryable<Product> GetProducts(NorthwindContext db) =>
  db.Products.Include(p => p.Category);
```

3. Запустите проект `Northwind.GraphQL.Service` в режиме `https` без отладки.
4. В окне инструмента Banana Cake Pop нажмите кнопку `+`, чтобы открыть новую вкладку.
5. Перейдите на вкладку Schema Definition (Определение схемы) и убедитесь, что запрос `products` теперь поддерживает входной фильтр:

```
products(where: ProductFilterInput): [Product!]!
```

6. Прокрутите схему вниз и найдите выражение `ProductFilterInput`. Фильтрация поддерживает логические операторы (`and`, `or`) и фильтры полей (такие как `IntOperationFilterInput`, `StringOperationFilterInput`):

```
input ProductFilterInput {
  and: [ProductFilterInput!]
  or: [ProductFilterInput!]
  productId: IntOperationFilterInput
  productName: StringOperationFilterInput
  supplierId: IntOperationFilterInput
  categoryId: IntOperationFilterInput
  quantityPerUnit: StringOperationFilterInput
  unitPrice: DecimalOperationFilterInput
  unitsInStock: ShortOperationFilterInput
  unitsOnOrder: ShortOperationFilterInput
  reorderLevel: ShortOperationFilterInput
  discontinued: BooleanOperationFilterInput
  category: CategoryFilterInput
  orderDetails: ListFilterInputTypeOfOrderDetailFilterInput
supplier: SupplierFilterInput
}
```

7. Прокрутите определение схемы еще и найдите выражения `IntOperationFilterInput` и `StringOperationFilterInput`. Они позволяют использовать операторы `eq` (равно), `neq` (не равно), `in` (в массиве), `gt` (больше), `contains` (содержит), `startsWith` (начинается с) и другие:

```
input IntOperationFilterInput {
  eq: Int
  neq: Int
  in: [Int]
  nin: [Int]
  gt: Int
  ngt: Int
  gte: Int
```

```
  ngte: Int
  lt: Int
  nlt: Int
  lte: Int
  nlte: Int
}

input StringOperationFilterInput {
  and: [StringOperationFilterInput!]
  or: [StringOperationFilterInput!]
  eq: String
  neq: String
  contains: String
  ncontains: String
  in: [String]
  nin: [String]
  startsWith: String
  nstartsWith: String
  endsWith: String
  nendsWith: String
}
```

8. Переключитесь на вкладку Operations (Операции) и напишите запрос для получения товаров с количеством единиц на складе больше 120:

```
query ProductsWithMoreThan40InStock {
  products(where: { unitsInStock: { gt: 120 } }) {
    productId
    productName
    unitsInStock
  }
}
```

9. Нажмите кнопку Run (Запуск) и изучите ответ:

```
{
  "data": {
    "products": [
      {
        "productId": 40,
        "productName": "Boston Crab Meat",
        "unitsInStock": 123
      },
      {
        "productId": 75,
        "productName": "Rhönbräu Klosterbier",
        "unitsInStock": 125
      }
    ]
  }
}
```

10. Откройте новую вкладку, перейдите на вкладку Operations (Операции) и напишите запрос для выборки товаров, названия которых начинаются с символов Cha:

```
query ProductNamesCha {
  products(where: { productName: { startsWith: "Cha" } }) {
    productId
    productName
  }
}
```

11. Нажмите кнопку Run (Запуск) и изучите ответ:

```
{
  "data": {
    "products": [
      {
        "productId": 1,
        "productName": "Chai"
      },
      {
        "productId": 2,
        "productName": "Chang"
      },
      {
        "productId": 39,
        "productName": "Chartreuse verte"
      }
    ]
  }
}
```

12. В терминале GraphQL-сервиса просмотрите SQL-фильтры, сгенерированные EF Core:

```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (2ms) [Parameters=[@__p_0_rewritten='?' (Size = 40)],
CommandType='Text', CommandTimeout='30']
      SELECT [p].[ProductId], [p].[CategoryId], [p].[Discontinued], [p].
[ProductName], [p].[QuantityPerUnit], [p].[ReorderLevel], [p].[SupplierId],
[p].[UnitPrice], [p].[UnitsInStock], [p].[UnitsOnOrder], [c].[CategoryId], [c].
[CategoryName], [c].[Description], [c].[Picture]
      FROM [Products] AS [p]
      LEFT JOIN [Categories] AS [c] ON [p].[CategoryId] = [c].
[CategoryId]
      WHERE [p].[ProductName] LIKE @__p_0_rewritten ESCAPE N'\'
```

13. Закройте браузер Chrome и завершите работу веб-сервера.

Реализация сортировки

Чтобы добавить сортировку в GraphQL-сервис, выполните следующие шаги.

1. В файле `Program.cs` вызовите метод `AddSorting()`:

```
builder.Services
  .AddGraphQLServer()
  .AddFiltering()
  .AddSorting()
  .RegisterDbContext<NorthwindContext>()
  .AddQueryType<Query>();
```

2. В файле `Query.cs` пометьте метод запроса, возвращающий объект `IQueryable<T>`, атрибутом `[UseSorting]`:

```
[UseFiltering]
[UseSorting]
public IQueryable<Product> GetProducts(NorthwindContext db) =>
  db.Products.Include(p => p.Category);
```

3. В GraphQL-запросах используйте параметры сортировки:

```
query ProductsSortedByMostExpensive {
  products(order: [ { unitPrice: DESC } ]) {
    productId
    productName
    unitPrice
  }
}
```

Перечисление `SortEnumType` поддерживает два значения:

- `ASC` — сортировка по возрастанию;
- `DESC` — сортировка по убыванию.

```
enum SortEnumType {
  ASC
  DESC
}
```

Добавление сортировки в GraphQL-сервис оставляю вам в качестве упражнения.

Создание .NET-клиентов для GraphQL-сервиса

После выполнения запросов с помощью инструмента *Banana Cake Pop* стоит рассмотреть, как клиент может взаимодействовать с GraphQL-сервисом. Хотя, безусловно, этот инструмент удобен, он работает в том же домене, что и сам сервис, поэтому некоторые ошибки могут проявляться только при создании отдельного клиента.

Форматы GraphQL-запросов

Большинство сервисов GraphQL поддерживают `GET`- и `POST`-запросы в форматах `application/graphql` и `application/json`:

- `application/graphql` — содержит только документ запроса;
- `application/json` — позволяет передавать не только сам запрос, но и несколько операций, а также переменные и их значения.

```
{
  "query": "...",
  "operationName": "...",
  "variables": { "variable1": "value1", ... }
}
```

Для передачи в запросах переменных со значениями мы будем использовать формат `application/json`.

Формат GraphQL-ответов

GraphQL-сервис возвращает JSON-документ, содержащий запрошенные данные. Если в процессе выполнения запроса возникли ошибки, они включаются в массив `errors` со следующей структурой:

```
{
  "data": { ... },
  "errors": [ ... ]
}
```

Массив ошибок должен присутствовать в документе только в случае обнаружения массивов.

Тестирование GraphQL с помощью REST-клиента

Перед работой над кодом клиента полезно проверить сервис, воспользовавшись возможностями `.http`-файлов в редакторе кода. Так вы сможете убедиться, что чаще всего ошибки клиентского .NET-приложения связаны именно с клиентом, а не с сервисом.

1. В Visual Studio Code должно быть установлено расширение REST Client Хуачао Мао (`humao.rest-client`). Если это не так, установите его.
2. В редакторе кода запустите веб-сервис проекта `Northwind.GraphQL.Service` в режиме `https` без отладки.
3. В редакторе кода в папке `HttpRequests` создайте файл `graphql-queries.http`. Напишите в нем запрос на получение товаров из категории морепродуктов:

```
### Настройка переменной для базового адреса GraphQL-сервиса
@base_address = https://localhost:5121/graphql
```

```
### Извлечение всех товаров в указанной категории
POST {{base_address}}
Content-Type: application/json
{
  "query" : "{productsInCategory(categoryId:8){productId productName
    unitsInStock}}"
}
```

4. Отправьте запрос и изучите ответ (рис. 12.8).

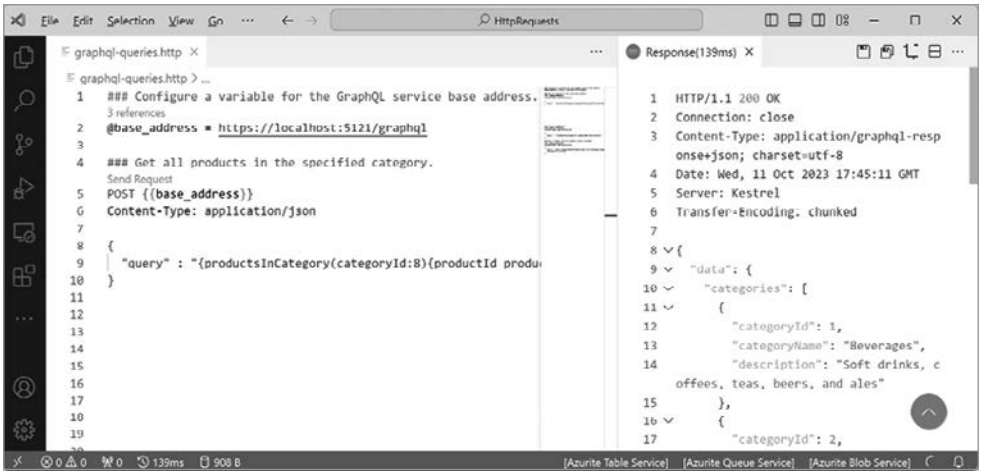


Рис. 12.8. Запрос товаров из категории морепродуктов из REST-клиента

5. Напишите запрос для получения идентификаторов, названий и описаний всех категорий:

```
### Извлечение всех категорий
POST {{base_address}}
Content-Type: application/json

{
  "query" : "{categories{categoryId categoryName description}}"
}
```

6. Отправьте запрос и убедитесь, что ответ содержит восемь категорий в разделе `data`.
7. Измените параметр `categoryId` на `id` и вновь отправьте запрос.
8. Вы должны увидеть массив ошибок:

```
Response time: 60 ms
Status code: BadRequest (400)
Alt-Svc: h3=":5121"; ma=86400
Transfer-Encoding: chunked
```

```
Date: Tue, 06 Jun 2023 16:35:18 GMT
Server: Kestrel

Content-Type: application/graphql-response+json; charset=utf-8
Content-Length: 338

-----------------------------------------------
Content:
{
  "errors": [
    {
      "message": "The field `id` does not exist on the type `Category`.",
      "locations": [
        {
          "line": 1,
          "column": 13
        }
      ],
      "path": [
        "categories"
      ],
      "extensions": {
        "type": "Category",
        "field": "id",
        "responseName": "id",
        "specifiedBy": "http://spec.graphql.org/October2021/#sec-Field-
Selections-on-Objects-Interfaces-and-Unions-Types"
      }
    }
  ]
}
```

9. Верните предыдущее имя параметра — `categoryId`.
10. Добавьте запрос на получение идентификатора, названия категории и списка товаров с идентификаторами и названиями:

```
### Извлечение категории и товаров в ее составе с помощью переменной
POST {{base_address}}
Content-Type: application/json

{
  "query": "query categoryAndItsProducts($id: Int!){category(categoryId:
➥ $id){categoryId categoryName products{productId productName}}}",
➥ "variables": {"id":1}
}
```

11. Отправьте запрос. Убедитесь, что запрос для категории 1 — `Beverages` — возвращает соответствующие товары.
12. Измените идентификатор на `4`, отправьте запрос и убедитесь, что загружаются товары категории `Dairy Products`.

После успешного тестирования запросов можно создать клиент, который будет выполнять их и обрабатывать ответы в формате JSON.

GraphQL-клиент на основе ASP.NET Core MVC

Создадим класс модели, упрощающий десериализацию ответа.

1. В редакторе кода создайте проект со следующими настройками:
   1) выберите шаблон проекта ASP.NET Core Web App (Model-View-Controller) (Веб-приложение ASP.NET Core (Модель — представление — контроллер)) / `mvc`;
   2) в поле Solution name (Имя решения) укажите значение `Chapter12`;
   3) в поле Project name (Имя проекта) укажите значение `Northwind.GraphQL.Client.Mvc`;
   4) другие параметры Visual Studio 2022:
      - в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
      - установите флажок Configure for HTTPS (Настроить для HTTPS);
      - снимите флажок Enable Docker (Включить поддержку контейнера);
      - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня).
2. В Visual Studio 2022 присвойте параметру Startup Project (Запускаемый проект) значение Current selection (Текущий вариант).
3. В файле проекта `Northwind.GraphQL.Client.Mvc` добавьте ссылку на проект моделей Northwind:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.EntityModels.
➥ SqlServer\Northwind.Common.EntityModels.SqlServer.csproj" />
</ItemGroup>
```

Путь к проекту не должен содержать символы разрыва строки.

4. Выполните сборку проекта `Northwind.GraphQL.Client.Mvc` в терминале.
5. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5123 для протокола HTTPS и 5124 — для HTTP:

```
"https": {
  "commandName": "Project",
  "dotnetRunMessages": true,
```

```
  "launchBrowser": true,
  "applicationUrl": "https://localhost:5123;http://localhost:5124",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
}
```

6. В папке `Models` проекта `Northwind.GraphQL.Client.Mvc` создайте файл класса `ResponseErrors.cs`:

```
namespace Northwind.GraphQL.Client.Mvc.Models;

public class ResponseErrors
{
  public Error[]? Errors { get; set; }
}

public class Error
{
  public string Message { get; set; } = null!;
  public Location[] Locations { get; set; } = null!;
  public string[] Path { get; set; } = null!;
}

public class Location
{
  public int Line { get; set; }
  public int Column { get; set; }
}
```

7. В той же папке создайте файл класса `ResponseProducts.cs`:

```
using Northwind.EntityModels; // Для класса Product

namespace Northwind.GraphQL.Client.Mvc.Models;

public class ResponseProducts
{
  public class DataProducts
  {
    public Product[]? ProductsInCategory { get; set; }
  }

  public DataProducts? Data { get; set; }
}
```

8. В той же папке создайте файл класса `ResponseCategories.cs`:

```
using Northwind.EntityModels; // Для класса Category

namespace Northwind.GraphQL.Client.Mvc.Models;
```

```
public class ResponseCategories
{
  public class DataCategories
  {
    public Category[]? Categories { get; set; }
  }

  public DataCategories? Data { get; set; }
}
```

9. В той же папке создайте файл класса `IndexViewModel.cs` со свойствами для хранения данных, отображаемых в представлении:

```
using Northwind.EntityModels; // Для класса Product
using System.Net; // Для перечисления HttpStatusCode

namespace Northwind.GraphQL.Client.Mvc.Models;

public class IndexViewModel
{
  public HttpStatusCode Code { get; set; }
  public string? RawResponseBody { get; set; }
  public Product[]? Products { get; set; }
  public Category[]? Categories { get; set; }
  public Error[]? Errors { get; set; }
}
```

10. В файле `Program.cs` импортируйте пространство имен для работы с HTTP-заголовками:

```
using System.Net.Http.Headers; // Для типа MediaTypeWithQualityHeaderValue
```

11. В файле `Program.cs` после вызова метода `CreateBuilder` добавьте код регистрации HTTP-клиента для GraphQL-сервиса:

```
builder.Services.AddHttpClient(name: "Northwind.GraphQL.Service",
  configureClient: options =>
  {
    options.BaseAddress = new Uri("https://localhost:5121/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
      "application/json", 1.0));
  });
```

12. В папке `Controllers` откройте файл `HomeController.cs` и подключите пространство имен для работы с кодировками текста и локальными моделями проекта:

```
using Northwind.Mvc.GraphQLClient.Models; // Для класса IndexViewModel
using System.Text; // Для класса Encoding
```

13. Добавьте поле для хранения фабрики HTTP-клиентов и инициализируйте его в конструкторе:

```
protected readonly IHttpClientFactory _clientFactory;

public HomeController(ILogger<HomeController> logger,
  IHttpClientFactory clientFactory)
{
  _logger = logger;
  _clientFactory = clientFactory;
}
```

14. Сделайте метод `Index` асинхронным и добавьте код для вызова GraphQL-сервиса. Запрос отправляется методом `POST`, носитель — `application/json`. В теле запроса содержится GraphQL-запрос, извлекающий идентификатор, название и количество единиц на складе всех товаров указанной категории. Категория передается в качестве параметра с именем `id`:

```
public async Task<IActionResult> Index(string id = "1")
{
  IndexViewModel model = new();

  try
  {
    HttpClient client = _clientFactory.CreateClient(
      name: "Northwind.GraphQL.Service");

    // Сначала выполняется обычный GET-запрос к корневому сервису

    HttpRequestMessage request = new(
      method: HttpMethod.Get, requestUri: "/");

    HttpResponseMessage response = await client.SendAsync(request);

    if (!response.IsSuccessStatusCode)
    {
      model.Code = response.StatusCode;
      model.Errors = new[] { new Error { Message =
        "Service is not successfully responding to GET requests." } };
      return View(model);
    }

    // Далее выполняется  запрос к конечной точке GraphQL

    request = new(
      method: HttpMethod.Post, requestUri: "graphql");

    request.Content = new StringContent(content: $$$"""
      {
        "query": "{productsInCategory(categoryId:{{{id}}}){productId
          productName unitsInStock}}"
```

```
      }
      """,
      encoding: Encoding.UTF8,
      mediaType: "application/json");

    response = await client.SendAsync(request);

    model.Code = response.StatusCode;
    model.RawResponseBody = await response.Content.ReadAsStringAsync();

    if (response.IsSuccessStatusCode)
    {
      model.Products = (await response.Content
        .ReadFromJsonAsync<ResponseProducts>())?.Data?.ProductsInCategory;
    }
    else
    {
      model.Errors = (await response.Content
        .ReadFromJsonAsync<ResponseErrors>())?.Errors;
    }
  }
  catch (Exception ex)
  {
    _logger.LogWarning(
      $"Northwind.GraphQL.Service exception: {ex.Message}");

    model.Errors = new[] { new Error { Message = ex.Message } };
  }

  return View(model);
}
```

Дельный совет: для задания содержимого запроса в C# 11 и более поздней версии можно воспользоваться интерполированным строковым литералом ($$$"""), который позволяет вставлять переменную id с помощью тройных фигурных скобок. Не перепутайте с двойными фигурными скобками после объекта unitsInStock, которыми заканчивается сам запрос.

15. В файле `Views/Home/Index.cshtml` замените разметку отображением списка морепродуктов:

```
@using Northwind.EntityModels
@using Northwind.GraphQL.Client.Mvc.Models @* for VS Code only *@
@model IndexViewModel
@{
  ViewData["Title"] = "Products from GraphQL service";
}
<div class="text-center">
  <h1 class="display-4">@ViewData["Title"]</h1>
```

```
<div class="card card-body">
  <form>
    Enter a category id
      <input name="id" value="1" />
      <input type="submit" />
  </form>
</div>
@if (Model.Errors is not null)
{
  <div class="alert alert-danger" role="alert">
    <table class="table table-striped">
      <thead>
      <tr>
        <td>Message</td>
        <td>Path</td>
        <td>Locations</td>
      </tr>
      </thead>
      <tbody>
        @foreach (Error error in Model.Errors)
        {
          <tr>
            <td>@error.Message</td>
            <td>
              @if (error.Path is not null)
              {
                @foreach (string path in error.Path)
                {
                  <span class="badge bg-danger">@path</span>
                }
              }
            </td>
            <td>
              @if (error.Locations is not null)
              {
                @foreach (Location location in error.Locations)
                {
                  <span class="badge bg-danger">
                    @location.Line, @location.Column
                  </span>
                }
              }
            </td>
        </tr>
        }
      </tbody>
    </table>
  </div>
}
```

```
  @if (Model.Categories is not null)
  {
    <div>
      <p class="alert alert-success" role="alert">
        There are @Model.Categories.Count() products.</p>
      <p>
        @foreach (Category category in Model.Categories)
        {
          <span class="badge bg-dark">
            @category.CategoryId
            @category.CategoryName
          </span>
        }
      </p>
    </div>
  }
  @if (Model.Products is not null)
  {
    <div>
      <p class="alert alert-success" role="alert">
        There are @Model.Products.Count() products.</p>
      <p>
        @foreach (Product p in Model.Products)
        {
          <span class="badge bg-dark">
            @p.ProductId
            @p.ProductName
            -
            @(p.UnitsInStock is null ? "0" : p.UnitsInStock.Value) in stock
          </span>
        }
      </p>
    </div>
  }
  <p>
    <a class="btn btn-primary" data-bs-toggle="collapse"
       href="#collapseExample" role="button"
       aria-expanded="false" aria-controls="collapseExample">
      Show/Hide Details
    </a>
  </p>
  <div class="collapse" id="collapseExample">
    <div class="card card-body">
      Status code @((int)Model.Code): @Model.Code
      <hr />
      @Model.RawResponseBody
    </div>
  </div>
</div>
```

Тестирование .NET-клиента

Протестируем .NET-клиент.

1. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
2. Запустите проект `Northwind.GraphQL.Service` в режиме `https` без отладки.
3. Запустите проект `Northwind.GraphQL.Client.Mvc` в режиме `https` без отладки.
4. Убедитесь, что товары загружаются с помощью GraphQL-запроса корректно (рис. 12.9).

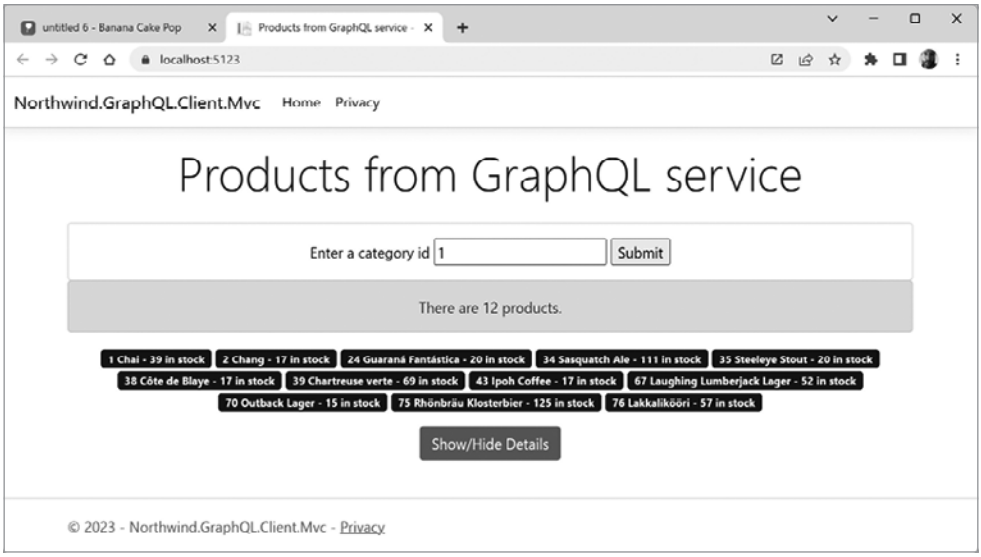


Рис. 12.9. Товары в категории Beverages из GraphQL-сервиса

5. Введите идентификатор другой существующей категории, например 4.
6. Введите идентификатор несуществующей категории, например 13, — должно вернуться 0 товаров.
7. Закройте браузер Chrome и завершите работу веб-сервера с проектом `Northwind.GraphQL.Client.Mvc`.
8. В файле `HomeController.cs` внесите в запрос ошибку (например, измените название параметра `productId` на `productid`).
9. Запустите проект `Northwind.GraphQL.Client.Mvc` в режиме `https` без отладки.
10. Нажмите кнопку Show/Hide Details (Показать/скрыть подробности) и обратите внимание на сообщение об ошибке и подробное описание ответа:

```
{"errors":[{"message":"The field \u0060productid\u0060 does not exist on the
type \u0060Product\u0060.",
 "locations":[{"line":1,"column":35}],
 "path":["productsInCategory"],
 "extensions":{"type":"Product","field":"productid",
 "responseName":"productid",
 "specifiedBy":"http://spec.graphql.org/October2021/
#sec-Field-Selections-on-Objects-Interfaces-and-Unions-Types"}}]}
```

11. Закройте браузер Chrome и остановите оба веб-сервера.
12. Исправьте ошибку в запросе!

Создание консольного клиента с помощью Strawberry Shake

Вместо стандартных HTTP-клиентов можно воспользоваться библиотекой ChilliCream Strawberry Shake, которая упрощает создание .NET-клиентов для GraphQL-сервисов.

В копилку знаний: подробнее о библиотеке Strawberry Shake можно прочитать на сайте по адресу https://chillicream.com/docs/strawberryshake.

Создадим с помощью Strawberry Shake еще один клиент и оценим преимущества этой библиотеки.

1. В редакторе кода создайте консольное приложение `Northwind.GraphQL.Client.Console`.
2. В терминале откройте папку проекта и создайте манифест инструментов:

```
dotnet new tool-manifest
```

3. В терминале установите инструменты Strawberry Shake:

```
dotnet tool install StrawberryShake.Tools --local
```

4. Убедитесь, что установка выполнена успешно:

```
You can invoke the tool from this directory using the following commands:
'dotnet tool run dotnet-graphql' or 'dotnet dotnet-graphql'.
Tool 'strawberryshake.tools' (version '13.5.1') was successfully installed.
Entry is added to the manifest file C:\apps-services-net8\Chapter12\Northwind.
GraphQL.Client.Console\.config\dotnet-tools.json.
```

5. В файле созданного проекта добавьте строгую обработку предупреждений, а также ссылки на NuGet-расширения Microsoft для внедрения зависимостей, работы с HTTP и генерации кода Strawberry Shake. Кроме того, статически импортируйте класс `System.Console` на глобальном уровне:

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

<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net8.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
  <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
</PropertyGroup>

<ItemGroup>
  <PackageReference Version="8.0.0"
    Include="Microsoft.Extensions.DependencyInjection" />
  <PackageReference Version="8.0.0"
    Include="Microsoft.Extensions.Http" />
  <PackageReference Version="13.5.1"
    Include="StrawberryShake.Server" />
</ItemGroup>

<ItemGroup>
  <Using Include="System.Console" Static="true" />
</ItemGroup>

</Project>
```

Для разных типов .NET-проектов требуются разные пакеты Strawberry Shake:

- StrawberryShake.Server — для консольных приложений и ASP.NET Core;
- StrawberryShake.Blazor — для Blazor WebAssembly;
- StrawberryShake.Maui — для .NET MAUI.

6. Соберите проект `Northwind.GraphQL.Client.Console` для восстановления зависимостей.
7. Запустите проект `Northwind.GraphQL.Service` в режиме `https` без отладки. Оставьте его запущенным для взаимодействия с инструментом Strawberry Shake.
8. Откройте проект `Northwind.GraphQL.Client.Console` в терминале и добавьте клиент для GraphQL-сервиса:

```
dotnet graphql init https://localhost:5121/graphql/ -n NorthwindClient
```

9. Обратите внимание на результаты:

```
Download schema started.
Download schema completed in 189 ms
Client configuration started.
Client configuration completed in 83 ms
```

10. В файле `.graphqlrc.json` проекта `Northwind.GraphQL.Client.Console` добавьте пространство имен C#, используемое при генерации кода:

```
{
  "schema": "schema.graphql",
  "documents": "**/*.graphql",
  "extensions": {
    "strawberryShake": {
      "name": "NorthwindClient",
      "namespace": "Northwind.GraphQL.Client.Console",
      "url": "https://localhost:5111/graphql/",
      "records": {
        "inputs": false,
        "entities": false
      },
      "transportProfiles": [
        {
          "default": "Http",
          "subscription": "WebSocket"
        }
      ]
    }
  }
}
```

11. В папке проекта `Northwind.GraphQL.Client.Console` создайте файл `seafoodProducts.graphql` с запросом на получение морепродуктов:

```
query SeafoodProducts {
  productsInCategory(categoryId:8) {
    productId
    productName
    unitsInStock
  }
}
```

GraphQL-запросам в Strawberry Shake нужно присваивать имена.

12. Программа Visual Studio 2022 должна исключить файл проекта из сборки, так как не распознала его. Если этого не произошло, вручную удалите или закомментируйте соответствующий код:

```
<!-- Закомментировав этот код, вы исключите файл из процесса сборки
<ItemGroup>
  <GraphQL Remove="seafoodProducts.graphql" />
</ItemGroup>
-->
```

Дельный совет: чтобы код Strawberry Shake генерировался автоматически, необходим хотя бы один .graphql-файл. Указанный выше элемент не позволит инструменту Strawberry Shake завершить генерацию кода, что в итоге вызовет ошибки компиляции. Вам следует обязательно удалить или закомментировать этот элемент.

13. Выполните сборку проекта `Northwind.GraphQL.Client.Console`, чтобы инструмент Strawberry Shake обработал GraphQL-запрос и сгенерировал прокси-классы.
14. Убедитесь, что в созданной папке `obj\Debug\net8.0\berry` появился файл `NorthwindClient.Client.cs` со множеством определенных типов, в том числе интерфейс `INorthwindClient` (рис. 12.10).

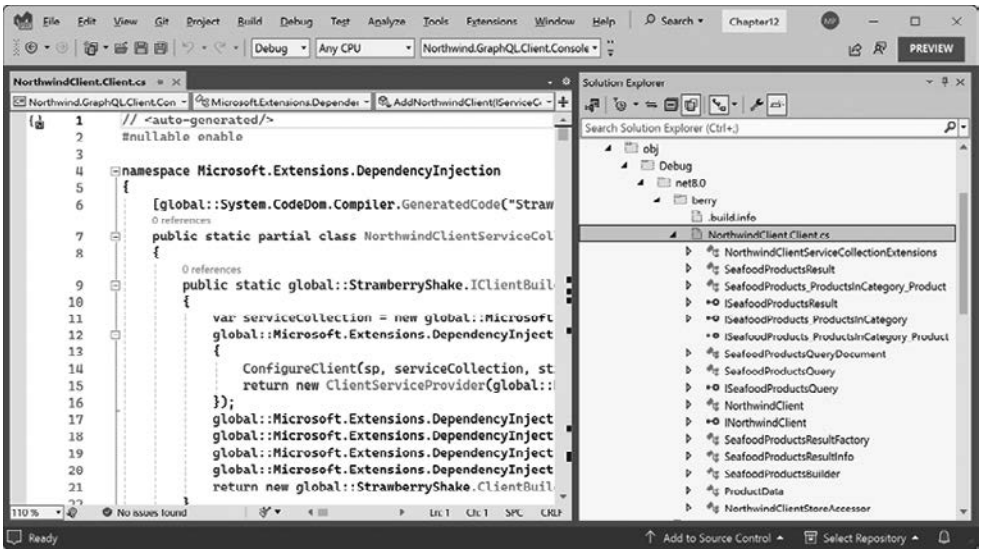


Рис. 12.10. Созданный файл класса для GraphQL-сервиса Northwind

15. В файле `Program.cs` удалите содержимое. Добавьте код для создания новой коллекции сервисов, включите в нее автоматически сгенерированный класс `NorthwindClient`, указав корректный URL-адрес GraphQL-сервиса. После этого получите экземпляр `NorthwindClient` через систему внедрения зависимостей и выполните запрос для извлечения товаров из категории морепродуктов:

```
using Microsoft.Extensions.DependencyInjection; // Для класса ServiceCollection
using Northwind.GraphQL.Client.Console; // Для интерфейса INorthwindClient
using StrawberryShake; // Для метода расширения EnsureNoErrors

ServiceCollection serviceCollection = new();

serviceCollection
  .AddNorthwindClient() // Метод расширения Strawberry Shake
  .ConfigureHttpClient(client =>
    client.BaseAddress = new Uri("https://localhost:5121/graphql"));

IServiceProvider services = serviceCollection.BuildServiceProvider();

INorthwindClient client = services.GetRequiredService<INorthwindClient>();

var result = await client.SeafoodProducts.ExecuteAsync();
result.EnsureNoErrors();

if (result.Data is null)
{
  WriteLine("No data!");
  return;
}

foreach (var product in result.Data.ProductsInCategory)
{
  WriteLine("{0}: {1}",
    product.ProductId, product.ProductName);
}
```

16. Запустите консольное приложение и изучите результаты:

```
10: Ikura
13: Konbu
18: Carnarvon Tigers
30: Nord-Ost Matjeshering
36: Inlagd Sill
37: Gravad lax
40: Boston Crab Meat
41: Jack's New England Clam Chowder
45: Rogede sild
46: Spegesild
58: Escargots de Bourgogne
73: Röd Kaviar
```

Реализация мутаций GraphQL

Большинству сервисов необходимо не только запрашивать данные, но и изменять их. В GraphQL для этого используются мутации.

Мутация состоит из трех взаимосвязанных компонентов.

- *Определение мутации* — описывает изменение, которое вносится в граф. Название должно включать глагол и существительное в стиле camelCase, например `addProduct`.
- *Входные данные* — объект с параметрами для мутации. Он должен называться так же, как мутация, но с суффиксом `Input`, например `AddProductInput`. Вход всегда один, но, поскольку это объектный граф, структура может быть сложной.
- *Полезная нагрузка* — объект, который возвращает мутация. Название совпадает с мутацией, но с суффиксом `Payload`, например `AddProductPayload`. Этот объект тоже всегда один и может содержать любую необходимую информацию.

Добавление мутаций в GraphQL-сервис

Рассмотрим процесс добавления мутаций для создания товаров. Позже мы добавим мутации для их обновления и удаления.

1. В проекте/папке `Northwind.GraphQL.Service` создайте файл класса `Mutation.cs`.
2. В файле класса определите запись и два класса для представления трех типов, необходимых для выполнения мутации `addProduct`:

```
using Northwind.EntityModels; // Для класса Product

namespace Northwind.GraphQL.Service;

// Входные данные доступны для чтения,
// поэтому мы будем использовать тип записи
public record AddProductInput(
  string ProductName,
  int? SupplierId,
  int? CategoryId,
  string QuantityPerUnit,
  decimal? UnitPrice,
  short? UnitsInStock,
  short? UnitsOnOrder,
  short? ReorderLevel,
  bool Discontinued);

public class AddProductPayload
{
```

```
  public AddProductPayload(Product product)
  {
    Product = product;
  }

  public Product Product { get; }
}

public class Mutation
{
  public async Task<AddProductPayload> AddProductAsync(
    AddProductInput input, NorthwindContext db)
  {
    // Можно использовать инструмент типа AutoMapper,
    // но мы будем сопоставлять объекты вручную

    Product product = new()
    {
      ProductName = input.ProductName,
      SupplierId = input.SupplierId,
      CategoryId = input.CategoryId,
      QuantityPerUnit = input.QuantityPerUnit,
      UnitPrice = input.UnitPrice,
      UnitsInStock = input.UnitsInStock,
      UnitsOnOrder = input.UnitsOnOrder,
      ReorderLevel = input.ReorderLevel,
      Discontinued = input.Discontinued
    };

    db.Products.Add(product);

    int affectedRows = await db.SaveChangesAsync();

    // Можно использовать свойство класса affectedRows для возврата ошибки
    // или некое другое действие, если значение результата равно 0
    return new AddProductPayload(product);
  }
}
```

3. В файле `Program.cs` добавьте вызов метода `AddMutationType<T>` для регистрации класса `Mutation`:

```
builder.Services
  .AddGraphQLServer()
  .AddFiltering()
  .AddSorting()
  .RegisterDbContext<NorthwindContext>()
  .AddQueryType<Query>()
  .AddMutationType<Mutation>();
```

Исследование мутации для добавления товара

Теперь можно протестировать мутацию в инструменте Banana Cake Pop.

1. Запустите проект `Northwind.GraphQL.Service` в режиме `https` без отладки.
2. В окне инструмента Banana Cake Pop нажмите кнопку +, чтобы открыть новую вкладку.
3. Перейдите на вкладку Schema Definition (Определение схемы) и изучите описание мутации (на рис. 12.11 показан фрагмент).

Рис. 12.11. Схема мутации добавления товара в GraphQL

4. Изучите полные определения схемы для мутации `addProduct` и связанных с ней типов:

```
type Mutation {
  addProduct(input: AddProductInput!): AddProductPayload!
}

type Product {
  productId: Int!
  productName: String!
  supplierId: Int
  categoryId: Int
  quantityPerUnit: String
  unitPrice: Decimal
  unitsInStock: Short
  unitsOnOrder: Short
  reorderLevel: Short
  discontinued: Boolean!
  category: Category
  supplier: Supplier
  orderDetails: [OrderDetail!]!
}
```

```
...
type AddProductPayload {
  product: Product!
}

input AddProductInput {
  productName: String!
  supplierId: Int
  categoryId: Int
  quantityPerUnit: String!
  unitPrice: Decimal
  unitsInStock: Short
  unitsOnOrder: Short
  reorderLevel: Short
  discontinued: Boolean!
}
```

5. Перейдите на вкладку Operations (Операции). Если нужно, создайте новый документ и введите мутацию для добавления товара Tasty Burgers. В ответе выберите идентификатор и название товара:

```
mutation AddProduct {
  addProduct(
    input: {
      productName: "Tasty Burgers"
      supplierId: 1
      categoryId: 2
      quantityPerUnit: "6 per box"
      unitPrice: 40
      unitsInStock: 0
      unitsOnOrder: 0
      reorderLevel: 0
      discontinued: false
    }
  )
  {
    product {
      productId
      productName
    }
  }
}
```

6. Нажмите кнопку Run (Запуск) и убедитесь, что товар добавлен. Обратите внимание на присвоенный ему порядковый номер в SQL Server. Это число будет больше 77, если ранее уже добавлялись товары (см. рис. 12.12 и вывод ниже):

```
{
  "data": {
    "addProduct": {
```

```
      "product": {
        "productId": 79,
        "productName": "Tasty Burgers",
      }
    }
  }
}
```

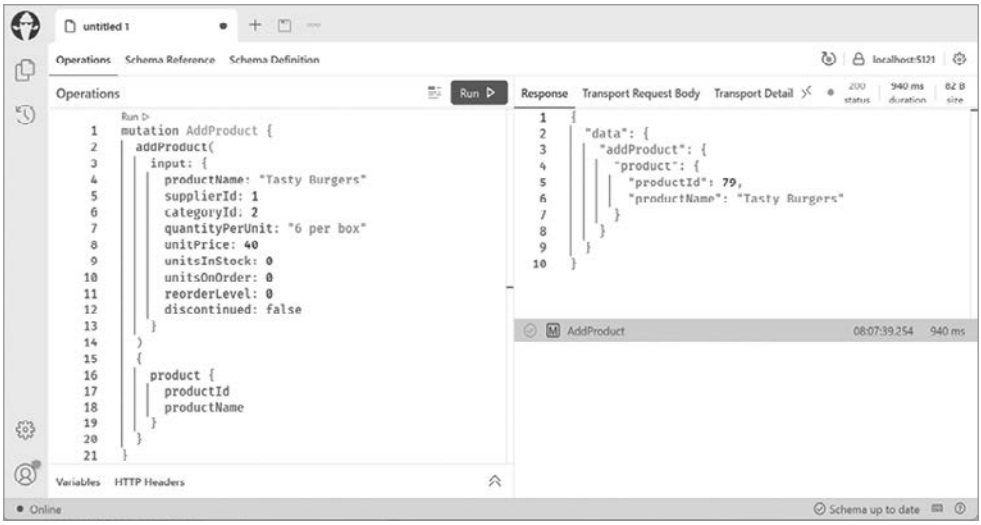


Рис. 12.12. Добавление товара с помощью мутации GraphQL

Внимание! Запомните идентификатор добавленного товара. В следующем разделе вы обновите и удалите его.

Товары с идентификаторами от 1 до 77 удалить нельзя — они связаны с другими таблицами и попытка приведет к нарушению ссылочной целостности!

7. Закройте браузер и завершите работу веб-сервера.

Реализация мутаций для обновления и удаления

Теперь добавим мутации для:

- обновления цены товара;
- обновления всех полей `units` товара;
- удаления товара.

1. В файле `Mutation.cs` создайте три типа записей для входных данных мутаций `updateProduct` и `deleteProduct`:

```
public record UpdateProductPriceInput(
  int? ProductId,
  decimal? UnitPrice);

public record UpdateProductUnitsInput(
  int? ProductId,
  short? UnitsInStock,
  short? UnitsOnOrder,
  short? ReorderLevel);

public record DeleteProductInput(
  int? ProductId);
```

2. В том же файле определите два класса для представления данных, которые возвращает мутация. В них должна присутствовать информация о результатах выполнения операций обновления/удаления, в том числе успешность действия:

```
public class UpdateProductPayload
{
  public UpdateProductPayload(Product? product, bool updated)
  {
    Product = product;
    Success = updated;
  }

  public Product? Product { get; }
  public bool Success { get; }
}

public class DeleteProductPayload
{
  public DeleteProductPayload(bool deleted)
  {
    Success = deleted;
  }

  public bool Success { get; }
}
```

3. В том же файле в классе `Mutation` добавьте три метода для реализации мутаций — двух для обновления (`updateProduct`) и одной для удаления (`deleteProduct`):

```
public async Task<UpdateProductPayload> UpdateProductPriceAsync(
  UpdateProductPriceInput input, NorthwindContext db)
{
  Product? product = await db.Products.FindAsync(input.ProductId);

  int affectedRows = 0;

  if (product is not null)
```

```
    {
      product.UnitPrice = input.UnitPrice;

      affectedRows = await db.SaveChangesAsync();
    }

    return new UpdateProductPayload(product,
      updated: affectedRows == 1);
  }

  public async Task<UpdateProductPayload> UpdateProductUnitsAsync(
    UpdateProductUnitsInput input, NorthwindContext db)
  {
    Product? product = await db.Products.FindAsync(input.ProductId);

    int affectedRows = 0;

    if (product is not null)
    {
      product.UnitsInStock = input.UnitsInStock;
      product.UnitsOnOrder = input.UnitsOnOrder;
      product.ReorderLevel = input.ReorderLevel;

      affectedRows = await db.SaveChangesAsync();
    }

    return new UpdateProductPayload(product,
      updated: affectedRows == 1);
  }

  public async Task<DeleteProductPayload> DeleteProductAsync(
    DeleteProductInput input, NorthwindContext db)
  {
    Product? product = await db.Products.FindAsync(input.ProductId);

    int affectedRows = 0;

    if (product is not null)
    {
      db.Products.Remove(product);

      affectedRows = await db.SaveChangesAsync();
    }

    return new DeleteProductPayload(
      deleted: affectedRows == 1);
  }
```

4. Если сервер БД не работает (например, если размещен в контейнере, на виртуальной машине или в облаке), запустите его.

5. Запустите проект `Northwind.GraphQL.Service` в режиме `https` без отладки.
6. В окне инструмента Banana Cake Pop нажмите кнопку `+`, чтобы открыть новую вкладку.
7. Напишите запрос для получения товаров, которые вы добавили, например со значением параметра `productid` больше 77:

```
query NewProducts {
  products(where: { productId: { gt: 77 } }) {
    productId
    productName
    unitPrice
    unitsInStock
    unitsOnOrder
    reorderLevel
  }
}
```

8. Нажмите кнопку `Run` (Запуск). В ответе появится ранее добавленный товар с ценой за единицу, равной `40`:

```
{
  "data": {
    "products": [
      {
        "productId": 79,
        "productName": "Tasty Burgers",
        "unitPrice": 40,
        "unitsInStock": 0,
        "unitsOnOrder": 0,
        "reorderLevel": 0
      }
    ]
  }
}
```

9. Запишите значение параметра `ProductId` добавленного товара. В моем случае это `79`.
10. В окне инструмента Banana Cake Pop нажмите кнопку `+`, чтобы открыть новую вкладку.
11. Добавьте мутацию для изменения цены товара. В ответе из возвращенного объекта `product` извлеките идентификатор, название, цену единицы и количество товара на складе:

```
mutation UpdateProductPrice {
  updateProductPrice(
    input: {
```

```
      productId: 79
      unitPrice: 75
    }
  )
  {
    product {
      productId
      productName
      unitPrice
      unitsInStock
    }
  }
}
```

12. Нажмите кнопку Run (Запуск) и изучите ответ:

```
{
  "data": {
    "updateProductPrice": {
      "product": {
        "productId": 79,
        "productName": "Tasty Burgers",
        "unitPrice": 75,
        "unitsInStock": 0
      }
    }
  }
}
```

13. Нажмите кнопку +, чтобы открыть новую вкладку. Добавьте мутацию для изменения количества товара, а затем из возвращенного объекта `product` извлеките те же данные, что и в шаге 11:

```
mutation UpdateProductUnits {
  updateProductUnits(
    input: {
      productId: 79
      unitsInStock: 20
      unitsOnOrder: 0
      reorderLevel: 10
    }
  )
  {
    success
  }
}
```

14. Нажмите кнопку Run (Запуск) и изучите ответ:

```
{
  "data": {
    "updateProductUnits": {
```

```
      "success": true
    }
  }
}
```

15. Переключитесь на вкладку с запросом товаров, снова нажмите кнопку Run (Запуск) и проанализируйте вывод с целью убедиться в обновлениях:

```
{
  "data": {
    "products": [
      {
        "productId": 79,
        "productName": "Tasty Burgers",
        "unitPrice": 75,
        "unitsInStock": 20,
        "unitsOnOrder": 0,
        "reorderLevel": 10
      }
    ]
  }
}
```

16. Откройте новую вкладку, добавьте мутацию для удаления товара и убедитесь, что операция прошла успешно:

```
mutation DeleteProduct {
  deleteProduct(
    input: {
      productId: 79
    }
  )
  {
    success
  }
}
```

Внимание! Нельзя удалить товары, если на них есть ссылки в других таблицах. Попытка удалить товары с идентификаторами 1–77 вызовет ошибку.

17. Нажмите кнопку Run (Запуск), чтобы выполнить удаление:

```
{
  "data": {
    "deleteProduct": {
      "success": true
    }
  }
}
```

18. Выполните запрос на получение товаров — массив должен быть пустым:

```
{
  "data": {
    "products": []
  }
}
```

19. Закройте браузер и завершите работу веб-сервера.

Реализация подписок в GraphQL

В GraphQL подписки по умолчанию работают через протокол WebSockets, но их можно настроить на использование *событий, отправляемых сервером (server-sent events, SSE)*, брокером SignalR или системой gRPC.

Представьте, что клиентское приложение хочет получать уведомления о снижении цены товара. Вместо периодического запрашивания обновленных данных клиент может подписаться на событие, которое срабатывает каждый раз при снижении цены.

Добавление подписки и темы в GraphQL-сервис

Добавим описанную возможность в GraphQL-сервис с помощью подписок.

1. Создайте файл класса `ProductDiscount.cs`.
2. Определите в нем модель для уведомления клиента о снижении цены товара:

```
namespace Northwind.GraphQL.Service;

public class ProductDiscount
{
  public int? ProductId { get; set; }
  public decimal? OriginalUnitPrice { get; set; }
  public decimal? NewUnitPrice { get; set; }
}
```

3. Создайте файл класса `Subscription.cs`.
4. Определите в нем подписку (тему) с именем `OnProductDiscounted`, на которую может подписаться клиент:

```
namespace Northwind.GraphQL.Service;

public class Subscription
{
  [Subscribe]
  [Topic]
  public ProductDiscount OnProductDiscounted(
```

```
    [EventMessage] ProductDiscount productDiscount)
      => productDiscount;
}
```

5. Открыв файлу `Mutation.cs`, добавьте в методе `UpdateProductPriceAsync` код для отправки уведомления в тему при снижении цены:

```
public async Task<UpdateProductPayload> UpdateProductPriceAsync(
  UpdateProductPriceInput input, NorthwindContext db,
  ITopicEventSender eventSender)
{
  Product? product = await db.Products.FindAsync(input.ProductId);

  int affectedRows = 0;

  if (product is not null)
  {
    if (input.UnitPrice < product.UnitPrice)
    {
      // Если цена товара была снижена,
      // отправляется сообщение подписчикам
      ProductDiscount productDiscount = new()
      {
        ProductId = input.ProductId,
        OriginalUnitPrice = product.UnitPrice,
        NewUnitPrice = input.UnitPrice
      };

      await eventSender.SendAsync(topicName:
        nameof(Subscription.OnProductDiscounted),
        message: productDiscount);
    }

    product.UnitPrice = input.UnitPrice;

    affectedRows = await db.SaveChangesAsync();
  }

  return new UpdateProductPayload(product,
    updated: affectedRows == 1);
}
```

6. В файле `Program.cs` настройте GraphQL-сервис: зарегистрируйте класс `Subscription` и настройте хранение активных подписок в памяти:

```
builder.Services
  .AddGraphQLServer()
  .AddFiltering()
  .AddSorting()
  .AddSubscriptionType<Subscription>()
  .AddInMemorySubscriptions()
  .RegisterDbContext<NorthwindContext>()
  .AddQueryType<Query>()
  .AddMutationType<Mutation>();
```

Для хранения активных подписок можно использовать не только память, но и Redis или другие хранилища данных.

7. Дополнительно настройте поддержку протокола WebSockets:

```
app.UseWebSockets(); // Для подписок
```

Это не обязательно, так как при необходимости GraphQL-сервис может переключиться на SSE через HTTPS.

Тестирование подписки на тему

Подпишемся на тему и проверим, как она работает.

1. Запустите проект `Northwind.GraphQL.Service` в режиме `https` без отладки.
2. В окне инструмента Banana Cake Pop нажмите кнопку +, чтобы открыть новую вкладку.
3. Перейдите на вкладку Schema Reference (Описание схемы), выберите тип `Subscription` и найдите тему `onProductDiscounted` (рис. 12.13).

Рис. 12.13. Подписка на тему

4. Перейдите на вкладку Operations (Операции), добавьте подписку на тему и выберите все поля для отображения:

```
subscription {
  onProductDiscounted {
    productId
```

```
    originalUnitPrice
    newUnitPrice
  }
}
```

5. Нажмите кнопку Run (Запуск). Подписка активируется, но пока не показывает результаты.
6. Нажмите кнопку +, чтобы открыть новую вкладку, и создайте мутацию, чтобы изменить цену товара с идентификатором `1` на значение `8`. В возвращенном объекте `product` выберите идентификатор и цену товара, а также успешность обновления:

```
mutation UpdateProductPrice {
  updateProductPrice(
    input: {
      productId: 1
      unitPrice: 8.99
    }
  )
  {
    product {
      productId
      unitPrice
    }
    success
  }
}
```

7. Нажмите кнопку Run (Запуск) и изучите ответ:

```
{
  "data": {
    "updateProductPrice": {
      "product": {
        "productId": 1,
        "unitPrice": 8.99
      },
      "success": true
    }
  }
}
```

8. Вернитесь на вкладку с подпиской — на ней должно отобразиться уведомление о снижении цены. Обратите внимание: подписка остается активной, что видно по индикаторам загрузки на вкладке и кнопке Cancel (Отмена) (рис. 12.14).
9. Переключитесь обратно на вкладку с мутацией обновления, измените цену товара на `7.99` и нажмите кнопку Run (Запуск). Затем вновь вернитесь на вкладку с подпиской. Вы увидите уведомление об обновлении.

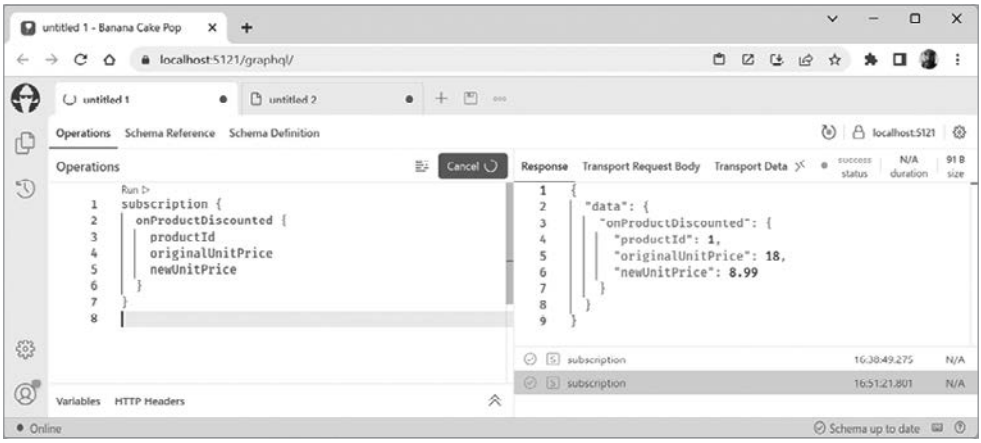


Рис. 12.14. На активную подписку указывает индикатор загрузки

10. На вкладке с мутацией обновления измените цену товара на `9.99` и нажмите кнопку Run (Запуск). Обратите внимание, что на вкладке с подпиской нет уведомлений, так как цена возросла, а не снизилась.
11. Закройте браузер и завершите работу веб-сервера.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 12.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. Какой транспортный протокол используется в GraphQL?
2. Какой формат передачи данных используется в GraphQL-запросах?
3. Как передавать параметры в GraphQL-запросах?
4. Каковы преимущества Strawberry Shake перед обычным HTTP-клиентом в рамках отправки GraphQL-запросов?
5. Как добавить новый товар в БД Northwind?

Упражнение 12.2. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-12---combining-data-sources-using-graphql.

Упражнение 12.3. Создание .NET-клиентов

В файле `HomeController.cs` добавьте метод `Categories`, который будет запрашивать поле `categories` с параметром `id`. Позвольте пользователю вводить `id` на странице и просматривать сведения о категории и товарах в ней.

Резюме

В этой главе мы рассмотрели:

- основные концепции GraphQL;
- как создать класс `Query` с полями для запроса сущностей;
- как использовать инструмент Banana Cake Pop для изучения схемы GraphQL-сервиса;
- как отправлять `POST`-запросы в GraphQL с помощью расширения REST Client;
- как разработать .NET-клиент для GraphQL;
- как реализовать мутации GraphQL;
- как управлять подписками в GraphQL.

В следующей главе речь пойдет о gRPC — технологии, которая помогает создавать эффективные микросервисы.

13 Создание эффективных микросервисов с помощью gRPC

В этой главе вы познакомитесь с технологией gRPC — мощным инструментом для создания сервисов, способных эффективно взаимодействовать на разных платформах.

Однако браузеры не предоставляют полного доступа ко всем возможностям HTTP/2, которые необходимы для работы gRPC. Поэтому gRPC особенно эффективен для реализации многоуровневых промежуточных сервисов и микросервисов, где требуется частое взаимодействие между компонентами системы. Эффективность такой коммуникации критически важна для масштабируемости и высокой производительности микросервисов.

Модульный монолитный сервис, построенный по двухуровневой архитектуре «клиент — сервис», работает быстрее, поскольку его модули взаимодействуют друг с другом напрямую, без сетевых задержек. Взаимодействие между клиентами и сервисом проходит через единственный уровень сетевого соединения.

В микросервисной архитектуре уровней больше, а значит, возрастает количество сетевых взаимодействий. Поэтому важно обеспечить максимально быструю и эффективную передачу данных между микросервисами. gRPC разработан специально для этой задачи, обеспечивая сверхбыстрое сетевое взаимодействие (рис. 13.1).

Дельный совет: в последние годы было распространено мнение, что микросервисы — лучший выбор для всех сценариев и новые системы нужно сразу проектировать как микросервисные. Однако в последнее время наблюдается отход от этой идеи. Все больше специалистов рекомендуют начинать разработку с модульного монолита, а при необходимости разделять его на микросервисы.

Микросервисы неизбежно работают медленнее из-за дополнительных сетевых взаимодействий. Кроме того, они требуют сложной координации, непростого развертывания и оркестрации. Прежде чем переходить к микросервисной архитектуре, важно оценить, оправдает ли она себя в конкретном проекте.

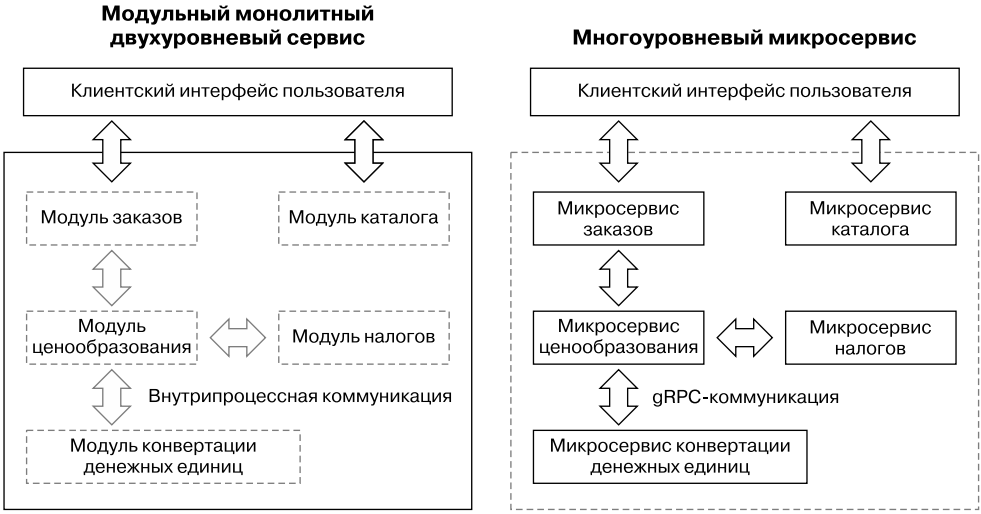


Рис. 13.1. Сравнение двухуровневого модульного монолитного сервиса с многоуровневым микросервисом

Обзор фреймворка gRPC

gRPC — современный высокопроизводительный фреймворк с открытым исходным кодом для *удаленного вызова процедур (remote procedure call, RPC)*. Он работает в различных средах и позволяет одному компьютеру вызывать процедуры на другом по сети так, будто они выполняются локально. Воплощает пример двухуровневой архитектуры «клиент — сервис».

Принцип работы gRPC

Разработчик gRPC-сервиса определяет его интерфейс, задавая удаленные методы, их параметры и типы возвращаемых данных. Затем этот интерфейс реализуется на сервере, который принимает и обрабатывает вызовы.

На клиентской стороне используется строго типизированный gRPC-клиент, который поддерживает те же методы, что и сервер.

Определение контрактов gRPC в .proto-файлах

В gRPC применяется контрактно-ориентированный подход к разработке API, не зависящий от языка. Контракт — это формальное описание сервиса: список методов с их параметрами и возвращаемыми значениями. Контракт гарантирует,

что сервис будет соответствовать заданным требованиям и не изменит сигнатуры методов со временем.

Контракты описываются в файлах с расширением `.proto`, которые имеют собственный синтаксис. Затем эти файлы компилируются в код для различных языков программирования, например C#. Сервер и клиент используют `.proto`-файлы для обмена сообщениями в правильном формате.

Ниже приведен пример `.proto`-файла с синтаксисом `proto3`, в котором определен запрос сообщения и используется пользовательское значение `enum`:

```
// Синтаксис указывается в первой незакомментированной строке
syntax = "proto3"; // Синтаксис proto2 используется по умолчанию

/* Этот .proto-файл в проекте .NET будет использовать указанное
   пространство имен C# для автоматически генерируемых файлов. */
option csharp_namespace = "Northwind.Grpc.Service";

enum SearchType {
  SEARCHTYPE_UNSPECIFIED = 0;
  SEARCHTYPE_STARTSWITH = 1;
  SEARCHTYPE_CONTAINS = 2;
  SEARCHTYPE_ENDSWITH = 3;
}

message SearchRequest {
  string query = 1; // Поля должны содержать номера заказов
  SearchType search_type = 2;
  int32 page = 3;
  int32 page_size = 4;
}

message SearchResponse {
  /* Типы сообщений могут быть вложены и/или повторяться
     для эквивалента коллекций или массивов. */
  repeated SearchResult results = 1;
}

message SearchResult {
  string url = 1;
  string title = 2;
  repeated string authors = 3;
}

service Searcher {
  rpc PerformSearch (SearchRequest) returns (SearchResponse);
}
```

Каждое поле сообщения должно иметь уникальный номер в диапазоне 1–536 870 911. Диапазон 19 000–19 999 зарезервирован для системных нужд `Protobuf` и не может использоваться. Эти номера заменяют имена полей при сериализации для экономии места в бинарном формате.

В копилку знаний: в руководстве по стилю Protobuf рекомендуется:

- использовать строчные буквы без подчеркиваний для имен полей;
- использовать прописные буквы с подчеркиваниями для значений перечислений и т. д.

Инструментарий C# автоматически адаптирует стили к стандартам .NET. Подробнее можно прочитать по адресу https://protobuf.dev/programming-guides/style/.

Дельный совет: в gRPC используется очень эффективный формат привязки данных, поэтому номера полей нельзя изменять после их использования — это эквивалентно удалению старого поля и добавлению нового. Также нельзя повторно использовать номера полей, так как это может привести к ошибкам. О последствиях неправильного использования номеров полей можно прочитать по адресу https://protobuf.dev/programming-guides/proto3/#consequences.

Поля не поддерживают значение `null`, поэтому все числовые типы по умолчанию равны нулю (`0`). Числовые и другие типы данных полей приведены в табл. 13.1.

Таблица 13.1. Числовые и другие типы полей в Protobuf

Тип	Описание
`string`	Текстовые строки (по умолчанию пустая строка)
`bool`	Логические значения (по умолчанию `false`)
`int32`, `int64`	32- и 64-битные целые числа со знаковым кодированием (эквивалент `int` и `long` в C#). Хотя эти типы и можно использовать для хранения отрицательных значений, лучше воспользоваться типами `sint32` и `sint64`
`sint32`, `sint64`	32- и 64-битные оптимизированные целые числа (эквивалент `int` и `long` в C#)
`uint32`, `uint64`	32- и 64-битные беззнаковые целые числа (`uint` и `ulong` в C#)
`fixed32`, `fixed64`, `sfixed32`, `sfixed64`	Целые числа фиксированного размера. Размер 4 байта у 32-битных типов и 8 байт — у 64-битных. Эквиваленты в C# — `uint` и `ulong`, а также `int` и `long`
`float`, `double`	Вещественные числа с плавающей точкой
`bytes`	Двоичные данные до 2^{32} байт (4 294 967 296). Для создания экземпляра используется метод `ByteString.CopyFrom(byte[]data)`. Для создания массива данных — метод `ToByteArray()`. По умолчанию используется пустое значение `ByteString`

В копилку знаний: официальное руководство опубликовано по адресу https://protobuf.dev/programming-guides/proto3/.

Преимущества gRPC

Фреймворк gRPC снижает нагрузку на сеть благодаря бинарной машинной сериализации *Protobuf*, которая эффективнее JSON и XML, используемых в веб-сервисах. Протокол HTTP/2, необходимый для gRPC, обеспечивает такие преимущества, как бинарный фрейминг, сжатие и мультиплексирование вызовов через одно соединение.

Слой бинарного фрейминга (binary framing) определяет способ передачи HTTP-сообщений между клиентом и сервером. В HTTP/1.x данные передаются в текстовом формате с разделителями строк, тогда как HTTP/2 разбивает их на более мелкие бинарные фреймы. Мультиплексирование позволяет объединять нескольких сообщений из разных источников в одно для оптимизации использования общего ресурса, например сетевого транспортного протокола.

В копилку знаний: дополнительная информация о HTTP/2 и его влиянии на gRPC доступна по адресу https://grpc.io/blog/ grpc-on-http2/.

Ограничения gRPC

Главное ограничение — невозможность работы в браузерах из-за недостаточного контроля над HTTP/2 на стороне клиента. К примеру, браузеры не позволяют вызывающему объекту требовать использования HTTP/2.

Кроме того, бинарный формат сообщений затрудняет диагностику и анализ, так как большинство инструментов не умеют его обрабатывать и не могут отображать сообщения в удобочитаемом формате.

Проект *gRPC-Web* добавляет прокси-уровень, позволяющий использовать часть возможностей gRPC в браузере и пересылать запросы на сервер gRPC, но из-за ограничений HTTP/2 поддерживаются не все сценарии.

Типы методов gRPC

В gRPC предусмотрены четыре типа методов.

- *Унарные методы* — наиболее распространенный вариант со структурированными запросами и ответами. Клиент отправляет запрос, сервер возвращает ответ, после чего вызов завершается. Подход годится для большинства сценариев, где не требуется поток.
- *Потоковые методы* используются для передачи больших объемов данных в виде потока байтов. В них применяется ключевое слово `stream` для входных и/или выходных параметров. Три типа потоковых методов:

- *серверные потоковые* — сервер получает запрос от клиента и отправляет несколько ответных сообщений (поток). Работа продолжается, пока сервер не завершит поток или не получит токен отмены;
- *клиентские потоковые* — клиент отправляет поток данных, сервер принимает их и после обработки возвращает единственный ответ. После этого вызов клиентского потокового метода завершается;
- *двунаправленные потоковые* — клиент и сервер обмениваются сообщениями в обоих направлениях без строгой последовательности. Сервер получает поток от клиента без оповещений и возвращает данные исключительно через второй поток. Вызов завершается после возврата результата.

В этой книге подробно рассматриваются только унарные методы. Если вас интересуют потоковые методы, сообщите мне, и я их опишу в следующем издании.

gRPC-пакеты корпорации Microsoft

С мая 2021 года компания Microsoft рекомендует свою реализацию gRPC для .NET. Набор .NET-пакетов компании Microsoft для работы с gRPC включает:

- `Grpc.AspNetCore` — для размещения gRPC-сервисов в ASP.NET Core;
- `Grpc.Net.Client` — для создания gRPC-клиентов в .NET с использованием `HttpClient`;
- `Grpc.Net.ClientFactory` — для интеграции gRPC-клиентов в проекты на основе `HttpClientFactory`.

Подробности доступны в репозитории по адресу https://github.com/grpc/grpc-dotnet.

Создание сервиса и клиента gRPC

Создадим сервис и клиент для отправки/получения простых сообщений.

Создание gRPC-сервиса Hello World

Начнем с проектирования gRPC-сервиса на основе стандартного шаблона.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта ASP.NET Core gRPC Service (Служба gRPC ASP.NET Core) / `grpc`;
   - в поле Solution name (Имя решения) укажите значение `Chapter13`;

- в поле Project name (Имя проекта) укажите значение `Northwind.Grpc.Service`;
- снимите флажок Enable Docker (Включить поддержку контейнера);
- снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
- установите флажок Enable native AOT publishing (Включить публикацию Native AOT).

Дельный совет: перед публикацией убедитесь, что флажок Enable Native AOT publishing (Включить публикацию Native AOT) установлен. В .NET 8 и более поздних версиях проекты gRPC для нативных платформ могут быть скомпилированы в режиме Native AOT. Благодаря этому значительно повышается производительность и сокращается время запуска, что особенно важно для микросервисов, часто развертываемых и масштабируемых динамически.

2. В папке `Protos` откройте файл `greet.proto`. В нем определен сервис `Greeter` с методом `SayHello`, который принимает сообщение `HelloRequest` и возвращает `HelloReply`:

```
syntax = "proto3";

option csharp_namespace = "Northwind.Grpc.Service";

package greet;

// Определение сервиса приветствия
service Greeter {
  // Отправка приветствия
  rpc SayHello (HelloRequest) returns (HelloReply);
}

// Сообщение запроса, содержащее имя пользователя
message HelloRequest {
  string name = 1;
}

// Ответное сообщение, содержащее приветствия
message HelloReply {
  string message = 1;
}
```

Для работы с .proto-файлами в программе Visual Studio Code установите расширение vscode-proto3 (zxh404.vscode-proto3). В программе JetBrains Rider используйте плагин Protocol Buffers компании JetBrains (https://plugins.jetbrains.com/plugin/14004-protocol-buffers).

3. В файле проекта `Northwind.Grpc.Service.csproj` включите режим публикации Native AOT. Проверьте также, что `.proto`-файл зарегистрирован на стороне сервера, а пакет для реализации gRPC добавлен в ASP.NET Core:

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>true</InvariantGlobalization>
    <PublishAot>true</PublishAot>
  </PropertyGroup>

  <ItemGroup>
    <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.59.0" />
  </ItemGroup>

</Project>
```

В JetBrains Rider вручную добавьте строку <PublishAot>true</PublishAot>, если ее нет по умолчанию.

4. В файле проекта присвойте параметру инвариантной глобализации (`InvariantGlobalization`) значение `false`:

```
<InvariantGlobalization>false</InvariantGlobalization>
```

5. В папке `Services` откройте файл `GreeterService.cs`. Класс `GreeterService` наследуется от `Greeter.GreeterBase`, асинхронно реализует контракт сервиса и содержит метод `SayHello`, принимающий `HelloRequest` и возвращающий `HelloReply`:

```
using Grpc.Core;
using Northwind.Grpc.Service

namespace Northwind.Grpc.Service.Services
{
  public class GreeterService : Greeter.GreeterBase
  {
    private readonly ILogger<GreeterService> _logger;
```

```
    public GreeterService(ILogger<GreeterService> logger)
    {
      _logger = logger;
    }

    public override Task<HelloReply> SayHello(
      HelloRequest request, ServerCallContext context)
    {
      return Task.FromResult(new HelloReply
      {
        Message = "Hello " + request.Name
      });
    }
  }
}
```

6. На панели Solution Explorer (Обозреватель решений) программы Visual Studio 2022 нажмите кнопку Show All Files (Показать все файлы). В JetBrains Rider щелкните кнопкой мыши на значке в виде глаза на панели Solution (Решение).
7. В папке `obj\Debug\net8.0\Protos` присутствуют два файла классов, `Greet.cs` и `GreetGrpc.cs`, которые автоматически генерируются из файла `greet.proto` (рис. 13.2).

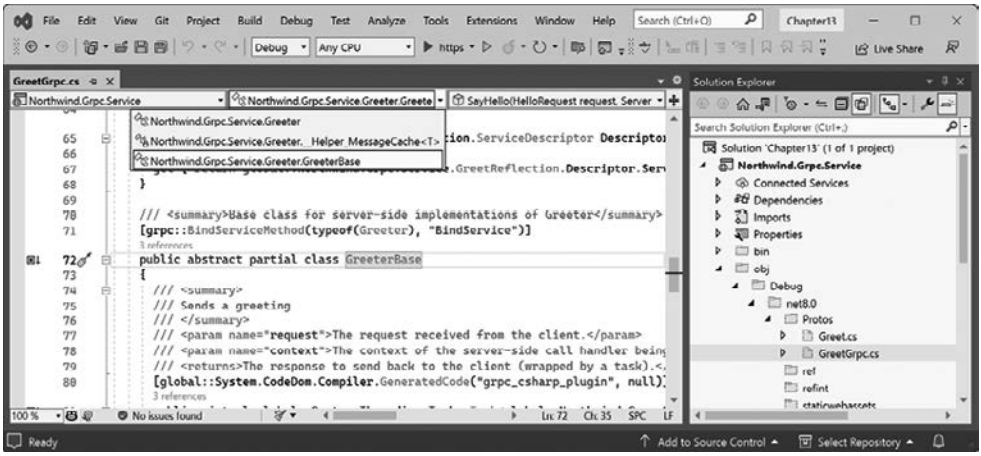


Рис. 13.2. Автоматически генерируемые классы из .proto-файла gRPC-сервиса

8. В файле `GreetGrpc.cs` обратите внимание, что класс `GreeterService` унаследован от `Greeter.GreeterBase`. Вам не обязательно разбираться в деталях его реализации, достаточно знать, что он обрабатывает всю логику взаимодействия gRPC.
9. На панели Solution Explorer (Обозреватель решений) программы Visual Studio 2022 разверните раздел Dependencies ▸ Packages ▸ Grpc.AspNetCore (Зависимости ▸ Пакеты ▸ Grpc.AspNetCore) и обратите внимание, что пакет `Grpc.Asp-`

NetCore зависит от пакетов Google.Protobuf, Microsoft.Grpc.AspNetCore.Server.ClientFactory и Grpc.Tools компании Microsoft (рис. 13.3).

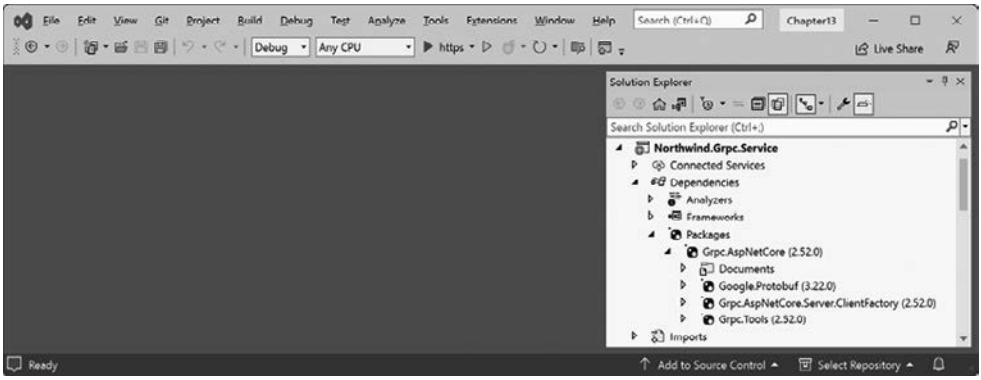


Рис. 13.3. Пакет Grpc.AspNetCore зависит от пакетов Google.Protobuf, Microsoft.Grpc.AspNetCore.Server.ClientFactory и Grpc.Tools

Пакет Grpc.Tools генерирует классы C# из .proto-файлов, используя типы из Google.Protobuf для сериализации данных в формат Protobuf. Пакет Grpc.AspNetCore.Server.ClientFactory добавляет поддержку gRPC как на стороне сервера, так и на стороне клиента.

10. В файле Program.cs найдите вызов метода AddGrpc в коллекции сервисов и убедитесь, что он добавлен:

```
builder.Services.AddGrpc();
```

11. В этом же файле найдите настройку HTTP-конвейера и проверьте вызов MapGrpcService<GreeterService>(), сопоставляющий сервис Greeter:

```
app.MapGrpcService<GreeterService>();
```

12. Откройте файл launchSettings.json, расположенный в папке Properties. Измените параметр applicationUrl для профиля https, указав порт 5131 для протокола HTTPS и 5132 — для HTTP:

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "http://localhost:5132",
      "environmentVariables": {
```

```
          "ASPNETCORE_ENVIRONMENT": "Development"
        }
      },
      "https": {
        "commandName": "Project",
        "dotnetRunMessages": true,
        "launchBrowser": false,
        "applicationUrl": "https://localhost:5131;http://localhost:5132",
        "environmentVariables": {
          "ASPNETCORE_ENVIRONMENT": "Development"
        }
      }
    }
  }
```

13. Выполните сборку проекта `Northwind.Grpc.Service`.

Конфигурация файлов проекта

Перед продолжением разберемся с конфигурацией элементов проекта и соответствующим синтаксисом.

В Visual Studio 2022 свойства элементов задаются атрибутами:

```
<Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
```

Свойствам, не заданным явно, присваиваются значения по умолчанию.

В других редакторах, таких как JetBrains Rider, используются дочерние элементы:

```
<Protobuf>
  <Include>Protos\greet.proto</Include>
  <GrpcServices>Client</>
  <Access>Public</Access>
  <ProtoCompile>True</ProtoCompile>
  <CompileOutputs>True</CompileOutputs>
  <OutputDir>obj\Debug\net8.0\</OutputDir>
  <Generator>MSBuild:Compile</Generator>
<Protobuf>
```

Оба варианта приводят к одному результату, но первый предпочтительнее, так как короче и удобнее.

Создание gRPC-клиента Hello World

Теперь создадим проект сайта MVC ASP.NET Core и добавим пакеты gRPC-клиента, чтобы он мог вызывать gRPC-сервис.

1. В редакторе кода создайте проект со следующими настройками:
   - выберите шаблон проекта ASP.NET Core Web App (Model-View-Controller) (Веб-приложение ASP.NET Core (Модель — представление — контроллер)) / `mvc`;

- в поле Solution name (Имя решения) укажите значение `Chapter13`;
- в поле Project name (Имя проекта) укажите значение `Northwind.Grpc.Client.Mvc`;
- в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
- установите флажок Configure for HTTPS (Настроить для HTTPS);
- снимите флажок Enable Docker (Включить поддержку контейнера);
- снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня).

2. В файле созданного проекта включите строгую обработку предупреждений, добавьте ссылки на пакеты `Grpc.Net.ClientFactory`, `Grpc.Tools` и `Google.Protobuf`:

```
<ItemGroup>
  <PackageReference Include="Google.Protobuf" Version="3.24.4" />
  <PackageReference Include="Grpc.Net.ClientFactory" Version="2.57.0" />
  <PackageReference Include="Grpc.Tools" Version="2.58.0">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles;
      analyzers; buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

Дельный совет: пакет Grpc.Net.ClientFactory зависит от пакета Grpc.Net.Client, реализующего поддержку gRPC на стороне клиента, но не содержит ссылок на пакеты Grpc.Tools и Google.Protobuf. Их нужно добавить вручную. Пакет Grpc.Tools используется только во время разработки, поэтому указываем PrivateAssets=all, чтобы он не попадал в публикуемую сборку.

3. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5133 для протокола HTTPS и 5134 — для HTTP:

```
"profiles": {
  ...
  "https": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "applicationUrl": "https://localhost:5133;http://localhost:5134",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
```

4. Скопируйте папку `Protos` из проекта/папки `Northwind.Grpc.Service` в проект/папку `Northwind.Grpc.Client.Mvc`.

В программе Visual Studio 2022 для копирования папку можно просто перетащить мышью. В Visual Studio Code или JetBrains Rider перетаскивайте папку, нажав и удерживая клавишу Ctrl (⌘ в macOS).

5. В папке `Protos` проекта `Northwind.Grpc.Client.Mvc` откройте файл `greet.proto`. Измените пространство имен, чтобы оно соответствовало клиентскому проекту и автоматически сгенерированные классы находились в том же пространстве:

```
option csharp_namespace = "Northwind.Grpc.Client.Mvc";
```

6. В файле проекта `Northwind.Grpc.Client.Mvc` найдите или добавьте группу элементов для регистрации `.proto`-файла и укажите, что он используется на стороне клиента:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

Программа Visual Studio 2022 создаст группу элементов автоматически, но по умолчанию использует параметр GrpcServices="Server". Нужно вручную присвоить ему значение Client. В других редакторах кода, вероятно, придется полностью написать элемент ItemGroup. В JetBrains Rider используется больше элементов конфигурации. Не обращайте на них внимание.

7. Выполните сборку проекта `Northwind.Grpc.Client.Mvc`, чтобы убедиться, что классы сгенерировались.
8. В папке `obj\Debug\net8.0\Protos` проекта `Northwind.Grpc.Client.Mvc` откройте файл `GreetGrpc.cs` и найдите класс `Greeter.GreeterClient` (ниже показан фрагмент файла):

```
public static partial class Greeter
{
  ...
  public partial class GreeterClient : grpc::ClientBase<GreeterClient>
  {
```

9. В файле `Program.cs` импортируйте пространство имен `Greeter.GreeterClient`:

```
using Northwind.Grpc.Client.Mvc; // Для класса Greeter.GreeterClient
```

10. В том же файле, в разделе настройки сервисов, зарегистрируйте `GreeterClient` как именованный gRPC-клиент, который будет взаимодействовать с сервисом, прослушивающим порт `5131`:

```
builder.Services.AddGrpcClient<Greeter.GreeterClient>("Greeter",
  options =>
  {
    options.Address = new Uri("https://localhost:5131");
  });
```

11. В папке `Models` создайте файл класса `HomeIndexViewModel.cs`.
12. В созданном файле определите класс для хранения сообщений об ошибках и приветствия:

```
namespace Northwind.Grpc.Client.Mvc.Models;

public class HomeIndexViewModel
{
  public string? Greeting { get; set; }
  public string? ErrorMessage { get; set; }
}
```

13. В папке `Controllers` откройте файл `HomeController.cs`. Импортируйте в нем пространство имен для работы с фабрикой gRPC-клиентов:

```
using Grpc.Net.ClientFactory; // Для класса GrpcClientFactory
```

14. В классе `Controller` объявите поле для хранения клиента `GreeterClient` и инициализируйте его в конструкторе:

```
public class HomeController : Controller
{
  private readonly ILogger<HomeController> _logger;
  private readonly Greeter.GreeterClient _greeterClient;

  public HomeController(ILogger<HomeController> logger,
    GrpcClientFactory factory)
  {
    _logger = logger;
    _greeterClient = factory.CreateClient<Greeter.GreeterClient>("Greeter");
  }
```

15. В методе `Index` добавьте строковый параметр `name="Henrietta"`. Затем выполните асинхронный вызов `SayHelloAsync` с объектом `HelloRequest` и передайте полученные данные `HelloReply` в словарь объектов `ViewData`, отлавливая потенциальные исключения:

```
public async Task<IActionResult> Index(string name = "Henrietta")
{
  HomeIndexViewModel model = new();

  try
  {
    HelloReply reply = await _greeterClient.SayHelloAsync(
      new HelloRequest { Name = name });

    model.Greeting = "Greeting from gRPC service: " + reply.Message;
  }
  catch (Exception ex)
  {
    _logger.LogWarning($"Northwind.Grpc.Service is not responding.");
```

```
      model.ErrorMessage = ex.Message;
    }

    return View(model);
  }
```

16. В представлении `Views/Home/Index.cshtml` после заголовка `Welcome` замените элемент `p` формой для ввода посетителем имени и отображение ответа gRPC-сервиса с приветствием:

```
@using Northwind.Grpc.Client.Mvc.Models
@model HomeIndexViewModel
@{
  ViewData["Title"] = "Home Page";
}
<div class="text-center">
  <h1 class="display-4">Welcome</h1>
  <div class="alert alert-secondary">
    <form>
      <input name="name" placeholder="Enter your name" />
      <input type="submit" />
    </form>
  </div>
  @if (Model.Greeting is not null)
  {
    <p class="alert alert-primary">@Model.Greeting</p>
  }
  @if (Model.ErrorMessage is not null)
  {
    <p class="alert alert-danger">@Model.ErrorMessage</p>
  }
</div>
```

Если очистить проект gRPC, автоматически сгенерированные классы удалятся, вызвав ошибки компиляции. Чтобы их восстановить, внесите изменения в .proto-файл или перезапустите проект.

Тестирование сервиса и клиента gRPC

Теперь мы можем запустить gRPC-сервис и проверить, получается ли сайту MVC его вызвать.

1. Запустите проект `Northwind.Grpc.Service` без отладки.
2. Запустите проект `Northwind.Grpc.Client.Mvc`.
3. При необходимости откройте браузер и перейдите на главную страницу по адресу `https://localhost:5133/`.

4. Проверьте, что на главной странице отображается приветствие (рис. 13.4).

Рис. 13.4. Главная страница с приветствием после вызова gRPC-сервиса

5. Проанализируйте вывод в терминале проекта MVC ASP.NET Core. Обратите внимание на уведомления об обработке `POST`-запроса HTTP/2 конечной точкой `greet.Greeter/SayHello` примерно за 41 миллисекунду:

```
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[100]
      Start processing HTTP request POST https://localhost:5131/greet.Greeter/
      SayHello
info: System.Net.Http.HttpClient.Greeter.ClientHandler[100]
      Sending HTTP request POST https://localhost:5131/greet.Greeter/SayHello
info: System.Net.Http.HttpClient.Greeter.ClientHandler[101]
      Received HTTP response headers after 60.5352ms - 200
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[101]
      End processing HTTP request after 69.1623ms - 200
```

6. Введите свое имя и отправьте форму.
7. Закройте браузер и завершите работу веб-сервера.

Теперь у вас есть рабочий gRPC-сервис и клиент.

Реализация gRPC в модели EF Core

Добавим в проект gRPC-сервис для работы с БД Northwind.

Создание gRPC-сервиса

Мы продолжим работу с моделью EF Core, созданной в главе 3, и реализуем gRPC-сервис. Для этого сначала определим контракт с помощью `.proto`-файла, а затем напишем сам сервис. Начнем с таблицы `Shippers`, так как она содержит всего три

свойства: одно числовое (`int`) и два строковых (`string`), а в БД хранится всего три записи. Приступим.

1. В файле проекта `Northwind.Grpc.Service` добавьте ссылку на проект контекста БД Northwind:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.DataContext.
  ➥ SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
</ItemGroup>
```

Убедитесь, что путь Include не содержит символов разрыва строки.

2. В терминале создайте проект `Northwind.Grpc.Service` с помощью команды `dotnet build`.
3. В папке `Protos` проекта `Northwind.Grpc.Service` создайте файл `shipper.proto` (шаблон *Protocol Buffer File* в Visual Studio 2022):

```
syntax = "proto3";

option csharp_namespace = "Northwind.Grpc.Service";

package shipper;

service Shipper {
  rpc GetShipper (ShipperRequest) returns (ShipperReply);
}

message ShipperRequest {
  int32 shipper_id = 1;
}

message ShipperReply {
  int32 shipper_id = 1;
  string company_name = 2;
  string phone = 3;
}
```

4. Откройте файл проекта и добавьте запись для включения файла `shipper.proto`:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
  <Protobuf Include="Protos\shipper.proto" GrpcServices="Server" />
</ItemGroup>
```

5. Выполните сборку проекта `Northwind.Grpc.Service`.
6. В папке `Services` создайте файл класса `ShipperService.cs` и определите в нем сервис для работы с грузоотправителями. Этот сервис должен использовать контекст БД `Northwind` для получения данных грузоотправителей:

```
using Grpc.Core; // Для контекста ServerCallContext
using Northwind.EntityModels; // Для контекста NorthwindContext
using ShipperEntity = Northwind.EntityModels.Shipper;

namespace Northwind.Grpc.Service.Services;

public class ShipperService : Shipper.ShipperBase
{
  private readonly ILogger<ShipperService> _logger;
  private readonly NorthwindContext _db;

  public ShipperService(ILogger<ShipperService> logger,
    NorthwindContext context)
  {
    _logger = logger;
    _db = context;
  }

  public override async Task<ShipperReply?> GetShipper(
    ShipperRequest request, ServerCallContext context)
  {
    ShipperEntity? shipper = await _db.Shippers
      .FindAsync(request.ShipperId);

    return shipper is null ? null : ToShipperReply(shipper);
  }

  // Метод сопоставления для преобразования экземпляра Shipper
  // в модели сущности в экземпляр ShipperReply в gRPC
  private ShipperReply ToShipperReply(ShipperEntity shipper)
  {
    return new ShipperReply
    {
      ShipperId = shipper.ShipperId,
      CompanyName = shipper.CompanyName,
      Phone = shipper.Phone
    };
  }
}
```

.proto-файл генерирует классы, представляющие сообщения, передаваемые в gRPC-сервисе и получаемые из него. Поэтому нельзя напрямую использовать классы сущностей из модели EF Core. Чтобы сопоставить их, потребуется вспомогательный метод ToShipperReply, преобразующий экземпляр Shipper в объект ShipperReply. Можно воспользоваться инструментом AutoMapper, но в данном случае преобразование настолько простое, что его можно написать вручную.

7. В файле `Program.cs` импортируйте пространство имен контекста БД Northwind:

```
using Northwind.EntityModels; // Для метода AddNorthwindContext
```

8. В разделе настройки сервисов добавьте регистрацию контекста БД Northwind:

```
builder.Services.AddNorthwindContext();
```

9. В разделе конфигурации HTTP-конвейера после вызова регистрации `GreeterService` добавьте выражение для регистрации класса `ShipperService`:

```
app.MapGrpcService<ShipperService>();
```

Реализация gRPC-клиента

Теперь добавим клиент для взаимодействия с сервисом на сайте MVC Northwind.

1. Скопируйте `shipper.proto` из папки `Protos` проекта `Northwind.Grpc.Service` в папку `Protos` проекта `Northwind.Grpc.Client.Mvc`.
2. В файле `shipper.proto`, находящемся в папке проекта `Northwind.Grpc.Client.Mvc`, измените пространство имен так, чтобы оно соответствовало текущему проекту. В этом случае автоматически сгенерированные классы будут находиться в нужном пространстве имен:

```
option csharp_namespace = "Northwind.Grpc.Client.Mvc";
```

3. Откройте файл проекта `Northwind.Grpc.Client.Mvc` и добавьте или измените запись для регистрации `.proto`-файла, используемого на стороне клиента:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
  <Protobuf Include="Protos\shipper.proto" GrpcServices="Client" />
</ItemGroup>
```

В некоторых редакторах кода (например, JetBrains Rider) могут появляться дополнительные настройки. Лучше упростить элементы, как показано в разметке, иначе возможны ошибки.

4. В файле `Program.cs` проекта `Northwind.Grpc.Client.Mvc` добавьте код регистрации класса `ShipperClient` для подключения к gRPC-сервису, прослушивающему порт 5131:

```
builder.Services.AddGrpcClient<Shipper.ShipperClient>("Shipper", options =>
  {
    options.Address = new Uri("https://localhost:5131");
  });
```

5. В папке `Models` откройте файл `HomeIndexViewModel.cs` и добавьте свойство для хранения информации о грузоотправителе:

```
public string? ShipperSummary { get; set; }
```

6. В папке `Controllers` откройте файл `HomeController.cs` и объявите поле для хранения экземпляра `ShipperClient`. В конструкторе установите его с помощью фабрики клиентов:

```
public class HomeController : Controller
{
  private readonly ILogger<HomeController> _logger;
  private readonly Greeter.GreeterClient _greeterClient;
  private readonly Shipper.ShipperClient _shipperClient;

public HomeController(ILogger<HomeController> logger,
  GrpcClientFactory factory)
{
  _logger = logger;
  _greeterClient = factory.CreateClient<Greeter.GreeterClient>("Greeter");
  _shipperClient = factory.CreateClient<Shipper.ShipperClient>("Shipper");
}
```

7. В методе `Index` контроллера `HomeController` добавьте параметр `id` и вызов gRPC-сервиса `Shipper` для получения данных грузоотправителя по идентификатору `ShipperId`:

```
public async Task<IActionResult> Index(
  string name = "Henrietta", int id = 1)
{
  HomeIndexViewModel model = new();

  try
  {
    HelloReply reply = await greeterClient.SayHelloAsync(
      new HelloRequest { Name = name });

    model.Greeting = "Greeting from gRPC service: " + reply.Message;

    ShipperReply shipperReply = await _shipperClient.GetShipperAsync(
      new ShipperRequest { ShipperId = id });

    model.ShipperSummary = "Shipper from gRPC service: " +
      $"ID: {shipperReply.ShipperId}, Name: {shipperReply.CompanyName},"
      + $" Phone: {shipperReply.Phone}.";
  }
  catch (Exception ex)
  {
    _logger.LogWarning($"Northwind.Grpc.Service is not responding.");

    model.ErrorMessage = ex.Message;
  }

  return View();
}
```

8. В папке `Views/Home` откройте файл `Index.cshtml` и добавьте форму для ввода идентификатора грузоотправителя. Также добавьте код для отображения информации о нем после приветствия:

```
@using Northwind.Grpc.Client.Mvc.Models
@model HomeIndexViewModel
@{
  ViewData["Title"] = "Home Page";
}
<div class="text-center">
  <h1 class="display-4">Welcome</h1>
  <div class="alert alert-secondary">
    <form>
      <input name="name" placeholder="Enter your name" />
      <input type="submit" />
    </form>
    <form>
      <input name="id" placeholder="Enter a shipper id" />
      <input type="submit" />
    </form>
  </div>
  @if (Model.Greeting is not null)
  {
    <p class="alert alert-primary">@Model.Greeting</p>
  }
  @if (Model.ErrorMessage is not null)
  {
    <p class="alert alert-danger">@Model.ErrorMessage</p>
  }
  @if (Model.ShipperSummary is not null)
  {
    <p class="alert alert-primary">@Model.ShipperSummary</p>
  }
</div>
```

9. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
10. Запустите проект `Northwind.Grpc.Service` без отладки.
11. Запустите проект `Northwind.Grpc.Client.Mvc`.
12. Откройте браузер и перейдите на главную страницу сайта MVC по адресу `https://localhost:5133/`.
13. Обратите внимание на исключение в сервисе gRPC. Оно возникает, потому что метод `GetShipper` использует EF Core, который при динамической компиляции LINQ-запроса не поддерживается в Native AOT (фрагмент ошибки показан ниже):

```
fail: Grpc.AspNetCore.Server.ServerCallHandler[6]
      Error when executing service method 'GetShipper'.
```

```
      System.PlatformNotSupportedException: Dynamic code generation is not
supported on this platform.
at System.Reflection.Emit.AssemblyBuilder.ThrowDynamicCodeNotSupported()
...
      at Microsoft.EntityFrameworkCore.Storage.Database.CompileQuery[TResult]
(Expression query, Boolean async)
...
      at Northwind.Grpc.Service.Services.ShipperService.
GetShipper(ShipperRequest request, ServerCallContext context) in C:\
appsservices-net8\Chapter13\Northwind.Grpc.Service\Services\ShipperService.
cs:line 22
...
```

14. Закройте браузер и завершите работу веб-сервера.
15. В файле проекта закомментируйте параметр `PublishAot`:

```
<!--<PublishAot>true</PublishAot>-->
```

Может возникнуть вопрос, зачем включать AOT при создании проекта, если затем приходится его отключать. Причины две.

- Важно увидеть ошибку, чтобы знать, как ее исправить, если возникнет подобная ситуация в другом gRPC-проекте.
- В будущих версиях .NET (например, .NET 9 или .NET 10) появится поддержка EF Core в Native AOT.

16. Запустите проект `Northwind.Grpc.Service` без отладки.
17. Запустите проект `Northwind.Grpc.Client.Mvc`.
18. Убедитесь, что на странице сервиса корректно отображается информация о грузоотправителе (рис. 13.5).

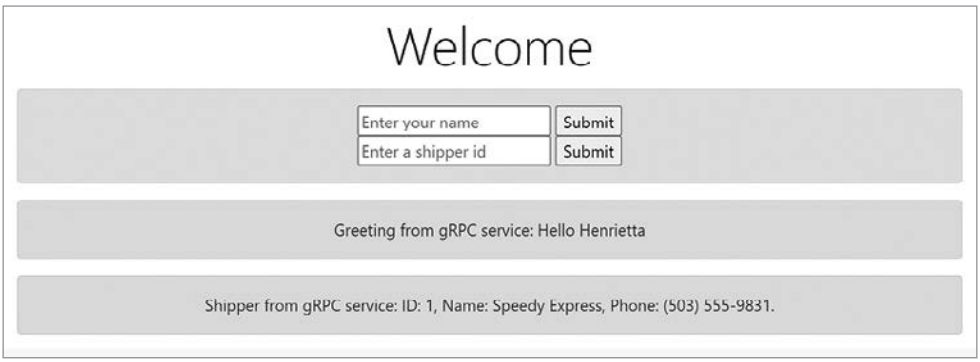


Рис. 13.5. Вид главной страницы после вызова gRPC-сервиса с запросом данных грузоотправителя

19. В БД Northwind есть три грузоотправителя с идентификаторами 1, 2 и 3. Проверьте работу сервиса, введя эти идентификаторы. Затем попробуйте ввести 4, которого в БД нет, и убедитесь, что обработка отсутствующих данных выполняется корректно.
20. Закройте браузер и завершите работу веб-сервера.

Улучшение gRPC-сервиса

Теперь рассмотрим более сложные темы: поддержку компиляции Native AOT, получение метаданных, установку сроков выполнения, обработку дат, времени и десятичных типов, добавление перехватчиков, а также обработку исключений и временных сбоев.

Улучшение gRPC-сервиса с помощью публикации Native AOT

В .NET 8 появилась поддержка gRPC для Native AOT. Однако, как вы уже видели, она пока несовместима с некоторыми частями .NET, например с Entity Framework Core.

Изменим gRPC-сервис, чтобы он работал с SQL-клиентом вместо EF Core. Код, связанный с EF Core, оставим в проекте, чтобы в будущем можно было к нему вернуться, например, при переходе на EF Core 9, если он получит поддержку Native AOT.

1. В файле проекта `Northwind.Grpc.Service` удалите параметр публикации AOT и подключите пакет SQL-клиента:

```
<PackageReference Include="Microsoft.Data.SqlClient" Verson="5.1.2" />
```

2. В файле `ShipperService.cs` (папка `Services`) импортируйте пространство имен для работы с `SqlClient`:

```
using Microsoft.Data.SqlClient; // Для класса SqlConnection и т. д.
using System.Data; // Для перечисления CommandType
```

3. В методе `GetShipper` закомментируйте код, который извлекает данные грузоотправителя через контекст данных Northwind, и замените его на получение данных с помощью `SqlClient`:

```
public override async Task<ShipperReply?> GetShipper(
  ShipperRequest request, ServerCallContext context)
{
```

```
    // Мы не можем использовать EF Core в проекте, скомпилированном с Native AOT
    // ShipperEntity? shipper = await _db.Shippers
    //   .FindAsync(request.ShipperId);

    SqlConnectionStringBuilder builder = new();

    builder.InitialCatalog = "Northwind";
    builder.MultipleActiveResultSets = true;
    builder.Encrypt = true;
    builder.TrustServerCertificate = true;
    builder.ConnectTimeout = 10; // По умолчанию 30 секунд
    builder.DataSource = "."; // Для локального SQL Server
    builder.IntegratedSecurity = true;

    /*
    // Для системы проверки подлинности SQL Server:
    builder.UserID = Environment.GetEnvironmentVariable("MY_SQL_USR");
    builder.Password = Environment.GetEnvironmentVariable("MY_SQL_PWD");
    builder.PersistSecurityInfo = false;
    */

    SqlConnection connection = new(builder.ConnectionString);

    await connection.OpenAsync();

    SqlCommand cmd = connection.CreateCommand();

    cmd.CommandType = CommandType.Text;
    cmd.CommandText = "SELECT ShipperId, CompanyName, Phone"
      + " FROM Shippers WHERE ShipperId = @id";
    cmd.Parameters.AddWithValue("id", request.ShipperId);

    SqlDataReader r = await cmd.ExecuteReaderAsync(
      CommandBehavior.SingleRow);

    ShipperReply? shipper = null;

    // Чтение одной ожидаемой строки
    if (await r.ReadAsync())
    {
      shipper = new()
      {
        ShipperId = r.GetInt32("ShipperId"),
        CompanyName = r.GetString("CompanyName"),
        Phone = r.GetString("Phone")
      };
    }

    await r.CloseAsync();

    return shipper;
  }
```

4. Включите параметр публикации AOT.
5. В файле `Program.cs` для создания веб-приложения можно использовать метод `CreateSlimBuilder`:

```
// Применение метода CreateSlimBuilder для уменьшения размера приложения
// при использовании параметра публикации проекта AOT.
// var builder = WebApplication.CreateSlimBuilder(args);
```

Метод CreateSlimBuilder не поддерживает HTTPS и HTTP/3, хотя их и можно добавить вручную. В этом случае придется использовать HTTP вместо HTTPS. В данной задаче остановимся на полном конструкторе, чтобы сохранить поддержку HTTPS.

6. В файле проекта `Northwind.Grpc.Service` добавьте элемент для эмуляции файлов, создаваемых компилятором:

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  ...
  <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
</PropertyGroup>
```

7. Выполните сборку проекта `Northwind.Grpc.Service`.
8. На панели Solution Explorer (Обозреватель решений) программы Visual Studio 2022 нажмите кнопку Show All Files (Показать все файлы). В JetBrains Rider щелкните кнопкой мыши на значке в виде глаза на панели Solution (Решение).
9. Откройте папку `obj\Debug\net8.0\generated` и обратите внимание на папки и файлы, созданные генераторами исходного кода для AOT и JSON-сериализации (рис. 13.6).

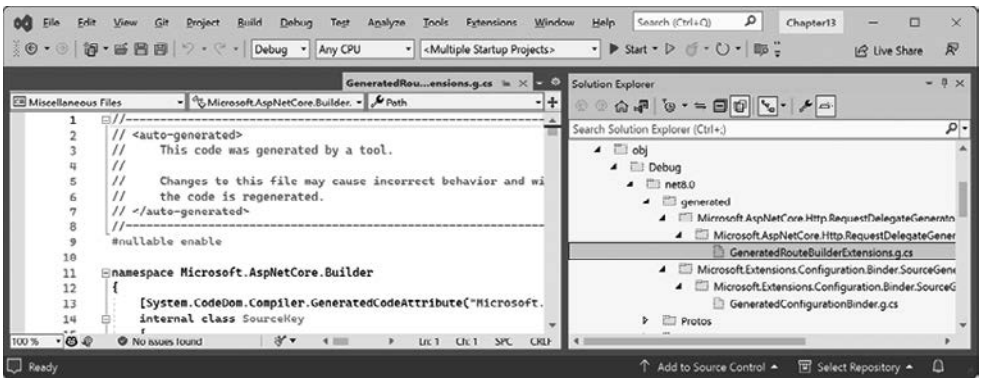


Рис. 13.6. Папки и файлы, созданные генераторами исходного кода в проекте gRPC AOT

10. В терминале выполните публикацию gRPC-сервиса с Native AOT:

```
dotnet publish
```

11. Обратите внимание на сообщения о генерации нативного кода и предупреждения об обрезке пакетов, например `Microsoft.Data.SqlClient` (ниже показан фрагмент):

```
Generating native code
...
C:\Users\markj\.nuget\packages\microsoft.data.sqlclient\5.1.1\runtimes\win\
lib\net6.0\Microsoft.Data.SqlClient.dll : warning IL2104: Assembly
'Microsoft.Data.SqlClient' produced trim warnings. For more information
see https://aka.ms/
dotnet-illink/libraries [C:\apps-services-net8\Chapter13\Northwind.Grpc.Service\
Northwind.Grpc.Service.csproj]
...
```

12. На панели File Explorer (Обозреватель файлов) откройте папку `bin\Release\net8.0\win-x64\publish` и найдите EXE-файл размером примерно 45 Мбайт. Для работы сервиса на другом компьютере под управлением Windows нужны только этот файл и библиотека `Microsoft.Data.SqlClient.SNI.dll`. Файлы `appsettings.json` требуются лишь для переопределения конфигурации, а PDB-файлы — для отладки, и только потому, что мы оставили код EF Core для возможности переключиться на публикацию без AOT.
13. Откройте папку `bin\Release\net8.0\win-x64\publish` в терминале.
14. Запустите файл `Northwind.Grpc.Service.exe`, явно указав URL и порт:

```
Northwind.Grpc.Service.exe --urls "https://localhost:5131"
```

Файлы launchSettings.json используются только в среде разработки, такой как Visual Studio 2022, и не влияют на рабочее развертывание. Указанные в них порты игнорируются и не используются работающим сервисом.

15. Запустите проект `Northwind.Grpc.Client.Mvc`.
16. Проверьте, что веб-страница корректно отображает грузоотправителя с идентификатором `1` и позволяет искать других грузоотправителей.
17. Закройте браузер и завершите работу веб-сервера.

В копилку знаний: вы можете узнать больше о gRPC и Native AOT на странице по адресу https://learn.microsoft.com/ru-ru/aspnet/core/grpc/native-aot.

Работа с метаданными запроса и ответа

Помимо передаваемых сообщений запроса и ответа, gRPC поддерживает метаданные в заголовках и трейлерах. Эти данные представляют собой словари, передаваемые вместе с сообщениями.

Давайте разбираться, как получить метаданные о вызове gRPC.

1. В файле `HomeController.cs` (проект `Northwind.Grpc.Client.Mvc`, папка `Controllers`) импортируйте пространство имен для использования класса `AsyncUnaryCall<T>`:

```
using Grpc.Core; // Для класса AsyncUnaryCall<T>.
```

2. В методе `Index` закомментируйте код вызова сервиса доставки gRPC. Вместо него:
   - получите объект `AsyncUnaryCall<T>`;
   - извлеките заголовки и выведите их в журнал;
   - получите ответ сервиса:

```
// ShipperReply shipperReply = await _shipperClient.GetShipperAsync(
//   new ShipperRequest { ShipperId = id });

// Тот же вызов, что и выше, но без ожидания
AsyncUnaryCall<ShipperReply> shipperCall = _shipperClient.
GetShipperAsync(
  new ShipperRequest { ShipperId = id });

Metadata metadata = await shipperCall.ResponseHeadersAsync;

foreach (Metadata.Entry entry in metadata)
{
  // Исключительно для удобства работы с кодом
  _logger.LogCritical($"Key: {entry.Key}, Value: {entry.Value}");
}

ShipperReply shipperReply = await shipperCall.ResponseAsync;

model.ShipperSummary = "Shipper from gRPC service: " +
  $"ID: {shipperReply.ShipperId}, Name: {shipperReply.CompanyName},"
  + $" Phone: {shipperReply.Phone}.";
```

3. Запустите проект `Northwind.Grpc.Service` без отладки.
4. Запустите проект `Northwind.Grpc.Client.Mvc`.
5. При необходимости запустите браузер и перейдите на главную страницу по адресу `https://localhost:5133/`.
6. Убедитесь, что клиент выполняет `POST`-запросы к gRPC-сервисам `Greeter` и `Shipper` и выводятся две записи метаданных с красной меткой «критично», касающиеся вызова метода `GetShipper` с ключами `date` и `server` (рис. 13.7).
7. Закройте браузер и завершите работу веб-сервера.

```
C:\apps-services-net8\Chapter13\Northwind.Grpc.Client.Mvc\bin\Debug\net8.0\Northwind.Grpc.Client.Mvc.exe
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\apps-services-net8\Chapter13\Northwind.Grpc.Client.Mvc
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[100]
      Start processing HTTP request POST https://localhost:5131/greet.Greeter/SayHello
info: System.Net.Http.HttpClient.Greeter.ClientHandler[100]
      Sending HTTP request POST https://localhost:5131/greet.Greeter/SayHello
info: System.Net.Http.HttpClient.Greeter.ClientHandler[101]
      Received HTTP response headers after 249.2595ms - 200
info: System.Net.Http.HttpClient.Greeter.LogicalHandler[101]
      End processing HTTP request after 261.7622ms - 200
info: System.Net.Http.HttpClient.Shipper.LogicalHandler[100]
      Start processing HTTP request POST https://localhost:5131/shipper.Shipper/GetShipper
info: System.Net.Http.HttpClient.Shipper.ClientHandler[100]
      Sending HTTP request POST https://localhost:5131/shipper.Shipper/GetShipper
info: System.Net.Http.HttpClient.Shipper.ClientHandler[101]
      Received HTTP response headers after 1278.3375ms - 200
info: System.Net.Http.HttpClient.Shipper.LogicalHandler[101]
      End processing HTTP request after 1278.4549ms - 200
crit: Northwind.Grpc.Client.Mvc.Controllers.HomeController[0]
      Key: date, Value: Tue, 13 Jun 2023 13:57:22 GMT
crit: Northwind.Grpc.Client.Mvc.Controllers.HomeController[0]
      Key: server, Value: Kestrel
```

Рис. 13.7. Регистрация метаданных при вызове gRPC

Эквивалентом свойства ResponseHeadersAsync для трейлеров служит метод GetTrailers, который возвращает объект Metadata со словарем трейлеров. Трейлеры доступны в конце вызова.

Ограничение срока выполнения в рамках повышения надежности

Настройка срока выполнения вызова gRPC — важный прием, позволяющий ограничивать время, отведенное на выполнение запроса. Он помогает избежать чрезмерного расхода ресурсов сервера gRPC-сервисами.

Значение срока выполнения передается сервису, чтобы тот мог прервать выполнение операции по истечении указанного времени, а не продолжать ее бесконечно. Несмотря на установленное время обработки запроса сервером, клиент может прервать операцию раньше, до получения ответа, из-за задержек на уровне сети.

Рассмотрим пример.

1. В папке `Services` проекта `Northwind.Grpc.Service` откройте файл `ShipperService.cs`. В методе `GetShipper` добавьте код для регистрации срока выполнения и добавления паузы 5 секунд:

```
public override async Task<ShipperReply> GetShipper(
  ShipperRequest request, ServerCallContext context)
{
  _logger.LogCritical($"This request has a deadline of {
    context.Deadline:T}. It is now {DateTime.UtcNow:T}.");

  await Task.Delay(TimeSpan.FromSeconds(5));
  ...
}
```

2. В файле `appsettings.Development.json` проекта `Northwind.Grpc.Service` для ASP.NET Core укажите уровень журналирования `Information`:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Information"
    }
  }
}
```

3. В папке `Controllers` проекта `Northwind.Grpc.Client.Mvc` откройте файл `HomeController.cs`. В методе `Index` настройте срок выполнения равным 3 секундам при вызове метода `GetShipperAsync`:

```
AsyncUnaryCall<ShipperReply> shipperCall = shipperClient.GetShipperAsync(
  new ShipperRequest { ShipperId = id },
  // Срок выполнения должен быть в формате UTC DateTime
  deadline: DateTime.UtcNow.AddSeconds(3));
```

4. В этом же методе перед блоком `catch` добавьте обработку исключения `RpcException`, если код ошибки указывает на превышение срока выполнения:

```
catch (RpcException rpcex) when (rpcex.StatusCode ==
  global::Grpc.Core.StatusCode.DeadlineExceeded)
{
  _logger.LogWarning("Northwind.Grpc.Service deadline exceeded.");

  model.ErrorMessage = rpcex.Message;
}
catch (Exception ex)
{
  _logger.LogWarning($"Northwind.Grpc.Service is not responding.");
  model.ErrorMessage = ex.Message;
}
```

5. В файле `appsettings.Development.json` проекта `Northwind.Grpc.Client.Mvc` для ASP.NET Core укажите уровень журналирования `Information`:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Information"
    }
  }
}
```

6. Запустите проект `Northwind.Grpc.Service` без отладки.
7. Запустите проект `Northwind.Grpc.Client.Mvc`.
8. Если браузер не открылся автоматически, запустите его и перейдите на главную страницу сервиса по адресу `https://localhost:5133/`.
9. В терминале gRPC-сервиса убедитесь, что для запроса задан трехсекундный срок выполнения:

```
crit: Northwind.Grpc.Service.Services.ShipperService[0]
      This request has a deadline of 14:56:30. It is now 14:56:27.
```

10. В браузере через 3 секунды должно появиться сообщение о превышении срока выполнения (рис. 13.8).

Рис. 13.8. Превышен срок выполнения

11. В терминале клиента MVC ASP.NET Core изучите журналы, фиксирующие момент превышения срока выполнения запроса к методу `GetShipper` в gRPC-сервисе:

```
info: System.Net.Http.HttpClient.Shipper.LogicalHandler[100]
      Start processing HTTP request POST https://localhost:5131/
      shipper.Shipper/GetShipper
info: System.Net.Http.HttpClient.Shipper.ClientHandler[100]
      Sending HTTP request POST https://localhost:5131/shipper.Shipper/
      GetShipper
warn: Grpc.Net.Client.Internal.GrpcCall[7]
      gRPC call deadline exceeded.
info: Grpc.Net.Client.Internal.GrpcCall[3]
      Call failed with gRPC error status. Status code: 'DeadlineExceeded',
      Message: ''.
```

12. Закройте браузер и завершите работу веб-сервера.
13. В файле `ShipperService.cs` закомментируйте код, вызывающий задержку на 5 секунд:

```
// await Task.Delay(TimeSpan.FromSeconds(5));
```

Дельный совет: по умолчанию срок выполнения не ограничивается, поэтому его всегда следует настраивать на стороне клиента. В сервисе используйте переданное значение, чтобы автоматически прерывать выполнение запроса при его истечении. Также передавайте токен отмены в асинхронные вызовы, чтобы своевременно освобождать ресурсы на сервере.

Обработка дат, времени и десятичных чисел

В gRPC нет встроенных типов для работы с датами, временем и десятичными числами. Для их хранения применяют расширения типа `google.protobuf.Timestamp` (эквивалент `DateTimeOffset`) и `google.protobuf.Duration` (эквивалент `TimeSpan`). Перед использованием в качестве типов полей в сообщении их нужно импортировать:

```
syntax = "proto3";

import "google/protobuf/duration.proto";
import "google/protobuf/timestamp.proto";

message Employee {
  int32 employeeId = 1;
  google.protobuf.Timestamp birth_date = 2;
  google.protobuf.Duration earned_vacation_time = 3;
  ...
}
```

.NET-классы не используют эти типы напрямую — вместо них применяются промежуточные объекты:

```
public class Employee
{
  public int EmployeeId;
  public Timestamp BirthDate;
  public Duration EarnedVacationTime;
}
```

Для преобразования существуют методы `FromDateTimeOffset`, `ToDateTimeOffset`, `FromTimeSpan` и `ToTimeSpan`:

```
Employee employee = new()
{
  EmployeeId = 1,
```

```
  BirthDate = Timestamp.FromDateTimeOffset(new DateTimeOffset(
    year: 1998, month: 11, day: 30, hour: 0, minute: 0, second: 0,
    offset: TimeSpan.FromHours(-5)),
  EarnedVacationTime = Duration.FromTimeSpan(TimeSpan.FromDays(15))
};

DateTimeOffset when = employee.BirthDate.ToDateTimeOffset();
TimeSpan daysoff = employee.EarnedVacationTime.ToTimeSpan();
```

Десятичные числа с высокой точностью (`decimal`) в gRPC тоже не поддерживаются. Хотя в будущем такая возможность может появиться, пока требуется создавать собственные сообщения. При этом разработчикам на других платформах придется реализовывать обработку этих данных вручную.

Определение пользовательского типа для десятичных чисел, обработка времени и дат

Добавим gRPC-сервисы для управления данными о товарах (свойство `UnitPrice` представлено десятичным числом) и сотрудниках (свойство HireDate представлено значением `DateTime`).

1. В папке `Protos` проекта `Northwind.Grpc.Service` создайте файл `decimal.proto` и определите в нем формат хранения десятичных значений:

```
syntax = "proto3";

option csharp_namespace = "Northwind.Grpc.Service";

package decimal;

// Представление десятичного числа в виде целых единиц и наносекунд
// Пример: 12345.6789 -> { units = 12345, nanos = 678900000 }

message DecimalValue {

    // Целая часть числа
    int64 units = 1;

    // Дробная часть в наносекундах (10^-9)
    // Значение должно иметь тот же знак, что и целая часть
    sfixed32 nanos = 2;
}
```

2. Создайте файл `product.proto` и добавьте в него определения сообщений и методов для получения списка товаров, одного товара и товаров с минимальной ценой:

```
syntax = "proto3";

option csharp_namespace = "Northwind.Grpc.Service";
```

```
import "Protos/decimal.proto";

package product;

service Product {
  rpc GetProduct (ProductRequest) returns (ProductReply);
  rpc GetProducts (ProductsRequest) returns (ProductsReply);
  rpc GetProductsMinimumPrice (ProductsMinimumPriceRequest)
      returns (ProductsReply);
}

message ProductRequest {
  int32 product_id = 1;
}

message ProductsRequest {
}

message ProductsMinimumPriceRequest {
  decimal.DecimalValue minimum_price = 1;
}

message ProductReply {
  int32 product_id = 1;
  string product_name = 2;
  int32 supplier_id = 3;
  int32 category_id = 4;
  string quantity_per_unit = 5;
  decimal.DecimalValue unit_price = 6;
  int32 units_in_stock = 7;
  int32 units_on_order = 8;
  int32 reorder_level = 9;
  bool discontinued = 10;
}

message ProductsReply {
  repeated ProductReply products = 1;
}
```

3. Создайте файл `employee.proto` и добавьте в него определения сообщений и служебных методов для работы с данными сотрудников. Не забудьте импортировать расширение Google `timestamp.proto` для работы с датами:

```
syntax = "proto3";

option csharp_namespace = "Northwind.Grpc.Service";

import "google/protobuf/duration.proto";
import "google/protobuf/timestamp.proto";

package employee;
```

```
service Employee {
  rpc GetEmployee (EmployeeRequest) returns (EmployeeReply);
  rpc GetEmployees (EmployeesRequest) returns (EmployeesReply);
}

message EmployeeRequest {
  int32 employee_id = 1;
}

message EmployeesRequest {
}

message EmployeeReply {
  int32 employee_id = 1;
  string last_name = 2;
  string first_name = 3;
  string title = 4;
  string title_of_courtesy = 5;
  google.protobuf.Timestamp birth_date = 6;
  google.protobuf.Timestamp hire_date = 7;
  string address = 8;
  string city = 9;
  string region = 10;
  string postal_code = 11;
  string country = 12;
  string home_phone = 13;
  string extension = 14;
  bytes photo = 15;
  string notes = 16;
  int32 reports_to = 17;
  string photo_path = 18;
}

message EmployeesReply {
  repeated EmployeeReply employees = 1;
}
```

4. В файле проекта укажите, что gRPC должен обрабатывать новые `.proto`-файлы:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
  <Protobuf Include="Protos\shipper.proto" GrpcServices="Server" />
  <Protobuf Include="Protos\decimal.proto" GrpcServices="Server" />
  <Protobuf Include="Protos\product.proto" GrpcServices="Server" />
  <Protobuf Include="Protos\employee.proto" GrpcServices="Server" />
</ItemGroup>
```

5. Снова выполните сборку проекта, чтобы gRPC сгенерировал C#-классы в `obj\Debug\net8.0\Protos` (рис. 13.9).
6. В папке проекта `Northwind.Grpc.Service` создайте каталог `Converters`.

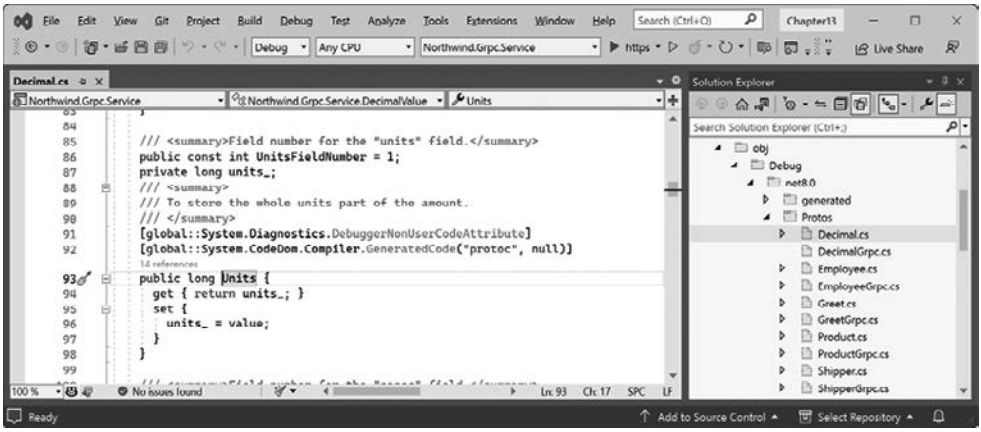


Рис. 13.9. Классы, сгенерированные инструментом gRPC, для пользовательского десятичного типа со свойством Units

7. В этой папке создайте файл класса `DecimalValue.Converters.cs`, откройте его и добавьте в разделяемый класс, созданный средствами gRPC, конструктор и пару операторов для преобразования между пользовательским типом `DecimalValue` и встроенным `decimal`:

```
namespace Northwind.Grpc.Service;

// Для слияния с типом DecimalValue, сгенерированным
// инструментом gRPC в файле obj\Debug\net8.0\Protos\Decimal.cs

public partial class DecimalValue
{
  private const decimal NanoFactor = 1_000_000_000;

  public DecimalValue(long units, int nanos)
  {
    Units = units;
    Nanos = nanos;
  }

public static implicit operator decimal(DecimalValue grpcDecimal)
{
  return grpcDecimal.Units + (grpcDecimal.Nanos / NanoFactor);
}

  public static implicit operator DecimalValue(decimal value)
  {
    long units = decimal.ToInt64(value);
    int nanos = decimal.ToInt32((value - units) * NanoFactor);
    return new DecimalValue(units, nanos);
  }
}
```

Реализация gRPC-сервисов для работы с товарами и сотрудниками

Теперь необходимо реализовать и зарегистрировать сервисы.

1. В папке `Services` проекта `Northwind.Grpc.Service` создайте файл класса `ProductService.cs` и реализуйте в нем сервис для работы с товарами. Пусть это будет дополнительное самостоятельное упражнение. Вы также можете скопировать код со страницы по адресу https://github.com/markjprice/apps-services-net8/blob/main/code/Chapter13/Northwind.Grpc.Service/Services/ProductService.cs.
2. В той же папке создайте еще один файл класса, `EmployeeService.cs`, и реализуйте в нем сервис для работы с сотрудниками. Попробуйте также выполнить это задание самостоятельно или скопируйте код со страницы по адресу https://github.com/markjprice/apps-services-net8/blob/main/code/Chapter13/Northwind.Grpc.Service/Services/EmployeeService.cs.
3. Зарегистрируйте созданные сервисы в файле `Program.cs`:

```
app.MapGrpcService<ProductService>();
app.MapGrpcService<EmployeeService>();
```

Создание gRPC-клиентов для вызова сервисов

Далее нужно в проект MVC добавить клиенты для вызова созданных gRPC-сервисов.

1. В проекте `Northwind.Grpc.Client.Mvc` скопируйте три `.proto`-файла из проекта сервиса в папку `Protos` проекта MVC.
2. Измените в этих файлах пространство имен:

```
option csharp_namespace = "Northwind.Grpc.Client.Mvc";
```

3. В файле проекта зарегистрируйте `.proto`-файлы для генерации клиентских классов:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
  <Protobuf Include="Protos\shipper.proto" GrpcServices="Client" />
  <Protobuf Include="Protos\decimal.proto" GrpcServices="Client" />
  <Protobuf Include="Protos\employee.proto" GrpcServices="Client" />
  <Protobuf Include="Protos\product.proto" GrpcServices="Client" />
</ItemGroup>
```

В некоторых редакторах кода (например, JetBrains Rider) могут появляться дополнительные настройки. Лучше упростить элементы, как показано выше, иначе возможны ошибки.

4. Скопируйте папку `Converters` из gRPC-сервиса в проект MVC.
5. В этой папке откройте файл `DecimalValue.Converters.cs` и измените в нем пространство имен, чтобы использовать класс клиента:

```
namespace Northwind.Grpc.Client.Mvc;
```

6. В файле `Program.cs` проекта `Northwind.Grpc.Client.Mvc` зарегистрируйте gRPC-клиенты для новых сервисов:

```
builder.Services.AddGrpcClient<Product.ProductClient>("Product", options =>
  {
    options.Address = new Uri("https://localhost:5131");
  });

builder.Services.AddGrpcClient<Employee.EmployeeClient>("Employee", options =>
  {
    options.Address = new Uri("https://localhost:5131");
  });
```

7. В файле `HomeController.cs` (папка `Controllers`) добавьте поля для обоих клиентов и инициализируйте их в конструкторе. (Подсказка: по аналогии с классами `greeter` и `shipper`.)
8. В этом же файле реализуйте методы действий для получения товаров и сотрудников:

```
public async Task<IActionResult> Products(decimal minimumPrice = 0M)
{
  ProductsReply reply = await _productClient.GetProductsMinimumPriceAsync(
    new ProductsMinimumPriceRequest() { MinimumPrice = minimumPrice });

  return View(reply.Products);
}

public async Task<IActionResult> Employees()
{
  EmployeesReply reply = await _employeeClient.GetEmployeesAsync(
    new EmployeesRequest());

  return View(reply.Employees);
}
```

9. Откройте папку `Views\Shared`, а затем файл `_Layout.cshtml`. Ниже ссылки на главную страницу добавьте ссылки для перехода к товарам и сотрудникам:

```
<li class="nav-item">
  <a class="nav-link text-dark" asp-area="" asp-controller="Home"
    aspaction="Index">Home</a>
</li>
```

```
<li class="nav-item">
  <a class="nav-link text-dark" asp-area="" asp-controller="Home"
    aspaction="Products">Products</a>
</li>
<li class="nav-item">
  <a class="nav-link text-dark" asp-area="" asp-controller="Home"
    aspaction="Employees">Employees</a>
</li>
```

10. В папке `Views\Home` создайте файл представления Razor и присвойте ему имя `Products.cshtml`. Настройте в нем таблицу товаров:

```
@using Google.Protobuf.Collections
@using Northwind.Grpc.Client.Mvc
@model RepeatedField<ProductReply>
@{
  ViewData["Title"] = "Products";
  decimal price = 0;
}
<h1>@ViewData["Title"]</h1>
<form asp-action="Products" method="post">
  <input name="minimumPrice" placeholder="Enter a minimum unit price" />
  <input type="submit" value="Filter Products" />
</form>
<table class="table table-primary table-bordered">
  <thead>
    <tr>
      <th>Product ID</th>
      <th>Product Name</th>
      <th>Unit Price</th>
      <th>Units In Stock</th>
      <th>Units On Order</th>
      <th>Reorder Level</th>
      <th>Discontinued</th>
    </tr>
  </thead>
  <tbody>
    @foreach (ProductReply p in Model)
    {
      <tr>
        <td>@p.ProductId</td>
        <td>@p.ProductName</td>
        @{ price = p.UnitPrice; }
        <td>@price.ToString("C")</td>
        <td>@p.UnitsInStock</td>
        <td>@p.UnitsOnOrder</td>
        <td>@p.ReorderLevel</td>
        <td>@p.Discontinued</td>
      </tr>
    }
  </tbody>
</table>
```

11. В той же папке создайте еще один файл представления Razor, на этот раз `Employees.cshtml`. Он будет содержать таблицу сотрудников:

```
@using Google.Protobuf.Collections
@using Northwind.Grpc.Client.Mvc
@model RepeatedField<EmployeeReply>
@{
  ViewData["Title"] = "Employees";
}
<h1>@ViewData["Title"]</h1>
<table class="table table-primary table-bordered">
  <thead>
    <tr>
      <th>Employee ID</th>
      <th>Full Name</th>
      <th>Job Title</th>
      <th>Address</th>
      <th>Birth Date</th>
      <th>Photo</th>
    </tr>
  </thead>
  <tbody>
    @foreach (EmployeeReply e in Model)
    {
      <tr>
        <td>@e.EmployeeId</td>
        <td>@e.TitleOfCourtesy @e.FirstName @e.LastName</td>
        <td>@e.Title</td>
        <td>@e.Address<br />@e.City<br />@e.Region<br />
          @e.PostalCode<br />@e.Country</td>
        <td>@e.BirthDate.ToDateTimeOffset().ToString("D")</td>
        <td><img src="
          @Convert.ToBase64String(e.Photo.ToByteArray())" /></td>
      </tr>
    }
  </tbody>
</table>
```

Проверка обработки десятичных дробей, дат и байтов

После завершения всех настроек протестируйте реализацию.

1. Запустите проект `Northwind.Grpc.Service` без отладки.
2. Запустите проект `Northwind.Grpc.Client.Mvc`.
3. В верхней части страницы щелкните на ссылке Products и изучите отобразившуюся таблицу с товарами (рис. 13.10).

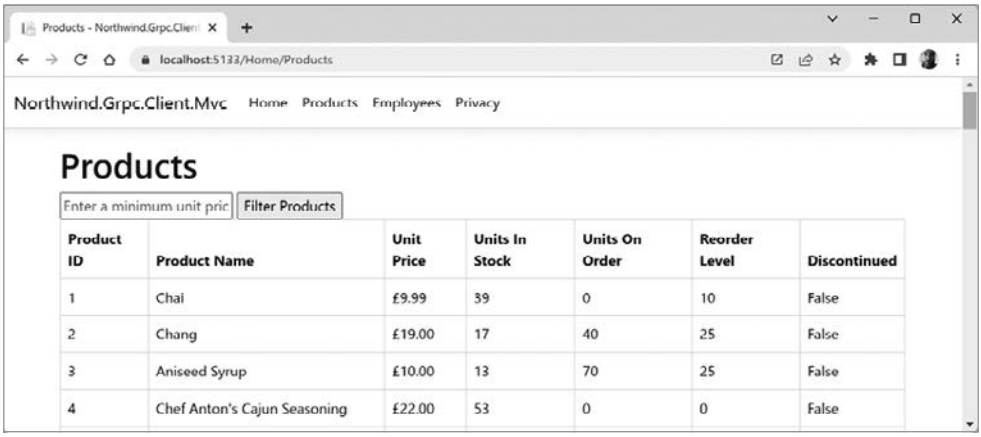


Рис. 13.10. Сведения о товарах и их стоимости, полученные с помощью нашего сервиса

4. Введите минимальную цену товара, например `100`, и нажмите кнопку Filter Products (Отфильтровать товары). В таблице останутся только товары ценой `100` и выше.
5. В верхней части страницы щелкните кнопкой мыши на ссылке Employees — в таблице появятся сведения о сотрудниках, включая даты их рождения и фотографии (рис. 13.11).
6. Закройте браузер и завершите работу веб-сервера.

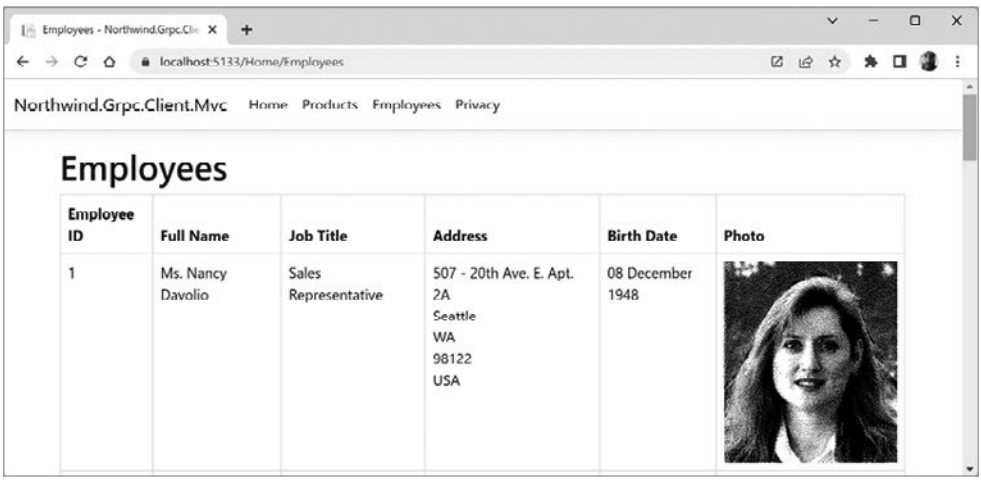


Рис. 13.11. Данные сотрудников, включая даты рождения и фотографии, полученные с помощью нашего сервиса

Теперь вы знаете, как воспользоваться gRPC для создания нескольких сервисов, обрабатывающих данные. Далее рассмотрим более продвинутые возможности gRPC.

Перехватчики gRPC и обработка ошибок

Перехватчики в gRPC позволяют выполнять дополнительную обработку запросов и ответов на стороне клиента или сервиса. Они широко применяются для журналирования, мониторинга и проверки данных.

Добавление перехватчика на стороне клиента

Добавим перехватчик gRPC на стороне клиента для ведения журнала.

1. В проекте `Northwind.Grpc.Client.Mvc` создайте папку `Interceptors`.
2. В этой папке создайте файл класса `ClientLoggingInterceptor.cs` и добавьте в него код для определения перехватчика:

```
using Grpc.Core.Interceptors; // Для типа Interceptor и т. д.
using Grpc.Core; // Для обьекта AsyncUnaryCall<T>

namespace Northwind.Grpc.Client.Mvc.Interceptors;

public class ClientLoggingInterceptor : Interceptor
{
  private readonly ILogger _logger;

  public ClientLoggingInterceptor(ILoggerFactory loggerFactory)
  {
    _logger = loggerFactory.CreateLogger<ClientLoggingInterceptor>();
  }

  public override AsyncUnaryCall<TResponse>
    AsyncUnaryCall<TRequest, TResponse>(TRequest request,
    ClientInterceptorContext<TRequest, TResponse> context,
    AsyncUnaryCallContinuation<TRequest, TResponse> continuation)
  {
    _logger.LogWarning("Starting call. Type: {0}. Method: {1}.",
      context.Method.Type, context.Method.Name);

    return continuation(request, context);
  }
}
```

Перехватчики образуют конвейер. В перехватчике необходимо вызвать следующий перехватчик в цепочке (представленный делегатом continuation), передав ему запрос и контекст.

3. В файле `Program.cs` проекта `Northwind.Grpc.Client.Mvc` перед регистрацией gRPC-клиента добавьте перехватчик в качестве сервиса-одиночки (singleton):

```
// Зарегистрируйте перехватчик, прежде чем прикрепить его к клиенту gRPC
builder.Services.AddSingleton<ClientLoggingInterceptor>();
```

4. В конце выражения регистрации клиента товаров добавьте код для подключения перехватчика:

```
builder.Services.AddGrpcClient<Product.ProductClient>("Product", options =>
  {
    options.Address = new Uri("https://localhost:5131");
  })
  .AddInterceptor<ClientLoggingInterceptor>();
```

Перехватчик журналирования можно подключить к любому количеству клиентов.

5. Запустите проект `Northwind.Grpc.Service` без отладки.
6. Запустите проект `Northwind.Grpc.Client.Mvc`.
7. На главной странице щелкните кнопкой мыши на ссылке Products.
8. В терминале веб-приложения появится предупреждение, отличающееся от информационных сообщений (по умолчанию оно желто-черное):

```
warn: Northwind.Grpc.Client.Mvc.Interceptors.ClientLoggingInterceptor[0]
      Starting call. Type: Unary. Method: GetProductsMinimumPrice.
```

9. Закройте браузер и завершите работу веб-сервера.

В копилку знаний: если у вас возникли подозрения, что перехватчики весьма похожи на ПО промежуточного слоя ASP.NET Core, рекомендую прочитать интересное сравнение на странице по адресу https://learn.microsoft.com/ru-ru/aspnet/core/grpc/interceptors#grpc-interceptors-versus-middleware.

Обработка исключений и временных сбоев

В gRPC встроена поддержка автоматического повторения неудачных вызовов, что помогает справляться с временными сбоями, например с отключением сети или перегруженностью сервиса.

При ошибках на стороне клиента выбрасывается исключение `RpcException`, содержащее подробную информацию об ошибке.

Давайте симулируем ошибку на gRPC-сервисе и посмотрим, как клиент ее обрабатывает.

1. В файле `GreeterService.cs` (папка `Services`, проект `Northwind.Grpc.Service`) измените метод `SayHello`, добавив задержку в одну секунду и вероятность в 2/3 случаев выбросить исключение `UNAVAILABLE` и в 1/3 случаев выполнить задачу успешно:

```
public override async Task<HelloReply> SayHello(
  HelloRequest request, ServerCallContext context)
{
  await Task.Delay(1000);

  if (Random.Shared.Next(1, 4) == 1)
  {
    return new HelloReply
    {
      Message = "Hello " + request.Name
    };
  }
  else
  {
    throw new RpcException(new Status(StatusCode.Unavailable,
      "Service is temporarily unavailable. Try again later."));
  }
}
```

2. Запустите проект `Northwind.Grpc.Service` без отладки.
3. Запустите проект `Northwind.Grpc.Client.Mvc`.
4. На главной странице отобразится исключение (рис. 13.12).

Рис. 13.12. Исключение «Сервис недоступен»

5. Если исключение не появилось, обновите страницу.
6. Закройте браузер и завершите работу веб-сервера.

Теперь выполним обработку временных сбоев на клиенте.

1. В файле `Program.cs` проекта `Northwind.Grpc.Client.Mvc` перед регистрацией клиента `greeter` в коллекции сервисов добавьте политику повторных попыток (`MethodConfig`). Она настроит до пяти повторов при ошибке `UNAVAILABLE`. После настройки адреса клиента приветствия примените метод `config`:

```
MethodConfig configForAllMethods = new()
{
  Names = { MethodName.Default },
  RetryPolicy = new RetryPolicy
  {
    MaxAttempts = 5,
    InitialBackoff = TimeSpan.FromSeconds(1),
    MaxBackoff = TimeSpan.FromSeconds(5),
    BackoffMultiplier = 1.5,
    RetryableStatusCodes = { StatusCode.Unavailable }
  }
};

builder.Services.AddGrpcClient<Greeter.GreeterClient>("Greeter", options =>
  {
    options.Address = new Uri("https://localhost:5131");
  })
  .ConfigureChannel(channel =>
  {
    channel.ServiceConfig = new ServiceConfig
    {
      MethodConfigs = { configForAllMethods }
    };
  });
```

2. Запустите проект `Northwind.Grpc.Service` без отладки.
3. Запустите проект `Northwind.Grpc.Client.Mvc`.
4. На появление главной страницы может уйти несколько секунд, но в конце концов она успешно загрузится с сообщением Hello Henrietta gRPC-сервиса. В выводе будут указаны попытки вызвать сообщение `SayHello`:

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:5131/greet.Greeter/
      SayHello - application/grpc -
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /greet.Greeter/SayHello'
info: Grpc.AspNetCore.Server.ServerCallHandler[7]
      Error status code 'Unavailable' with detail 'Service is temporarily
      unavailable. Try again later.' raised.
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/2 POST https://localhost:5131/greet.Greeter/
      SayHello - 200 0 application/grpc 1039.4626ms
```

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:5131/greet.Greeter/
      SayHello - application/grpc -
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /greet.Greeter/SayHello'
info: Grpc.AspNetCore.Server.ServerCallHandler[7]
      Error status code 'Unavailable' with detail 'Service is temporarily
      unavailable. Try again later.' raised.
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/2 POST https://localhost:5131/greet.Greeter/
      SayHello - 200 0 application/grpc 1008.1375ms
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:5131/greet.Greeter/
      SayHello - application/grpc -
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/2 POST https://localhost:5131/greet.Greeter/
      SayHello - 200 - application/grpc 1016.4590ms
```

Транскодирование JSON в gRPC

JSON — самый распространенный формат передачи данных в браузер или на мобильные устройства. Было бы удобно, если бы gRPC-сервис можно было вызывать не только через HTTP/2, но и с помощью JSON.

На этот случай в компании Microsoft есть решение — *транскодирование JSON в gRPC*. Это расширение ASP.NET Core автоматически создает конечные точки HTTP с поддержкой JSON для gRPC-сервисов. В нем используется класс Google `HttpRule` для преобразования HTTP-запросов в gRPC-вызовы.

В копилку знаний: дополнительная информация о классе Google HttpRule приведена по адресу https://cloud.google.com/dotnet/docs/reference/Google.Api.CommonProtos/latest/Google.Api.HttpRule.

Включение транскодирования JSON

Разберемся, как включить транскодирование JSON в нашем gRPC-сервисе.

1. В файле проекта `Northwind.Grpc.Service` подключите пакет для транскодирования JSON в gRPC:

```
<ItemGroup>
  <PackageReference Include="Grpc.AspNetCore" Version="2.59.0" />
  <PackageReference Include="Microsoft.Data.SqlClient" Version="5.1.2" />
  <PackageReference Include="Microsoft.AspNetCore.Grpc.JsonTranscoding"
                    Version="8.0.0" />
</ItemGroup>
```

2. Выполните сборку проекта `Northwind.Grpc.Service` для восстановления зависимостей.
3. В файле `appsettings.json` измените параметр `Protocols`, добавив поддержку HTTP/1.1 и HTTP/2:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1AndHttp2"
    }
  }
}
```

По умолчанию gRPC работает только с HTTP/2-запросами. Чтобы поддерживать клиенты, такие как .http-файлы в редакторе кода и среда Unity, необходимо включить поддержку HTTP/1.1 и HTTP/2.

При использовании на одном порте обоих протоколов необходим протокол TLS для согласования, поэтому протокол HTTPS в gRPC-сервисе рекомендуется оставить включенным. По этой же причине не следует использовать метод CreateSlimBuilder.

4. В файле `Program.cs` после регистрации gRPC добавьте вызов метода для добавления транскодирования JSON:

```
builder.Services.AddGrpc().AddJsonTranscoding();
```

5. Откройте проект `Northwind.Grpc.Service` и добавьте папку `google`.
6. Вложите в нее каталог `api`.
7. В этом каталоге создайте файлы `http.proto` и `annotations.proto`.
8. Скопируйте в них содержимое из файлов, опубликованных по адресу https://github.com/dotnet/aspnetcore/tree/main/src/Grpc/JsonTranscoding/test/testassets/Sandbox/google/api.

9. В папке `Protos` откройте файл `employee.proto` и добавьте в него оператор импорта файла `annotations.proto`. Затем с помощью этого импорта добавьте параметр, который будет указывать конечную точку для выполнения HTTP-запроса к методу `GetEmployee`:

```
syntax = "proto3";

option csharp_namespace = "Northwind.Grpc.Service";

import "google/protobuf/duration.proto";
import "google/protobuf/timestamp.proto";
import "google/api/annotations.proto";

package employee;

service Employee {
  rpc GetEmployee (EmployeeRequest) returns (EmployeeReply) {
    option (google.api.http) = {
      get: "/v1/employee/{employee_id}"
    };
  };
  rpc GetEmployees (EmployeesRequest) returns (EmployeesReply);
}
```

Проверка транскодирования JSON

Теперь мы можем запустить gRPC-сервис и вызывать его из любого браузера.

1. Запустите проект `Northwind.Grpc.Service`.
2. Запустите любой браузер, откройте инструменты разработчика и перейдите на вкладку Network (Сеть) для просмотра сетевого трафика.
3. Перейдите по URL-адресу `https://localhost:5131/v1/employee/1`, отправив тем самым `GET`-запрос, вызывающий метод `GetEmployee`. Убедитесь, что gRPC-сервис возвращает JSON-ответ (рис. 13.13).
4. В папке `HttpRequests` создайте файл `grpc-json-transcoding.http` и добавьте в него запросы сотрудников по протоколу HTTP/1.1:

```
### Настройка переменной для базового адреса gRPC-сервиса
@base_address = https://localhost:5131/

### Извлечение данных Nancy Davolio
GET {{base_address}}v1/employee/1

### Извлечение данных Andrew Fuller Davolio
GET {{base_address}}v1/employee/2
```

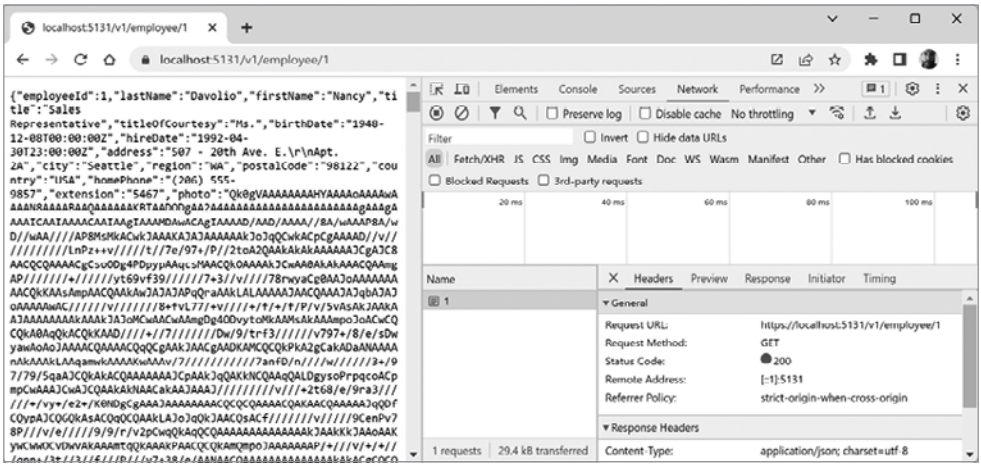


Рис. 13.13. Процесс выполнения HTTP 1.1 GET-запроса к gRPC-сервису с получением ответа в формате JSON

5. Отправьте оба запроса и убедитесь, что сервис корректно отвечает. Проанализируйте терминал gRPC-сервиса и удостоверьтесь, что запросы были сделаны по протоколу HTTP/1.1:

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:5131/v1/employee/2
- - -
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /v1/employee/{employee_id}'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /v1/employee/{employee_id}'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/1.1 GET https://localhost:5131/v1/employee/2
- 200 - application/json;+charset=utf-8 7.9328ms
```

6. Закройте `.http`-файл, браузер и завершите работу веб-сервера.

Сравнение с gRPC-Web

Использование *gRPC-Web* — это альтернатива транскодированию JSON в gRPC, позволяющая вызывать gRPC-сервисы непосредственно из браузера с выполнением в нем gRPC-клиента. Преимущество этого подхода в том, что при обмене данными между браузером и gRPC-сервисом используется протокол `Protobuf`, обеспечивающий все преимущества производительности и масштабируемости, характерные для истинных gRPC-коммуникаций.

Транскодирование JSON в gRPC позволяет браузерам вызывать gRPC-сервисы как обычные HTTP-API с поддержкой JSON. При этом браузер не нужно информировать о gRPC, а сервис отвечает за преобразование HTTP-вызовов в реальные вызовы gRPC-сервиса.

Вкратце:

- транскодирование JSON в gRPC выполняется на стороне сервера;
- функции gRPC-Web выполняются на стороне клиента.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 13.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. Благодаря каким трем преимуществам gRPC прекрасно подходит для реализации сервисов?
2. Как в gRPC определяются контракты?
3. Каким типам .NET (`int`, `double`, `DateTime`) требуются расширения для импорта?
4. Почему при вызове метода gRPC важно ограничивать срок выполнения?
5. В чем заключаются преимущества транскодирования JSON для gRPC-сервиса, размещенного в ASP. NET Core?

Упражнение 13.2. Сравнение сервисов gRPC с API HTTP

Рекомендую прочитать статью, размещенную по адресу https://learn.microsoft.com/ru-ru/aspnet/core/grpc/comparison.

Упражнение 13.3. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-13---building-efficient-microservices-using-grpc.

Резюме

В этой главе вы:

- познакомились с gRPC-сервисами, узнали об их работе и преимуществах;
- реализовали простой gRPC-сервис;
- создали gRPC-сервис с использованием модели EF Core, которая пока не поддерживает публикацию AOT;
- запустили gRPC-сервис с библиотеками SqlClient, которые поддерживают публикацию AOT, благодаря чему его размер меньше, а скорость работы выше;
- научились ограничивать сроки выполнения и читать метаданные, отправленные в виде заголовков и трейлеров;
- реализовали пользовательский тип `decimal` и использовали расширенные типы даты/времени;
- реализовали перехватчик на стороне клиента;
- дополнили gRPC-сервис, добавив поддержку вызова через HTTP с JSON для клиентов, не поддерживающих нативную работу с gRPC;

В следующей главе вы познакомитесь с процессом создания пользовательских интерфейсов сайтов с применением MVC в ASP.NET Core.

14 Создание пользовательских веб-интерфейсов с помощью платформы ASP.NET Core

Эта глава посвящена разработке пользовательских веб-интерфейсов с применением платформы ASP.NET Core. Вы познакомитесь с представлениями MVC, синтаксисом Razor, вспомогательными сервисами HTML и тег-хелперами, а также узнаете, как реализовать интернационализацию сайта. Дополнительно мы рассмотрим, как воспользоваться фреймворком Bootstrap для быстрого создания прототипов пользовательского интерфейса.

Настройка сайта MVC ASP.NET Core

Шаблон проектирования *Model-View-Controller (MVC)* четко разделяет следующие технические понятия.

- *Модели* — классы, соответствующие данным и моделям представлений, используемые в веб-приложении.
- *Представления* — файлы Razor с расширением `.cshtml`, преобразующие данные моделей в HTML-страницы. Важно не путать их с файлами Blazor, которые имеют расширение `.razor`!
- *Контроллеры* — классы, обрабатывающие HTTP-запросы на веб-сервер. Методы контроллеров создают модели представлений, содержащие модели сущностей, передают их в представления и формируют HTTP-ответы для браузера или другого клиента.

В ASP.NET Core используется несколько типов файлов Razor, что может привести к путанице. В табл. 14.1 приведены основные различия.

Внимание! Следует быть внимательными при выборе правильного расширения файла и использования соответствующей директивы в начале.

Таблица 14.1. Ключевые сходства и различия между файлами Razor

Технология	Специальное имя файла	Расширение файла	Директива
Компонент Razor (для Blazor)		`.razor`	
Компонент Razor (для Blazor с маршрутизацией страниц)		`.razor`	`@page`
Страница Razor		`.cshtml`	`@page`
Представление Razor (для MVC)		`.cshtml`	
Макет Razor	`_{customname}`	`.cshtml`	
Представление Razor (частичное)	`_{customname}`	`.cshtml`	
Начальный код представления Razor	`_ViewStart`	`.cshtml`	
Импорты представления Razor	`_ViewImports`	`.cshtml`	

Файлы представлений (частичных) и макетов Razor технически идентичны. Вот почему важно соблюдать префиксы с подчеркиванием у этих шаблонов.

Различие в том, что макет Razor задается в свойстве `Layout` другого Razor-файла или в файле `_ViewStart.cshtml`:

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

Частичное представление Razor подключается с помощью элемента `<partial>`:

```
<partial name="_Product" model="product" />
```

Дельный совет: при присвоении имен специальным и общим файлам Razor, таким как макеты и частичные представления, используется префикс с подчеркиванием _, например _ViewStart.cshtml, _Layout.cshtml или _Product.cshtml (может быть, к примеру, частичным представлением карточки товара).

Создание сайта MVC ASP.NET Core

При создании проекта сайта MVC ASP.NET Core для проверки подлинности и авторизации пользователей по умолчанию применяется локальная БД.

- В Visual Studio 2022 для этих целей предназначена SQL Server LocalDB.
- В Visual Studio Code (через команду `dotnet`) используется SQLite, но можно явно указать SQL Server LocalDB.

Посмотрим, как это реализовать.

1. В редакторе кода создайте проект сайта MVC ASP.NET Core с аутентифицируемыми учетными записями, хранящимися в БД:
   - выберите шаблон проекта ASP.NET Core Web App (Model-View-Controller) (Веб-приложение ASP.NET Core (Модель — представление — контроллер)) на языке C# / `mvc`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Mvc`;
   - в поле Solution name (Имя решения) укажите значение `Chapter14`;
   - в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт Individual Accounts (Индивидуальные учетные записи) / `--auth Individual`;
   - установите флажок Configure for HTTPS (Настроить для HTTPS);
   - снимите флажок Enable Docker (Включить поддержку контейнера);
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня).
2. Создайте проект `Northwind.Mvc`.
   - Если MVC-проект создан в программе Visual Studio 2022, то БД для проверки подлинности и авторизации будет храниться в SQL Server LocalDB, но пока ее не существует. Для ее создания в терминале перейдите в каталог `Northwind.Mvc` и выполните команду миграции:

     ```
     dotnet ef database update
     ```

   - Если проект создан с помощью команды `dotnet new`, то в качестве БД используется SQLite, а файл `app.db` уже создан.
3. В корневой папке проекта MVC откройте файл `appsettings.json` и укажите в нем строку подключения к БД проверки подлинности `DefaultConnection`:
   - Для БД SQL Server LocalDB:

     ```
     {
       "ConnectionStrings": {
         "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=
         ➥ aspnet-Northwind.Mvc-...;Trusted_Connection=True;
         ➥ MultipleActiveResultSets=true"
       },
     ```

   - Для SQLite:

     ```
     {
       "ConnectionStrings": {
         "DefaultConnection": "DataSource=app.db;Cache=Shared"
     },
     ```

Обзор стандартного сайта MVC ASP.NET Core

Рассмотрим стандартный проект сайта MVC ASP.NET Core.

1. В папке `Properties` проекта `Northwind.Mvc` откройте файл `launchSettings.json`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5141 для протокола HTTPS и 5142 — для HTTP:

```
"applicationUrl": "https://localhost:5141;http://localhost:5142",
```

2. Сохраните изменения в файле `launchSettings.json`.
3. В редакторе кода или в терминале запустите проект `Northwind.Mvc` с профилем `https`, выбрав браузер Chrome.
   - В Visual Studio 2022: выберите профиль `https` в раскрывающемся списке Startup Project (Запускаемый проект) и браузер Google Chrome — в раскрывающемся списке Web Browser (Браузер), после чего запустите проект сайта `Northind.Mvc` без отладки.
   - В Visual Studio Code: в терминале выполните команду `dotnet run --launch-profile https`. Запустите браузер Chrome и перейдите по адресу `https://localhost:5141/`.
   - В Windows, если появится запрос брандмауэра Windows Defender с сообщением, что «заблокированы некоторые функции запускаемого приложения», нажмите кнопку Allow access (Разрешить доступ).
4. В браузере Chrome откройте панель Developer Tools (Инструменты разработчика).
5. Обновите главную страницу и обратите внимание на следующее (рис. 14.1).
   - Верхнее навигационное меню содержит ссылки на страницы Home, Privacy, Register и Login. Если ширина экрана менее 575 пикселей, меню сворачивается в «гамбургер».
   - Верхний и нижний колонтитулы содержат название сайта, в нашем случае — Northwind.Mvc.

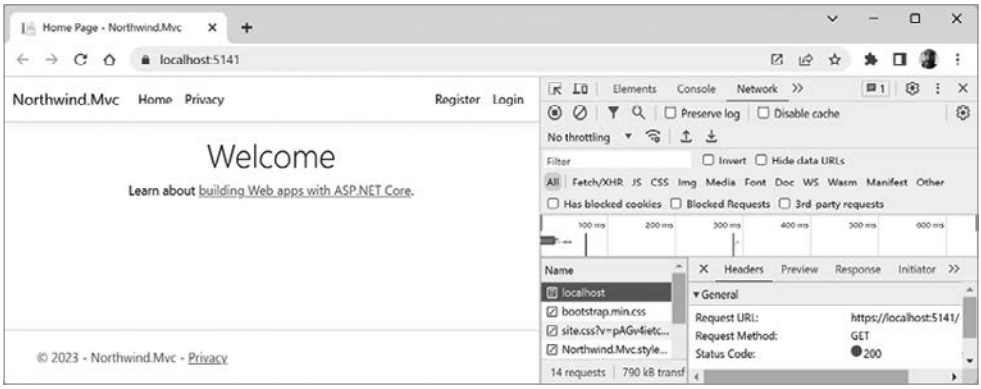


Рис. 14.1. Главная страница веб-приложения на ASP.NET Core MVC

Регистрация посетителей

Пароль по умолчанию должен содержать:

- минимум один не алфавитно-цифровой символ;
- минимум одну цифру (0–9);
- минимум одну прописную букву (A — Z).

В моих тестовых примерах используется пароль `Pa$$w0rd`.

Шаблон проекта поддерживает *двухэтапную регистрацию (double-opt-in, DOI)*: после ввода адреса электронной почты и пароля пользователь получает письмо для подтверждения регистрации. Он должен щелкнуть на ссылке в этом письме, чтобы подтвердить, что хочет зарегистрироваться. Поскольку почтовый сервер еще не настроен, этот шаг можно смоделировать.

1. Закройте панель Developer Tools (Инструменты разработчика).
2. В верхнем навигационном меню страницы щелкните на ссылке Register.
3. Введите адрес электронной почты и пароль, а затем нажмите кнопку Register. (Я использовал значения `test@example.com` и `Pa$$w0rd` соответственно).
4. Щелкните кнопкой мыши на ссылке Click here to confirm your account. Вы будете перенаправлены на веб-страницу Confirm email, которую можете настроить по своему вкусу.
5. В верхней части страницы щелкните кнопкой мыши на ссылке Login, введите адрес электронной почты и пароль (обратите внимание на флажок, который можно установить, чтобы сохранить учетные данные, а также ссылки регистрации нового пользователя и сброса пароля), а затем нажмите кнопку Login.
6. Щелкните кнопкой мыши на ссылке в виде вашего адреса электронной почты в верхнем навигационном меню. Откроется страница управления учетной записью. Здесь можно указать номер телефона, изменить адрес электронной почты и пароль, включить двухфакторную проверку подлинности (если добавлено приложение-аутентификатор), а также управлять (в том числе и удалять) персональными данными. Последняя функция необходима для соблюдения законодательных норм, таких как европейский регламент GDPR.
7. Закройте браузер Chrome, а затем в терминале сайта MVC нажмите сочетание клавиш Ctrl+C, чтобы отключить веб-сервер.

Структура проекта сайта MVC

Откройте редактор кода. В программе Visual Studio 2022 на панели Solution Explorer (Обозреватель решений) нажмите кнопку Show All Files (Показать все файлы). В программе Visual Studio Code выберите команду меню Explorer ▸ Solution Explorer

(Обозреватель ▸ Обозреватель решений). Вы увидите структуру сайта MVC (рис. 14.2).

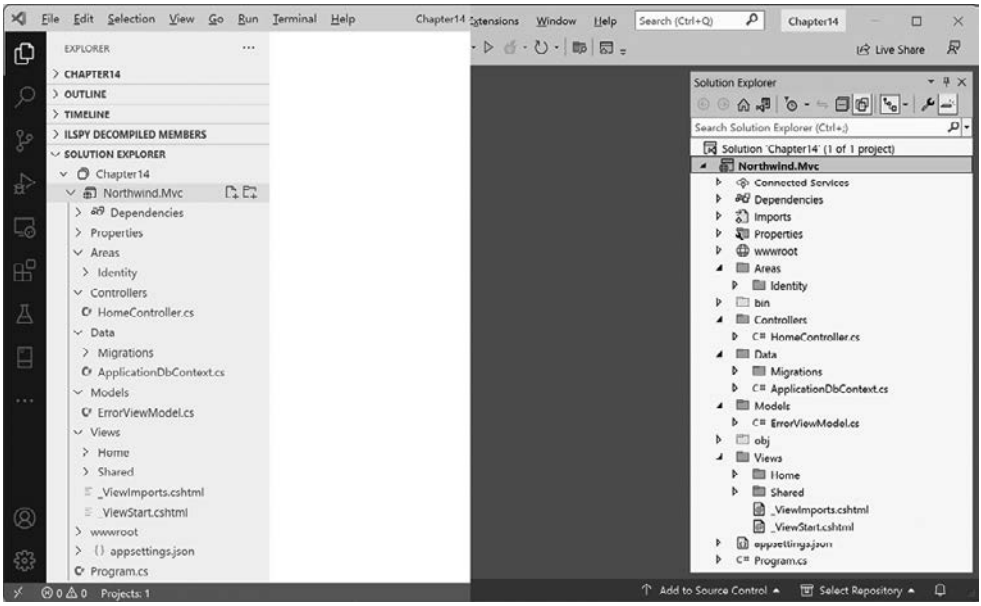


Рис. 14.2. Обозреватели решений в программах VS Code и VS 2022 с проектом ASP.NET Core MVC

Более подробно некоторые из представленных компонентов будут рассмотрены позже, а пока познакомимся с ними поверхностно.

- `Areas` — содержит вложенные папки и файл, необходимый для интеграции проекта сайта с системой ASP.NET Core Identity, которая используется для проверки подлинности.
- `bin`, `obj` — включают временные файлы, используемые в процессе сборки, и готовые сборки проекта. На панели Solution Explorer (Обозреватель решений) программы Visual Studio Code с C# Dev Kit скрытые папки не отображаются, но их можно увидеть в представлении папок (например, Chapter14 на рис. 14.2).
- `Controllers` — содержит классы C# с методами действия, извлекающими модель и передающими ее в представление, например `HomeController.cs`
- `Data` — содержит классы миграции Entity Framework Core, используемые системой ASP.NET Core Identity для хранения данных проверки подлинности и авторизации, например `ApplicationDbContext.cs`.

- `Models` — содержит классы C#, представляющие данные, собранные контроллером и переданные в представление, например `ErrorViewModel.cs`.
- `Properties` — включает файл конфигурации для веб-сервера Kestrel (или IIS или IIS Express в ОС Windows), `launchSettings.json`, который используется для запуска сайта во время разработки. Этот файл применяется только на локальной машине и не развертывается на рабочем сайте.
- `Views` — содержит `.cshtml`-файлы Razor с разметкой HTML и C#-кодом для динамической генерации HTML-ответов. Файл `_ViewStart.cshtml` задает шаблон по умолчанию, а файл `_ViewImports.cshtml` импортирует пространства имен общего назначения, например тег-хелперы:
  - `Home` — содержит файлы Razor для главной страницы и страницы Privacy;
  - `Shared` — содержит файлы Razor для общего шаблона, страницы ошибок и двух частичных представлений для входа в систему и проверки данных.
- `wwwroot` — содержит статические файлы сайта: CSS (каскадные таблицы стилей), JavaScript, библиотеки (например, Bootstrap и jQuery), favicon.ico, изображения и другие ресурсы, например PDF-документы.
- `appsettings.json` и `appsettings.Development.json` — хранят настройки, загружаемые сайтом во время выполнения, например строку подключения к БД ASP.NET Core Identity и уровни регистрации.
- `Northwind.Mvc.csproj` — определяет настройки проекта, включая использование Web .NET SDK, копирование файла `app.db` в выходную папку, а также список необходимых NuGet-пакетов, например:
  - `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore`;
  - `Microsoft.AspNetCore.Identity.EntityFrameworkCore`;
  - `Microsoft.AspNetCore.Identity.UI`;
  - `Microsoft.EntityFrameworkCore.Sqlite` или `Microsoft.EntityFrameworkCore.SqlServer`;
  - `Microsoft.EntityFrameworkCore.Tools`.
- `Program.cs` — содержит автоматически сгенерированный класс `Program` с точкой входа `<Main>$`. Он формирует конвейер обработки HTTP-запросов, размещает сайт с настройками по умолчанию (веб-сервер Kestrel, загрузка `appsettings.json`), добавляет и настраивает необходимые сервисы (например, ASP.NET Core Identity для проверки подлинности, SQLite или SQL Server для хранения идентификационных данных) и определяет маршруты приложения.

Если выбрана SQLite вместо SQL Server для БД ASP.NET Core Identity, появится файл app.db — это БД SQLite, где хранятся данные зарегистрированных пользователей.

Ссылка на библиотеку классов EF Core и регистрация контекста данных

Мы будем ссылаться на модель EF Core, созданную в главе 3.

1. В файле проекта `Northwind.Mvc.csproj` добавьте строгую обработку предупреждений и ссылку на проект контекста БД Northwind:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.DataContext.
SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
</ItemGroup>
```

Убедитесь, что путь Include не содержит символов разрыва строки.

2. В терминале выполните сборку проекта `Northwind.Mvc` командой `dotnet build`.
3. В файле `Program.cs` импортируйте пространство имен для поддержки метода расширения `AddNorthwindContext`:

```
using Northwind.EntityModels; // Для метода AddNorthwindContext
```

4. В разделе, где добавляются сервисы в контейнер, добавьте выражение для регистрации `NorthwindContext` в качестве сервиса:

```
builder.Services.AddNorthwindContext();
```

Определение пользовательских веб-интерфейсов с помощью представлений Razor

Рассмотрим, как создать пользовательский интерфейс веб-страницы на современном сайте MVC ASP.NET Core.

Обзор представлений Razor

В аббревиатуре MVC буква V означает view — «представление». Оно отвечает за преобразование модели в HTML-код или другие форматы.

Для этого используются различные *движки представления*. Движок представления по умолчанию — *Razor*, в котором используется символ `@` для обозначения выполнения кода на сервере.

Разберем главную страницу и шаблон общего назначения.

1. В папке `Views/Home` откройте файл `Index.cshtml`. В нем есть блок кода C#, обернутый в фигурные скобки (`{}`). Этот код выполняется первым и может использоваться для хранения данных, например названия страницы, передаваемого в шаблон общего назначения:

```
@{
  ViewData["Title"] = "Home Page";
}
```

2. В элементе `div` содержится статический HTML-контент с классами Bootstrap (`text-center`, `display-4`), отвечающими за оформление страницы:

```
<div class="text-center">
  <h1 class="display-4">Welcome</h1>
  <p>Learn about <a href="https://learn.microsoft.com/aspnet/core">
    building Web apps with ASP.NET Core</a>.</p>
</div>
```

3. В папке `Views` откройте файл `_ViewImports.cshtml`. В нем импортируется пространство имен для проекта и папки `Models`, а также тег-хелперы ASP.NET Core, о которых мы поговорим позже в этой главе:

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

4. В папке `Views` расположен файл `_ViewStart.cshtml`. Он выполняется при вызове метода `View` в контроллере и устанавливает параметры по умолчанию для всех представлений. К примеру, в нем задан макет общего назначения через свойство `Layout` (без расширения файла):

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

Этот файл не используется при визуализации частичных представлений, например при вызове PartialView вместо метода View.

5. В папке `Shared` откройте файл `_Layout.cshtml`.
6. Элементу `title` присваивается значение из словаря `ViewData`, расположенного в представлении `Index.cshtml`:

```
<title>@ViewData["Title"] - Northwind.Mvc</title>
```

Название страницы отображается на вкладке или в заголовке окна браузера.

7. Далее обрабатываются ссылки на Bootstrap и стили сайта. Символ ~ ссылается на папку `wwwroot`:

```
<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
<link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
<link rel="stylesheet" href="~/Northwind.Mvc.styles.css"
  asp-append-version="true" />
```

8. В элементе `header` визуализируется навигационная панель:

```
<body>
  <header>
    <nav class="navbar ...">
```

9. В разделе `div` со сворачиваемым содержимым визуализируется частичное представление `_LoginPartial`, содержащее форму входа и гиперссылки для навигации. Используются тег-хелперы ASP.NET Core с атрибутами `asp-controller` и `asp-action`:

```
<div class="navbar-collapse collapse d-sm-inline-flex
  justify-content-between">
  <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>
  <partial name="_LoginPartial" />
</div>
```

В элементах a используются тег-хелперы asp-controller и asp-action, определяющие имя контроллера и действие при переходе по ссылке. Атрибут asp-area помогает организовать и группировать страницы в больших проектах MVC-сайтов. В библиотеках Razor asp-area можно также использовать для указания имени функции.

10. В элементе `main` вызывается метод `RenderBody`, который визуализирует содержимое текущего представления:

```
<div class="container">
  <main role="main" class="pb-3">
    @RenderBody()
  </main>
</div>
```

Метод RenderBody внедряет представление Razor, например файла Index.cshtml, в указанную область макета.

11. В элементе `script` размещены сценарии. Они располагаются в нижней части страницы, чтобы не замедлять ее загрузку. Также можно добавить свои сценарии в специальный раздел `Scripts`:

```
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
@await RenderSectionAsync("Scripts", required: false)
```

12. Откройте файл `_LoginPartial.cshtml`, расположенный в папке `Shared`. В нем есть условная конструкция `if`: если пользователь вошел в систему, отображаются ссылки на учетную запись и выход; если не вошел — ссылки на вход и регистрацию.
13. В папке `Shared` откройте файл `_ValidationScriptsPartial.cshtml`. В нем содержатся два сценария, добавляющие проверку на стороне клиента с помощью JavaScript-сценария:

```
<script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
<script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.
min.js">
</script>
```

Дельный совет: укажите файл _ValidationScriptsPartial в разделе Scripts, если хотите включить валидацию в представлении Razor, например Index.cshtml:

```
@section Scripts {
  <partial name="_ValidationScriptsPartial" />
}
```

14. Откройте файл `Error.cshtml` в папке `Shared`. Он содержит разметку для вывода исключений.

Создание прототипов с помощью Bootstrap

Bootstrap — самый популярный фреймворк для создания адаптивных сайтов, удобных для использования на мобильных устройствах. Он сочетает таблицы стилей CSS и библиотеки JavaScript, обеспечивая широкие возможности для разработки.

Этот фреймворк прекрасно подойдет для быстрого создания прототипа пользовательского интерфейса. Однако перед тем, как представить сайт пользовате-

лям, возможно, стоит привлечь веб-дизайнера, чтобы создать уникальную тему Bootstrap или полностью заменить стили на собственные CSS, придавая проекту индивидуальный облик.

Отношение к Bootstrap неоднозначное: одни разработчики его любят, другие — избегают. Среди преимуществ этого фреймворка:

- экономия времени;
- гибкие настройки;
- открытый исходный код;
- подробная официальная документация и множество решений на Stack Overflow.

Тем не менее при небрежном внедрении Bootstrap сайт может:

- выглядеть шаблонно;
- не сочетаться с представлениями ASP.NET Core Identity;
- быть тяжелее по сравнению с решениями, созданными вручную.

Дельный совет: создавая стили, лучше основываться на общей библиотеке, например Bootstrap, которая уже реализует адаптивный дизайн. Однако если важен уникальный стиль или фирменный дизайн, обязательно настройте тему, а не используйте стандартные стили по умолчанию.

Синтаксис и выражения Razor

Прежде чем настраивать главную страницу, разберем пример кода Razor. В нем создается заказ с ценой и количеством, а затем информация выводится на веб-страницу:

```
@{
  Order order = new()
  {
    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>
```

Показанный файл Razor приведет к некорректному выводу:

```
Your order for 3 of Sushi has a total cost of $8.49 * 3
```

Хотя в Razor и допускается синтаксис `@объект.свойство` с любыми свойствами, само выражение следует заключить в круглые скобки:

```
<div>Your order for @order.Quantity of @order.Product has a total cost
of $@(order.Price * order.Quantity)</div>
```

Такой вариант приведет к правильному результату:

```
Your order for 3 of Sushi has a total cost of $25.47
```

Так гарантируется корректное вычисление и отображение результата.

HTML-хелперы

При создании представлений в ASP.NET Core MVC для генерации разметки можно использовать объект `Html` и его методы. Когда в 2009 году Microsoft впервые представила ASP.NET MVC, такие вспомогательные методы (хелперы) служили основным способом программной визуализации HTML-разметки.

В современных версиях ASP.NET Core HTML-хелперы сохранены в рамках поддержки обратной совместимости, но теперь вместо них обычно применяют тег-хелперы, поскольку интерпретировать и писать их проще. Однако есть ситуации, когда тег-хелперы использовать нельзя, например в компонентах Razor.

Подробнее о тег-хелперах поговорим позже в этой главе.

К числу полезных методов объекта `Html` относятся следующие.

- `ActionLink` — создает ссылку `a`, ведущую к указанному контроллеру и действию. Например, код `Html.ActionLink(linkText: "Binding", actionName: "ModelBinding", controllerName: "Home")` сгенерирует элемент `<a href="/home/modelbinding">Binding</a>`.
- `AntiForgeryToken` — вставляет в элемент `form` скрытое поле `input` с токеном защиты от подделки. Проверяется при отправке формы.
- `Display` и `DisplayFor` — генерируют HTML-разметку для свойства модели с учетом шаблона отображения. Существуют встроенные шаблоны для типов .NET, а пользовательские можно разместить в папке `DisplayTemplates`. Обратите внимание, что в системах, чувствительных к регистру, регистр букв в имени папок крайне важен.
- `DisplayForModel` — создает разметку не для одного свойства, а для всей модели.
- `Editor` и `EditorFor` — создают HTML-разметку для свойства модели, используя шаблон редактора. Встроенные шаблоны используют элементы `label` и `input`, а пользовательские размещаются в папке `EditorTemplates`. Обратите внимание, что в системах, чувствительных к регистру, регистр букв в имени папок крайне важен.

- `EditorForModel` — работает аналогично `EditorFor`, но применяется ко всей модели.
- `Encode` — выполняет безопасное кодирование объекта или строки в формат HTML. Например, строковое значение `"<script>"` будет преобразовано в `"<script>"`. Razor автоматически кодирует строковые значения, поэтому этот метод обычно не нужен.
- `Raw` — выводит строку без HTML-кодирования.
- `PartialAsync` и `RenderPartialAsync` — вставляют частичное представление в текущую страницу. Можно передавать модель и данные представления.

Строго типизированное представление Razor

Чтобы улучшить работу IntelliSense при создании представлений, можно определить тип модели с помощью директивы `@model` в верхней части файла. Изменим главную страницу, чтобы она отображала таблицу заказов из БД Northwind.

1. В папке `Controllers` откройте файл `HomeController.cs` и импортируйте в нем пространство имен для моделей сущностей Northwind и функций EF Core:

```
using Northwind.EntityModels; // Для моделей сущностей Northwind
using Microsoft.EntityFrameworkCore; // Для метода Include
```

2. В классе контроллера создайте поле для хранения контекста данных Northwind и инициализируйте его в конструкторе:

```
private readonly NorthwindContext _db;

public HomeController(ILogger<HomeController> logger
  , NorthwindContext db)
{
  _logger = logger;
  _db = db;
}
```

3. В методе действия `Index` добавьте инструкции для создания модели представления, содержащей все заказы и связанные с ними данные:

```
public IActionResult Index()
{
  IEnumerable<Order> model = _db.Orders
    .Include(order => order.Customer)
    .Include(order => order.OrderDetails)
    .OrderByDescending(order => order.OrderDetails
      .Sum(detail => detail.Quantity * detail.UnitPrice))
    .AsEnumerable();

  return View(model);
}
```

4. В папке `Views` откройте файл `_ViewImports.cshtml`. Добавьте в него оператор для импорта моделей сущностей EF Core для всех представлений и страниц Razor:

```
@using Northwind.EntityModels
```

5. В папке `Views\Home` откройте файл `Index.cshtml`. В начале файла установите тип модели для коллекции заказов:

```
@model IEnumerable<Order>
```

Теперь, когда вы будете использовать слово `Model` в этом представлении, благодаря функции IntelliSense редактор кода будет автоматически подсказывать правильный тип модели.

При вводе кода в режиме просмотра учитывайте следующие моменты:

- для объявления типа модели используйте ключевое слово @model (со строчной буквы m);
- для взаимодействия с экземпляром модели используйте ключевое слово @Model (с заглавной буквой M).

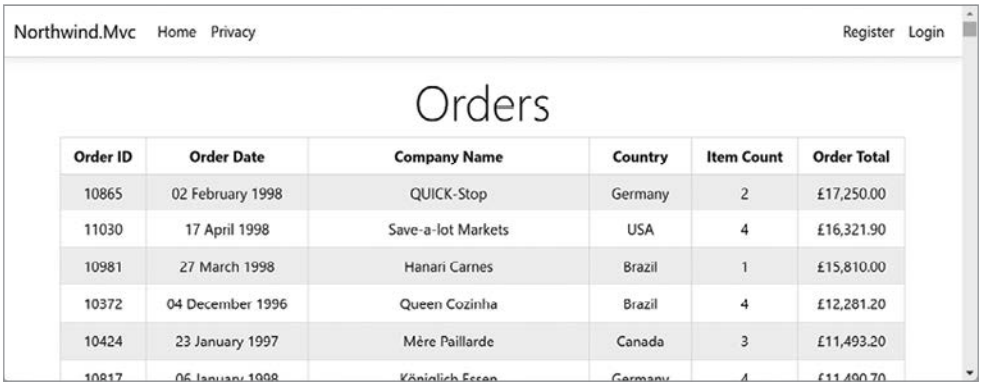
6. Откройте файл `Index.cshtml` и в начальном блоке кода Razor замените содержимое HTML-таблицей заказов:

```
@model IEnumerable<Order>
@{
  ViewData["Title"] = "Orders";
}
<div class="text-center">
  <h1 class="display-4">@ViewData["Title"]</h1>
  <table class="table table-bordered table-striped">
    <thead>
      <tr>
        <th>Order ID</th>
        <th>Order Date</th>
        <th>Company Name</th>
        <th>Country</th>
        <th>Item Count</th>
        <th>Order Total</th>
      </tr>
    </thead>
    <tbody>
      @foreach (Order order in Model)
      {
        <tr>
          <td>@order.OrderId</td>
```

```
            <td>@order.OrderDate?.ToString("D")</td>
            <td>@order.Customer?.CompanyName</td>
            <td>@order.Customer?.Country</td>
            <td>@order.OrderDetails.Count()</td>
            <td>@order.OrderDetails.Sum(detail => detail.Quantity *
                detail.UnitPrice).ToString("C")</td>
          </tr>
        }
      </tbody>
    </table>
  </div>
```

Посмотрим на результат настроенной главной страницы.

7. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
8. Запустите проект сайта `Northwind.Mvc` без отладки.
9. Обратите внимание, что на главной странице теперь отображается таблица заказов, где первым указан заказ с наибольшей стоимостью (рис. 14.3).



Order ID	Order Date	Company Name	Country	Item Count	Order Total
10865	02 February 1998	QUICK-Stop	Germany	2	£17,250.00
11030	17 April 1998	Save-a-lot Markets	USA	4	£16,321.90
10981	27 March 1998	Hanari Carnes	Brazil	1	£15,810.00
10372	04 December 1996	Queen Cozinha	Brazil	4	£12,281.20
10424	23 January 1997	Mère Paillarde	Canada	3	£11,493.20
10817	06 January 1998	Königlich Essen	Germany	4	£11,490.70

Рис. 14.3. Обновленная главная страница сайта MVC Northwind

Мой веб-сервер работает локально на ноутбуке с операционной системой Windows 11. Даты, время и денежные единицы форматируются с учетом культуры «Великобритания». Далее мы обсудим, как адаптировать веб-страницу для культуры, предпочтительной для каждого посетителя.

10. Закройте Chrome и завершите работу веб-сервера.

Теперь, рассмотрев основы создания MVC-сайта для отображения данных, обратим внимание на важный аспект, который часто игнорируют при разработке сайтов для Всемирной паутины: поддержку разных языков и культур мира.

Локализация и глобализация в ASP.NET Core

В главе 7 вы научились работать с датами, временем и часовыми поясами, а также узнали о подходах к глобализации и локализации кодовой базы .NET. В этом разделе мы сосредоточимся на локализации сайта, построенного на ASP.NET Core.

Для перевода строковых значений на разные языки, например французский или испанский, можно использовать интерфейс `IStringLocalizer`. Также существует возможность локализовать HTML-контент через интерфейс `IHtmlLocalizer`, но им следует пользоваться с осторожностью, так как HTML-разметка, как правило, должна оставаться одинаковой для всех локалей. Для представлений можно использовать интерфейс `IViewLocalizer`.

Благодаря *локализации запросов* браузер может запрашивать предпочтительный культурный контекст различными способами.

- Путем добавления параметра строки запроса, например `?culture=en-US&ui-culture=en-US`.
- Отправкой в запросе cookie-файлов, например `c=en-US|uic=en-US`.
- Настройкой HTTP-заголовка, например `Accept-Language: en-US,en;q=0.9, fr- FR;q=0.8,fr;q=0.7,en-GB;q=0.6`.

Для локализации запросов нужно вызвать метод `UseRequestLocalization` в настройках конвейера HTTP-запросов в файле `Program.cs`. ASP.NET Core сможет определять такие запросы и автоматически изменять поток обработки, чтобы применить соответствующую культуру для форматирования данных и загрузки значений ресурсов.

Теперь создадим несколько файлов ресурсов для локализации пользовательского интерфейса на американский английский, британский английский и французский языки, а также глобализуем данные, такие как даты и денежные единицы.

1. В папке проекта `Northwind.Mvc` создайте каталог `Resources`. Это стандартное имя для папки, в которой службы локализации ищут файлы ресурсов `*.resx`.
2. В этом каталоге создайте папку `Views`.
3. Вложите в нее папку `Home`.

Создание файлов ресурсов

Процесс создания файлов ресурсов (`*.resx`) зависит от используемого редактора кода.

Чтобы сэкономить время, можно скачать .resx-файлы из репозитория GitHub, расположенного по адресу https://github.com/markjprice/apps-services-net8/tree/main/code/Chapter14/Northwind.Mvc/Resources/Views/Home.

В Visual Studio 2022

Для создания локализованных файлов потребуется редактор и специальный проект.

1. В папке `Home` создайте ресурсный файл `Index.en-US.resx`.
2. Откройте его в редакторе и определите имена и значения локализованных меток (рис. 14.4).

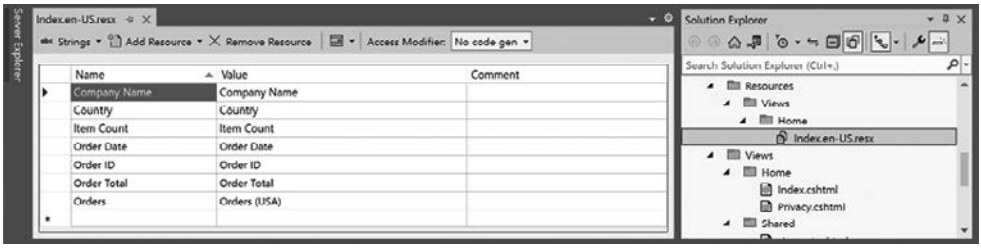


Рис. 14.4. Пример использования редактора для определения локализованных меток

В JetBrains Rider есть собственный редактор файлов ресурсов, который отображает файлы .resx в виде сетки, где столбцы языков следуют друг за другом — каждый для своего языка. Это гораздо удобнее, чем редактировать каждый файл по отдельности, как в программе Visual Studio 2022.

3. Закройте редактор.
4. Продублируйте файл и присвойте копии имя `Index.en-GB.resx`.

Внимание! Будьте осторожны и не изменяйте записи в столбце Name (Имя), так как они используются для поиска локализованных значений для всех языков! Можно изменять только записи в столбцах Value (Значение) и Comment (Комментарий).

5. В файле `Index.en-GB.resx` измените значение `Orders (USA)` на `Orders (UK)`, чтобы подчеркнуть разницу.
6. Закройте редактор.
7. Продублируйте файл и присвойте копии имя `Index.fr-FR.resx`.
8. В файле `Index.fr-FR.resx` в столбце `value` укажите французский язык. (Переводы приведены в пошаговых инструкциях в следующем разделе, посвященном программе Visual Studio Code.)
9. Продублируйте файл и присвойте копии имя `Index.fr.resx`.
10. В файле `Index.fr.resx` измените последнее значение на `Commandes (Neutral French)`.

В Visual Studio Code

Если вы используете Visual Studio Code, то вам придется править файл без специального редактора.

1. В папке `Home` создайте файл `Index.en-US.resx`.
2. Добавьте в этот файл ресурсы на американском английском языке:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="Company Name" xml:space="preserve">
    <value>Company Name</value>
  </data>
  <data name="Country" xml:space="preserve">
    <value>Country</value>
  </data>
  <data name="Item Count" xml:space="preserve">
    <value>Item Count</value>
  </data>
  <data name="Order Date" xml:space="preserve">
    <value>Order Date</value>
  </data>
  <data name="Order ID" xml:space="preserve">
    <value>Order ID</value>
  </data>
  <data name="Order Total" xml:space="preserve">
    <value>Order Total</value>
  </data>
  <data name="Orders" xml:space="preserve">
    <value>Orders (USA)</value>
  </data>
</root>
```

3. Продублируйте файл и присвойте копии имя `Index.en-GB.resx`.
4. В файле `Index.en-GB.resx` измените значение `Orders (USA)` на `Orders (UK)`, чтобы подчеркнуть разницу.
5. Продублируйте файл и присвойте копии имя `Index.fr-FR.resx`.
6. В файле `Index.fr-FR.resx` в столбце `value` укажите французский язык:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="Company Name" xml:space="preserve">
    <value>Nom de l'entreprise</value>
  </data>
  <data name="Country" xml:space="preserve">
    <value>Pays</value>
  </data>
  <data name="Item Count" xml:space="preserve">
    <value>Nombre d'éléments</value>
  </data>
```

```
  <data name="Order Date" xml:space="preserve">
    <value>Date de commande</value>
  </data>
  <data name="Order ID" xml:space="preserve">
    <value>Numéro de commande</value>
  </data>
  <data name="Order Total" xml:space="preserve">
    <value>Total de la commande</value>
  </data>
  <data name="Orders" xml:space="preserve">
    <value>Commandes (France)</value>
  </data>
</root>
```

7. Продублируйте файл и присвойте копии имя `Index.fr.resx`.
8. В файле `Index.fr.resx` измените последнее значение на `Commandes (Neutral French)`.

Изменение представлений Razor с помощью внедряемого сервиса локализатора представлений

Для выполнения следующих шагов можно использовать любой редактор кода Microsoft.

1. В папке `Views/Home` откройте файл `Index.cshtml`, импортируйте в нем пространство имен для работы с локализацией, внедрите сервис `IViewLocalizer` и добавьте код для использования меток в модели представления:

```
@using Microsoft.AspNetCore.Mvc.Localization
@model IEnumerable<Order>
@inject IViewLocalizer Localizer
@{
  ViewData["Title"] = Localizer["Orders"];
}

<div class="text-center">
  <h1 class="display-4">@ViewData["Title"]</h1>
  <table class="table table-bordered table-striped">
    <thead>
      <tr>
        <th>@Localizer["Order ID"]</th>
        <th>@Localizer["Order Date"]</th>
        <th>@Localizer["Company Name"]</th>
        <th>@Localizer["Country"]</th>
        <th>@Localizer["Item Count"]</th>
        <th>@Localizer["Order Total"]</th>
      </tr>
    </thead>
```

Дельный совет: ключевые значения, такие как Order ID, используются для поиска локализованных значений. Если нужное значение отсутствует, возвращается ключ по умолчанию. Рекомендуется использовать ключи, которые могут служить запасным вариантом, например приводить английские названия с пробелами как в .resx-файлах, так и в коде.

2. В файле `Program.cs` перед вызовом метода `AddControllersWithViews` добавьте код для внедрения локализации и укажите папку `Resources` в качестве локации файлов ресурсов. После этого добавьте вызов для локализации представлений:

```
builder.Services.AddLocalization(
  options => options.ResourcesPath = "Resources");

builder.Services.AddControllersWithViews()
  .AddViewLocalization();
```

3. В файле `Program.cs` после вызова метода `Build()` для объекта `app` добавьте код для объявления поддерживаемых культур: английский (США), английский (Великобритания), нейтральный французский и французский (Франция). Создайте объект параметров локализации и добавьте эти культуры для локализации пользовательского интерфейса (`UICultures`) и глобализации данных, таких как даты и денежные единицы (`Cultures`):

```
string[] cultures = { "en-US", "en-GB", "fr", "fr-FR" };

RequestLocalizationOptions localizationOptions = new();

// cultures[0] даст значение "en-US"
localizationOptions.SetDefaultCulture(cultures[0])

  // Установка глобализации форматов данных, таких как даты
  // и денежные единицы
  .AddSupportedCultures(cultures)

  // Установка локализации текста пользовательского интерфейса
  .AddSupportedUICultures(cultures);

app.UseRequestLocalization(localizationOptions);
```

4. Запустите проект сайта `Northwind.Mvc`.
5. В браузере Chrome откройте страницу Settings (Настройки).
6. В поисковую строку введите запрос `lang` и обратите внимание на раздел Preferred languages (Предпочитаемые языки) (рис. 14.5).

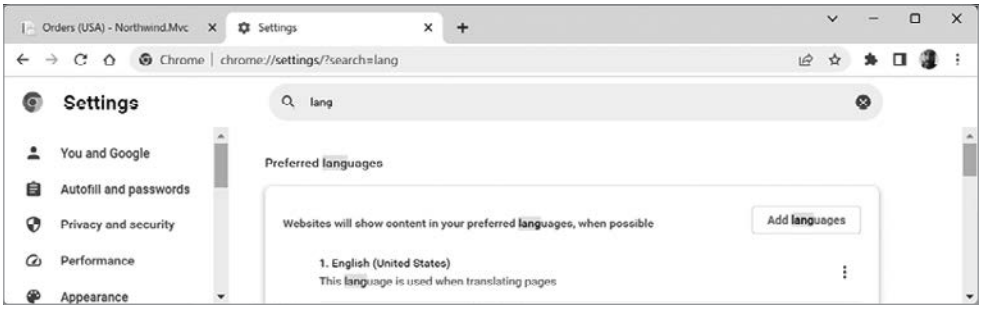


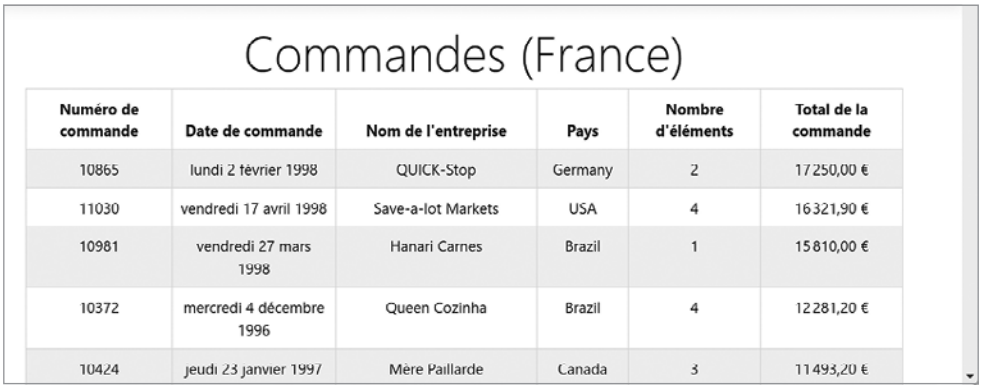
Рис. 14.5. Поиск в настройках Chrome раздела предпочитаемых языков

Внимание! В локализованной версии Chrome (когда пользовательский интерфейс отображается на вашем родном языке, например французском) вам придется искать слово «язык» на вашем языке. Например, на французском это будет langue (поиск по запросу lang все равно сработает), на испанском — idioma, а на русском — язык.

7. Нажмите кнопку Add languages (Добавить языки), выполните поиск по запросу `french` и выберите оба варианта: français и français (France), а затем нажмите кнопку Add (Добавить).

Внимание! В локализованной версии браузера Chrome выполнять поиск нужно на своем языке. Например, на испанском это будет Francés, а на валлийском — Ffrangeg.

8. Аналогичным образом добавьте языки English (United States) и English (United Kingdom), если их нет в списке.
9. Нажмите кнопку в виде трех вертикальных точек справа от пункта français (France) и выберите пункт Move to the top (К началу), чтобы перенести французский язык в начало списка.
10. Закройте вкладку Settings (Настройки).
11. В браузере Chrome выполните принудительную перезагрузку/обновление страницы (нажав и удерживая клавишу Ctrl, нажмите кнопку Refresh (Обновить страницу)) и обратите внимание, что на главной странице теперь используются локализованные метки и французские форматы для дат и денежных единиц (рис. 14.6).
12. Повторите описанные выше действия для других языков, например для британского английского.



Commandes (France)

Numéro de commande	Date de commande	Nom de l'entreprise	Pays	Nombre d'éléments	Total de la commande
10865	lundi 2 février 1998	QUICK-Stop	Germany	2	17 250,00 €
11030	vendredi 17 avril 1998	Save-a-lot Markets	USA	4	16 321,90 €
10981	vendredi 27 mars 1998	Hanari Carnes	Brazil	1	15 810,00 €
10372	mercredi 4 décembre 1996	Queen Cozinha	Brazil	4	12 281,20 €
10424	jeudi 23 janvier 1997	Mère Paillarde	Canada	3	11 493,20 €

Рис. 14.6. Таблица заказов, локализованная и глобализированная для Франции

13. Откройте панель Developer Tools (Инструменты разработчика) и обратите внимание на заголовки запросов с британским английским языком (`en-GB`) (рис. 14.7).

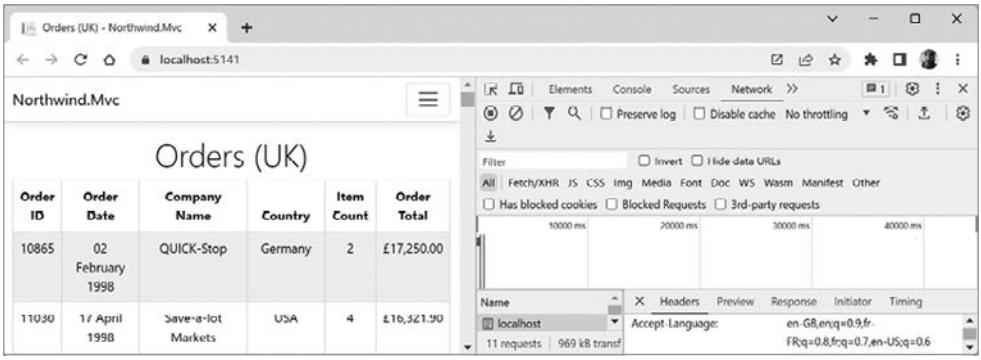


Рис. 14.7. Заказы, переведенные на британский английский язык благодаря заголовку Accept-Language: en-GB

14. Закройте браузер и завершите работу веб-сервера.

Заголовок Accept-Language

Рассмотрим, как устроен заголовок `Accept-Language`:

```
Accept-Language: en-US,en;q=0.9,fr-FR;q=0.8,fr;q=0.7,en-GB;q=0.6
```

Этот заголовок играет важную роль в локализации, поскольку с его помощью браузер может указать предпочтительные языки. Коды культур в заголовке раз-

деляются запятыми. Каждый код может быть нейтральным (только язык) или специфичным (язык и регион). Кроме того, может быть указан *приоритет quality value* (`q`) в диапазоне от `0.0` до `1.0` (по умолчанию `1.0`). Так, представленный выше пример заголовка `Accept-Language` нужно читать следующим образом.

- `en-US` — американский английский имеет наивысший приоритет `1.0` (если приоритет не указан явно).
- `en;q=0.9` — английский в любой точке мира имеет приоритет `0.9`.
- `fr-FR;q=0.8` — французский (Франция) имеет приоритет `0.8`.
- `fr;q=0.7` — французский в любой точке мира — приоритет `0.7`.
- `en-GB;q=0.6` — британский английский имеет самый низкий приоритет — `0.6`.

Применение тег-хелперов к HTML-элементам

Тег-хелперы упрощают создание динамических HTML-элементов, улучшая читаемость, редактируемость и поддержку разметки. Код становится чище по сравнению с использованием HTML-хелперов.

Тем не менее тег-хелперы не могут полностью заменить HTML-хелперы. Есть ситуации, когда только с помощью последних можно достичь нужного результата, например при визуализации сложных вложенных элементов. Кроме того, тег-хелперы неприменимы в компонентах Razor. Поэтому стоит рассматривать их как дополнительный инструмент, который подходит лишь для определенных случаев.

Тег-хелперы особенно полезны для *фронтенд-разработчиков*, работающих с языками HTML, CSS и JavaScript. Они не требуют знания C#, так как применимы к обычным HTML-атрибутам. Кроме того, если ваш редактор поддерживает функцию IntelliSense (например, Visual Studio 2022 или Visual Studio Code), вы сможете быстро выбирать имена и значения атрибутов с его помощью.

Сравнение HTML-хелперов и тег-хелперов

Для отображения гиперссылки на действие контроллера с помощью HTML-хелперов можно использовать следующий код:

```
@Html.ActionLink("View our privacy policy.", "Privacy", "Index")
```

Чтобы было понятно, как это работает, можно использовать именованные параметры:

```
@Html.ActionLink(linkText: "View our privacy policy.",
  action: "Privacy", controller: "Index")
```

Однако код с тег-хелперами будет более понятным и ясным для тех, кто работает в первую очередь с HTML, а не с C#:

```
<a asp-action="Privacy" asp-controller="Home">View our privacy policy.</a>
```

В результате во всех трех случаях генерируется одинаковый HTML-элемент:

```
<a href="/home/privacy">View our privacy policy.</a>
```

В следующих разделах мы рассмотрим наиболее распространенные тег-хелперы:

- привязки;
- кэша;
- среды;
- изображения;
- формы.

Тег-хелпер привязки

Сначала создадим три кликабельные гиперссылки, стилизованные под кнопки. Они будут вести на страницы с заказами: все заказы, заказы одного клиента и заказы в одной стране. Так вы сможете ознакомиться с основами создания ссылок на контроллеры и действия, а также с передачей параметров через маршрут и произвольные параметры строки запроса.

Начнем с тег-хелпера привязки.

1. В папке `Views` откройте файл `_ViewImports.cshtml` и обратите в нем внимание на директиву `@addTagHelper`, которая добавляет хелперы ASP.NET Core:

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

Вы можете создать собственные тег-хелперы, и они будут регистрироваться аналогичным образом. Однако эта тема выходит за рамки данной книги. Для подробностей обратитесь к документации: https://learn.microsoft.com/ru-ru/aspnet/core/mvc/views/tag-helpers/authoring.

2. В папке `Views/Home` откройте файл `Privacy.cshtml` и добавьте в него разметку для форматирования абзаца с кликабельными гиперссылками, стилизованными под кнопки с помощью элемента `a`:

```
<p>
  <a asp-controller="Home" asp-action="Index" class="btn btn-primary"
    role="button">Orders</a>

  <a asp-controller="Home" class="btn btn-outline-primary"
    role="button">This Page</a>

  <a asp-controller="Home" asp-action="Index" asp-route-id="ALFKI"
    class="btn btn-outline-primary" role="button">Orders for Alfreds
      Futterkiste</a>

  <a asp-controller="Home" asp-action="Index" asp-route-country="Brazil"
    class="btn btn-outline-primary" role="button">Orders in Brazil</a>
</p>
```

Если вы приводите имя контроллера без указания действия, по умолчанию используется текущее действие, в данном случае Privacy. Атрибут asp-route-{имя_параметра} может быть произвольным. В примере выше мы использовали параметры id и country. Параметр id будет сопоставлен с одноименным параметром маршрута, а country, не являясь параметром маршрута, будет передана как строка запроса.

3. В папке `Controllers` откройте файл `HomeController.cs`. Измените метод действия `Index`, добавив два параметра для передачи идентификатора клиента и страны. Затем измените LINQ-запрос, применив эти параметры для фильтрации заказов, если они указаны:

```
public IActionResult Index(
  string? id = null, string? country = null)
{

  // Сначала используется упрощенная начальная модель
  IEnumerable<Order> model = db.Orders
    .Include(order => order.Customer)
    .Include(order => order.OrderDetails);

  // Добавление фильтрации на основе параметров
  if (id is not null)
  {
    model = model.Where(order => order.Customer?.CustomerId == id);
  }
  else if (country is not null)
  {
    model = model.Where(order => order.Customer?.Country == country);
  }
```

```
    // Добавляется упорядочение, и элементы становятся перечисляемыми
    model = model
      .OrderByDescending(order => order.OrderDetails
        .Sum(detail => detail.Quantity * detail.UnitPrice))
      .AsEnumerable();

    return View(model);
  }
```

4. Запустите проект сайта `Northwind.Mvc`.
5. Откройте панель Developer Tools (Инструменты разработчика) и перейдите на вкладку Elements (Элементы).
6. На главной странице щелкните кнопкой мыши на ссылке Privacy, чтобы перейти на эту страницу. Вы увидите кнопки, в том числе их исходный HTML-код с путями в атрибутах `href`, сгенерированными тег-хелперами привязки (рис. 14.8).

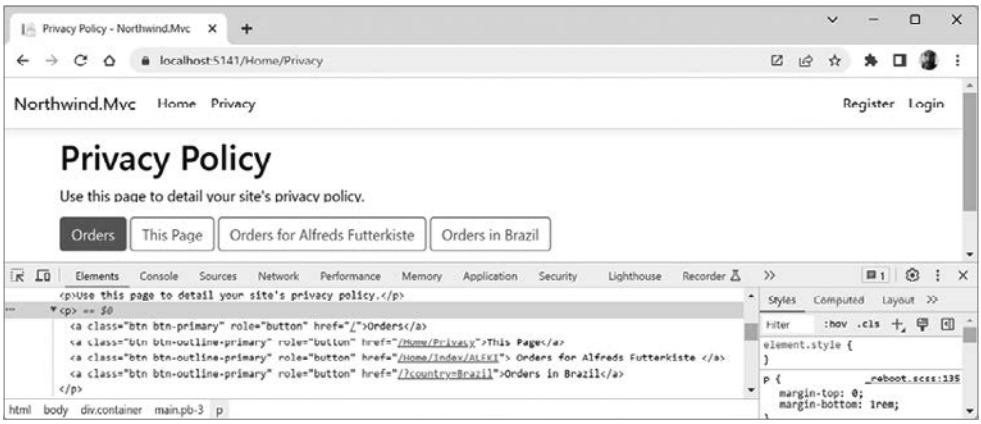


Рис. 14.8. Три гиперссылки, стилизованные под кнопки, созданные тег-хелперами привязки

7. Нажмите каждую кнопку, а затем вернитесь на страницу Privacy, чтобы убедиться, что ссылки работают корректно.
8. Закройте браузер и завершите работу веб-сервера.
9. В папке `Views/Home` откройте файл `Index.cshtml` и добавьте элемент `a` в конце таблицы заказов:

```
  </table>
  <a id="endOfTable" />
</div>
```

10. В папке `Views/Home` откройте файл `Privacy.cshtml`. После элементов `a` добавьте еще один тег `<a>` со ссылкой на элемент с идентификатором `endOfTable` и атрибутом `asp-fragment`. Не забудьте закрыть элемент привязки:

```
<a asp-controller="Home" asp-action="Index" asp-fragment="endOfTable"
  class="btn btn-outline-primary">Orders (end of table)</a>
```

11. Измените второй элемент `a`, явно указав протокол HTTPS:

```
<a asp-controller="Home" asp-protocol="https"
  class="btn btn-outline-primary">This Page</a>
```

12. В папке `Controllers` откройте файл `HomeController.cs` и добавьте в него метод действия `Shipper`, принимающий параметр для сущности `shipper` и передающий ее в представление:

```
public IActionResult Shipper(Shipper shipper)
{
  return View(shipper);
}
```

Этот метод позволяет обрабатывать любые типы запросов, например GET или POST. При GET-запросе сущность shipper в виде пар «ключ — значение» передается через строку запроса, а при POST-запросе — через его тело.

13. В папке `Views/Home` добавьте пустое представление Razor и присвойте ему имя `Shipper.cshtml`.
14. Добавьте в него следующую разметку:

```
@model Shipper
@{
  ViewData["Title"] = "Shippers";
}
<h1>@ViewData["Title"]</h1>
<div>
  <div class="mb-3">
    <label for="shipperIdInput" class="form-label">Shipper Id</label>
    <input type="number" class="form-control" id="shipperIdInput"
          value="@Model.ShipperId">
  </div>
  <div class="mb-3">
    <label for="companyNameInput" class="form-label">Company Name</label>
    <input class="form-control" id="companyNameInput"
    ➥ value="@Model.CompanyName">
  </div>
```

```
  <div class="mb-3">
    <label for="phoneInput" class="form-label">Phone</label>
    <input class="form-control" id="phoneInput" value="@Model.Phone">
  </div>
</div>
```

15. В папке `Views/Home` откройте файл `Privacy.cshtml`. В начале файла добавьте код для включения контекста БД Northwind и создайте Razor-функцию для формирования словаря со строковыми ключами и значениями из таблицы грузоотправителей:

```
@inject NorthwindContext db
@{
  ViewData["Title"] = "Privacy Policy";
}
@functions {
  public async Task<IDictionary<string, string>> GetShipperData()
  {
    // Поиск грузоотправителя с идентификатором 1
    Shipper? shipper = await db.Shippers.FindAsync(1);

    Dictionary<string, string> keyValuePairs = new();

    if (shipper != null)
    {
      keyValuePairs = new()
      {
        { "ShipperId", shipper.ShipperId.ToString() },
        { "CompanyName", shipper.CompanyName },
        { "Phone", shipper.Phone ?? string.Empty }
      };
    }

    return keyValuePairs;
  }
}
```

16. Ниже элементов `a` добавьте еще один элемент привязки — для передачи словаря на текущую страницу:

```
<a asp-controller="Home" asp-action="Shipper"
  asp-all-route-data="await GetShipperData()"
  class="btn btn-outline-primary">Shipper</a>
```

Передача сложного объекта через строку запроса (как в этом примере) быстро достигает ограничения в 1000 символов для URL-адресов. Для передачи больших объектов рекомендуется использовать метод POST вместо GET и элемент form, а не a.

17. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.

18. Запустите проект сайта `Northwind.Mvc`.
19. Откройте панель Developer Tools (Инструменты разработчика) и перейдите на вкладку Elements (Элементы).
20. На главной странице щелкните на ссылке Privacy, чтобы перейти на эту страницу. Вы увидите кнопки, в том числе их исходный HTML-код с путями в атрибутах `href`, сгенерированными тег-хелперами привязки (рис. 14.9).

Рис. 14.9. Фрагмент страницы и разметка для передачи сложного объекта в строке запроса

Указав протокол, вы сгенерируете абсолютный URL-адрес, правда для этого потребуется ввести протокол, домен и номер порта, а также относительный путь. Это удобно, если нужно получить абсолютный URL-адрес вместо обычного относительного, показанного во второй ссылке выше.

21. Нажмите кнопку Orders (end of table) и убедитесь, что браузер переходит на главную страницу и сразу прокручивает страницу до конца таблицы заказов.
22. Вернитесь на страницу Privacy, нажмите кнопку Shipper и убедитесь, что данные о грузоотправителе заранее внесены в форму.
23. Закройте браузер и завершите работу веб-сервера.

Тег-хелперы кэша

Тег-хелперы кэша и распределенного кэша повышают производительность веб-страниц, кэшируя их содержимое с помощью поставщиков в оперативной памяти или зарегистрированных поставщиков распределенного кэша. Чтение и запись объектов в эти кэши рассматривались в главе 9. Теперь разберемся, как хранить в них фрагменты HTML для представления.

Кэш в оперативной памяти лучше всего подходит для одного веб-сервера или фермы веб-серверов с привязкой к сессиям, когда повторные запросы от одного

браузера обслуживаются одним сервером. Распределенный кэш удобнее для фермы веб-серверов и облачных решений, например Azure. Можно зарегистрировать провайдеров для SQL Server, Redis или NCache либо создать собственного.

Атрибуты, применимые к тег-хелперам кэша:

- `enabled` — по умолчанию `true`. Позволяет включить элемент `cache` в разметку и на этапе выполнения решить, использовать его или нет;
- `expires-after` — значение типа `TimeSpan`, определяющее длительность хранения кэша. По умолчанию 20 минут (`00:20:00`);
- `expires-on` — значение типа `DateTimeOffset`, определяющее конкретное время истечения срока хранения. По умолчанию не используется;
- `expires-sliding` — значение типа TimeSpan, определяющее, через сколько времени объект будет удален, если к нему не обращались. Полезно для кэширования сущностей БД с разной востребованностью: часто используемые останутся в кэше, а редко запрашиваемые будут удаляться. По умолчанию не используется;
- `vary-by-{тип}` — создает разные кэшированные версии в зависимости от значений HTTP-заголовков, пользователя, маршрута, cookie-файлов, строки запроса или пользовательского параметра.

Рассмотрим пример использования тег-хелперов кэша.

1. В папке `Views/Home` откройте файл `Index.cshtml` и между шапкой и таблицей добавьте элементы `div` с разметкой Bootstrap:
   - в одном столбце текущие дата и время в UTC отображаются в реальном времени;
   - в другом — кэшированное время:

```
<div class="row">
  <div class="col">
    <h2>Live</h2>
    <p class="alert alert-info">
      UTC: @DateTime.UtcNow.ToLongDateString() at
           @DateTime.UtcNow.ToLongTimeString()
    </p>
  </div>
  <div class="col">
    <h2>Cached</h2>
    <p class="alert alert-secondary">
      <cache>
        UTC: @DateTime.UtcNow.ToLongDateString() at
             @DateTime.UtcNow.ToLongTimeString()
      </cache>
    </p>
  </div>
</div>
```

2. Запустите проект сайта `Northwind.Mvc`.
3. Несколько раз обновите главную страницу: время в левом столбце обновляется постоянно в соответствии с реальным, а в правом остается неизменным (по умолчанию 20 минут) (рис. 14.10).

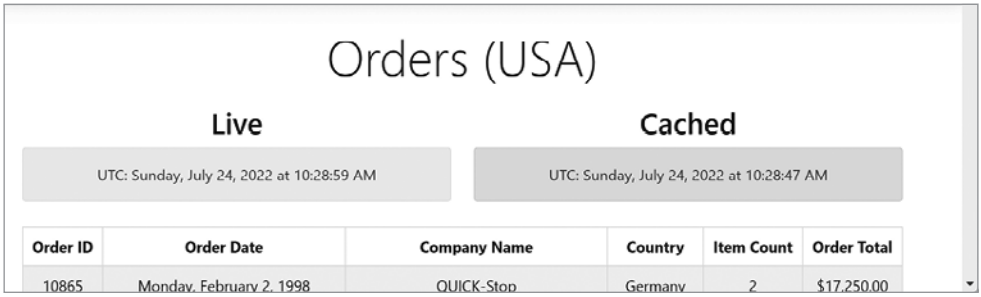


Рис. 14.10. Актуальное и кэшированное время в формате UTC

4. Закройте браузер и завершите работу веб-сервера.
5. В папке `Views/Home` откройте файл `Index.cshtml` и измените элемент `cache` так, чтобы срок хранения истекал через 10 секунд:

```
<cache expires-after="@TimeSpan.FromSeconds(10)">
```

6. Запустите проект сайта `Northwind.Mvc`.
7. Вновь несколько раз обновите главную страницу: время в левом столбце обновляется постоянно в соответствии с реальным, а в правом кэшируется на 10 секунд, после чего обновляется.
8. Закройте браузер и завершите работу веб-сервера.

Тег-хелпер среды

Тег-хелпер среды отображает содержимое только в том случае, если текущая среда соответствует одному из значений в списке имен, разделенных запятыми. Это удобно, например, когда нужно показать инструкции для тестировщика в промежуточной среде или скрыть специфическую для клиента информацию в производственной среде.

Помимо атрибута `names` со списком сред через запятую, можно использовать атрибуты:

- `include` — работает так же, как `names`;
- `exclude` — исключает указанные среды и визуализирует содержимое во всех остальных.

Рассмотрим пример.

1. В папке `Views/Home` откройте файл `Privacy.cshtml` и внедрите зависимость для среды веб-хоста:

```
@inject IWebHostEnvironment webhost
```

2. После заголовка добавьте два элемента `environment`. Первый будет виден только разработчикам и тестировщикам. Второй — только обычным пользователям:

```
<environment names="Development,Staging">
  <div class="alert alert-warning">
    <h2>Attention developers and testers</h2>
    <p>This is a warning that only developers and testers will see.
      Current environment:
      <span class="badge bg-warning">@webhost.EnvironmentName</span>
    </p>
  </div>
</environment>
<environment names="Production">
  <div class="alert alert-info">
    <h2>Welcome, visitor!</h2>
    <p> This is information that only a visitor to the production
      Website will see. Current environment:
      <span class="badge bg-info">@webhost.EnvironmentName</span>
    </p>
  </div>
</environment>
```

3. Запустите проект сайта `Northwind.Mvc`.
4. Перейдите на страницу Privacy и убедитесь, что сообщение для разработчиков и тестировщиков отображается (рис. 14.11).

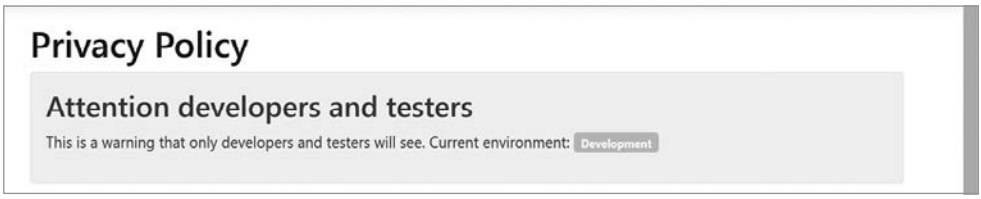


Рис. 14.11. Страница Privacy в промежуточной среде

5. Закройте браузер и завершите работу веб-сервера.
6. В файле `launchSettings.json` (папка `Properties`) в профиле `https` измените среду на `Production`, как показано ниже:

```
"https": {
  ...
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Production"
  }
},
```

7. Запустите проект сайта `Northwind.Mvc`.
8. Перейдите на страницу Privacy — теперь появится сообщение для обычных посетителей (рис. 14.12).

Рис. 14.12. Страница Privacy в производственной среде

9. Закройте браузер и завершите работу веб-сервера.
10. В папке `Properties` откройте файл `launchSettings.json` и в профиле `https` смените среду на `Development`.

Кэширование и тег-хелперы

При добавлении атрибута `asp-append-version="true"` в элемент `link`, `img` или `script` автоматически вызывается соответствующий тег-хелпер. Он добавляет в строку запроса параметр `v` со значением, генерируемым на основе SHA256-хеша исходного файла. Пример атрибута `src` показан ниже:

```
<script src="~/js/site.js? v=Kl_dqr9NVtnMdsM2MUg4qthUnWZm5T1fCEimBPWDNgM"></script>
```

Вы можете убедиться в этом на примере текущего проекта, поскольку в файле _Layout.cshtml есть элемент <script src="~/js/site.js" asp-append-version="true"> </script>.

Если в файле `site.js` изменится хотя бы на один байт, его хеш обновится и браузер или CDN заменят кэшированную версию.

Важно: атрибут `asp-append-version` поддерживается только с локальными файлами (например, из папки `wwwroot`). Удаленные ссылки не поддерживаются.

Тег-хелперы формы

Тег-хелперы формы генерируют атрибут `action` в элементах `form`, привязывая их к именованному маршруту или действию контроллера MVC. Как и тег-хелперы привязки, они позволяют передавать параметры через атрибут `asp-route-<имя_параметра>`. Дополнительно она автоматически создает скрытый проверочный токен для защиты от подделки межсайтовых запросов. Чтобы эта защита работала, необходимо добавить атрибут `[ValidateAntiForgeryToken]` к методу действия, обрабатывающему `POST`-запросы.

Тег-хелперы метки и ввода создают связанные метки и поля ввода для свойств модели. Они автоматически формируют атрибуты `id`, `name` и `for`, а также добавляют атрибуты проверки и сообщения об ошибках.

Рассмотрим пример формы для ввода информации о грузоотправителе.

1. В папке `Views/Home` откройте файл `Shipper.cshtml`. Скопируйте код, содержащий информацию о грузоотправителе, оберните его в элемент `form`, воспользовавшись тег-хелпером формы, и настройте элементы `label` и `input`, задействовав тег-хелперы метки и ввода:

```
@model Shipper
@{
  ViewData["Title"] = "Shippers";
}
<h1>@ViewData["Title"]</h1>
<h2> Without Form Tag Helper</h2>
<div>
  <div class="mb-3">
    <label for="shipperIdInput" class="form-label">Shipper ID</label>
    <input type="number" class="form-control" id="shipperIdInput"
          value="@Model.ShipperId">
  </div>
  <div class="mb-3">
    <label for="companyNameInput" class="form-label">Company Name</label>
    <input class="form-control" id="companyNameInput"
          value="@Model.CompanyName">
  </div>
  <div class="mb-3">
    <label for="phoneInput" class="form-label">Phone</label>
    <input class="form-control" id="phoneInput" value="@Model.Phone">
  </div>
</div>
<h2> With Form Tag Helper</h2>
<form asp-controller="Home" asp-action="ProcessShipper" class="form-horizontal"
      role="form">
  <div>
```

```
    <div class="mb-3">
      <label asp-for="ShipperId" class="form-label" />
      <input asp-for="ShipperId" class="form-control">
    </div>
    <div class="mb-3">
      <label asp-for="CompanyName" class="form-label" />
      <input asp-for="CompanyName" class="form-control">
    </div>
    <div class="mb-3">
      <label asp-for="Phone" class="form-label" />
      <input asp-for="Phone" class="form-control">
    </div>
    <div class="mb-3">
      <input type="submit" class="form-control">
    </div>
  </div>
</form>
```

2. В папке `Controllers` откройте файл `HomeController.cs`. Добавьте метод действия `ProcessShipper`. Он должен принимать сущность грузоотправителя в качестве параметра и возвращать ее в виде JSON-документа с помощью метода `Json()`:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult ProcessShipper(Shipper shipper)
{
  return Json(shipper);
}
```

3. Запустите проект сайта `Northwind.Mvc`.
4. Перейдите на страницу Privacy, а затем нажмите кнопку Shipper.
5. Щелкните правой кнопкой мыши на открывшейся странице, выберите в контекстном меню пункт View page source (Просмотр кода страницы) и обратите внимание на сгенерированные HTML-атрибуты для элементов `form`, `input` и `label`, включая скрытый элемент `RequestVerificationToken`:

```
<h2>With Form Tag Helper </h2>
<form class="form-horizontal" role="form" action="/Home/ProcessShipper"
method="post">
  <div>
    <div class="mb-3">
      <label class="form-label" for="ShipperId" />
      <input class="form-control" type="number" data-val="true"
        data-val-required="The ShipperId field is required."
        id="ShipperId" name="ShipperId" value="1">
    </div>
```

```
    <div class="mb-3">
      <label class="form-label" for="CompanyName" />
      <input class="form-control" type="text" data-val="true"
        data-vallength="The field CompanyName must be a string with
        a maximum length of 40." data-val-length-max="40"
        data-val-required="The CompanyName field is required." id="CompanyName"
        maxlength="40" name="CompanyName" value="Speedy Express">
    </div>
    <div class="mb-3">
      <label class="form-label" for="Phone" />
      <input class="form-control" type="text" data-val="true"
        data-val-length="The field Phone must be a string with a maximum
        length of 24." data-val-length-max="24" id="Phone" maxlength="24"
        name="Phone" value="(503) 555-9831">
    </div>
    <div class="mb-3">
      <input type="submit" class="form-control">
    </div>
  </div>
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8NTt08jabvBCqd1P4J-HCq3X9CDrTPjBphdDdVmG6UT0GFBJk1w7F1OLmNT-jE
GjlGIjfV3kmNUaofOAxlGgiZJwbAR73g-QgFw8oFV_0vjlo45t9dL9E1l1hZzjLXtj8B7y
sDkCYcm8W9zS0T7V3R0" />
</form>
```

6. В разметке формы измените идентификатор грузоотправителя и название компании. Проверьте, как работают атрибуты `maxlength="40"` (ограничивает название компании 40 символами) и `type="number"` (допускает только числа в поле идентификатора).
7. Нажмите кнопку Submit и посмотрите, как браузер отправляет данные и получает JSON-ответ:

```
{"shipperId":1,"companyName":"Speedy Express","phone":"(503)
555-9831","orders":[]}
```

8. Закройте браузер и завершите работу веб-сервера.

Кэширование выходных данных

Кэширование выходных данных похоже на кэширование HTTP-ответов (описано в главе 9), но работает иначе. Оно сохраняет динамически сгенерированные ответы на сервере, чтобы повторно использовать их при следующих запросах без необходимости заново выполнять вычисления. Это повышает производительность. В отличие от кэширования HTTP-ответов, кэширование выходных данных не зависит от клиентов и промежуточных серверов, которые следуют заголовкам HTTP.

Конечные точки кэширования выходных данных

Рассмотрим процесс кэширования выходных данных на простом примере, применяя его к различным конечным точкам и проверяя корректность работы.

1. В файле `Program.cs` проекта `Northwind.Mvc` после вызова метода `AddNorthwindContext` добавьте промежуточное ПО выходного кэша и измените стандартное время хранения кэша, ограничив его 10 секундами:

```
builder.Services.AddOutputCache(options =>
{
  options.DefaultExpirationTimeSpan = TimeSpan.FromSeconds(10);
});
```

Дельный совет: по умолчанию срок хранения кэша — 1 минута. Подумайте, какое значение лучше подойдет для вашего сценария.

2. В том же файле перед сопоставлением маршрутов контроллера добавьте строку для включения кэширования выходных данных:

```
app.UseOutputCache();
```

3. Далее после сопоставления Razor Pages создайте две простые конечные точки, отвечающие обычным текстом, одна из которых не кэшируется, а другая использует выходной кэш данных:

```
app.MapGet("/notcached", () => DateTime.Now.ToString());
app.MapGet("/cached", () => DateTime.Now.ToString()).CacheOutput();
```

4. В файле `appsettings.Development.json` укажите уровень журналирования для промежуточного ПО кэширования выходных данных как `Information`:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.OutputCaching": "Information"
    }
  }
}
```

5. Запустите проект сайта `Northwind.Mvc` и расположите окна браузера и терминала рядом друг с другом.
6. Перейдите в браузере по адресу `https://localhost:5141/notcached`. Обратите внимание, что в терминале ничего не фиксируется.

7. Обновите страницу несколько раз — время будет обновляться, так как кэширование отсутствует.
8. Перейдите по адресу `https://localhost:5141/cached`. В терминале появится сообщение о запросе к кэшированной конечной точке. Так как кэш пуст, создается новый кэшированный ответ:

```
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[7]
      No cached response available for this request.
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[9]
      The response has been cached.
```

9. Обновите страницу несколько раз — время не изменится, так как данные загружаются из кэша:

```
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[5]
      Serving response from cache.
```

10. Продолжайте обновлять страницу, пока не пройдет 10 секунд. После этого кэш будет обновлен и в терминале появится соответствующее сообщение.
11. Закройте браузер и завершите работу веб-сервера.

Кэширование выходных данных представления MVC

Теперь рассмотрим кэширование выходных данных представления MVC.

1. В файле `Program.cs` в конце блока сопоставления контроллеров добавьте вызов метода `CacheOutput`:

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}")
  .CacheOutput();
```

2. Запустите проект сайта `Northwind.Mvc` и расположите терминал и окно браузера так, чтобы были видны оба окна.
3. В терминале можно увидеть, что главная страница с таблицей заказов отсутствует в кэше. Для получения данных выполняются SQL-запросы, а затем сгенерированная Razor-страница сохраняется в кэше:

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5141
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5142
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: C:\apps-services-net8\Chapter14\Northwind.Mvc
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[7]
      No cached response available for this request.
dbug: 6/16/2023 10:40:15.252 RelationalEventId.CommandExecuting[20100]
(Microsoft.EntityFrameworkCore.Database.Command)
      Executing DbCommand [Parameters=[], CommandType='Text',
CommandTimeout='30']
      SELECT [o].[OrderId], [o].[CustomerId], [o].[EmployeeId], [o].[Freight],
[o].[OrderDate], [o].[RequiredDate], [o].[ShipAddress], [o].[ShipCity], [o].
[ShipCountry], [o].[ShipName], [o].[ShipPostalCode], [o].[ShipRegion], [o].
[ShipVia], [o].[ShippedDate], [c].[CustomerId], [c].[Address], [c].[City], [c].
[CompanyName], [c].[ContactName], [c].[ContactTitle], [c].[Country], [c].[Fax],
[c].[Phone], [c].[PostalCode], [c].[Region], [o0].[OrderId], [o0].[ProductId],
[o0].[Discount], [o0].[Quantity], [o0].[UnitPrice]
      FROM [Orders] AS [o]
      LEFT JOIN [Customers] AS [c] ON [o].[CustomerId] = [c].[CustomerId]
      LEFT JOIN [Order Details] AS [o0] ON [o].[OrderId] = [o0].[OrderId]
      ORDER BY [o].[OrderId], [c].[CustomerId], [o0].[OrderId]
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (32ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      SELECT [o].[OrderId], [o].[CustomerId], [o].[EmployeeId], [o].[Freight],
[o].[OrderDate], [o].[RequiredDate], [o].[ShipAddress], [o].[ShipCity], [o].
[ShipCountry], [o].[ShipName], [o].[ShipPostalCode], [o].[ShipRegion], [o].
[ShipVia], [o].[ShippedDate], [c].[CustomerId], [c].[Address], [c].[City], [c].
[CompanyName], [c].[ContactName], [c].[ContactTitle], [c].[Country], [c].[Fax],
[c].[Phone], [c].[PostalCode], [c].[Region], [o0].[OrderId], [o0].[ProductId],
[o0].[Discount], [o0].[Quantity], [o0].[UnitPrice]
      FROM [Orders] AS [o]
      LEFT JOIN [Customers] AS [c] ON [o].[CustomerId] = [c].[CustomerId]
      LEFT JOIN [Order Details] AS [o0] ON [o].[OrderId] = [o0].[OrderId]
      ORDER BY [o].[OrderId], [c].[CustomerId], [o0].[OrderId]
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[8]
      The response has been cached.
```

4. На главной странице обратите внимание на текущее время, затем обновите страницу. Вся страница, включая таблицу заказов, загружается из кэша:

```
info: Microsoft.AspNetCore.OutputCaching.OutputCacheMiddleware[5]
      Serving response from cache.
```

5. Продолжайте обновлять страницу. Через 10 секунд данные будут загружены заново, в том числе актуальное время.
6. Закройте браузер и завершите работу веб-сервера.

Существует множество способов настройки кэширования выходных данных. Команда ASP.NET Core планирует добавить новые функции в будущих версиях.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 14.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. В чем преимущество объявления строго типизированного представления Razor и как это сделать?
2. Как использовать тег-хелперы в представлении?
3. В чем преимущества и недостатки HTML-хелперов по сравнению с тег-хелперами?
4. Как браузер запрашивает предпочтительный язык для локализации?
5. Как локализовать текст в представлении?
6. Какой префикс у атрибутов, распознаваемых тег-хелперами?
7. Как передать сложный объект в качестве параметра строки запроса?
8. Как управлять сроком кэширования с помощью элемента `cache`?
9. Для каких целей используется элемент `environment`?
10. Как выполняется кэширование с помощью тег-хелперов?

Упражнение 14.2. Создание пользовательских интерфейсов с помощью библиотеки Bootstrap

Создайте новый проект ASP.NET Core MVC и присвойте ему имя `Ch14Ex02_ExploringBootstrap`. Добавьте в него представления с поддержкой следующих компонентов Bootstrap:

- `Accordion`: https://getbootstrap.com/docs/5.3/components/accordion/;
- `Cards`: https://getbootstrap.com/docs/5.3/components/card/;
- `Carousel`: https://getbootstrap.com/docs/5.3/components/carousel/;
- `NavBar`: https://getbootstrap.com/docs/5.3/components/navbar/;
- `Popovers`: https://getbootstrap.com/docs/5.3/components/popovers/;
- `Toast`: https://getbootstrap.com/docs/5.3/components/toasts/;
- `Tooltips`: https://getbootstrap.com/docs/5.3/components/tooltips/.

Упражнение 14.3. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-14---building-web-user-interfaces-using-aspnet-core.

Резюме

В этой главе вы научились создавать пользовательские интерфейсы с помощью MVC ASP.NET Core. Мы рассмотрели:

- представления ASP.NET Core Razor и синтаксис Razor;
- основные стили Bootstrap;
- локализацию и глобализацию в ASP.NET Core;
- тег-хелперы и HTML-хелперы;
- кэширование выходных данных конечных точек и представлений.

В следующей главе речь пойдет о разработке компонентов пользовательского интерфейса с помощью фреймворка Blazor.

15 Создание веб-компонентов с помощью Blazor

Эта глава посвящена созданию веб-компонентов с использованием фреймворка Blazor, который позволяет с помощью HTML и CSS разрабатывать богатые и интерактивные пользовательские интерфейсы для обеспечения кросс-платформенной поддержки в браузерах.

Использование .NET для разработки веб-приложений на стороне клиента имеет множество преимуществ. Вы можете написать 99 % кода на C# вместо JavaScript, а для оставшихся 1 % использовать взаимодействие с модулями JavaScript. Также возможно разделение бизнес-логики между сервером и клиентом. В Blazor реализованы стандарты .NET, а также последние версии библиотеки .NET 8, что позволяет использовать обширные старые библиотеки .NET как компании Microsoft, так и сторонних разработчиков.

В предыдущем издании книги в этой главе рассматривалась *Blazor WebAssembly* — модель хостинга, при которой приложение Blazor и среда выполнения .NET загружались в браузер и выполнялись на клиентской стороне. Одна из проблем Blazor WebAssembly — медленный начальный запуск для пользователя, так как необходимо загружать и выполнять многое на клиенте.

Многие разработчики .NET были разочарованы, выбирая между различными технологиями для создания веб-приложений, так как каждая из них имеет свои достоинства и недостатки.

В этом издании глава посвящена новой унифицированной модели Blazor Full Stack, представленной в .NET 8. Используя эту модель, вы сможете сочетать лучшие технологии в одном проекте, включая следующее.

- Компоненты Blazor, которые выполняются на клиентской стороне с помощью WebAssembly. Взамен возможностям проекта Blazor WebAssembly.
- Компоненты Blazor, которые выполняются на сервере и взаимодействуют с *объектной моделью документа (Document Object Model, DOM)* в браузере, используя брокер SignalR для выполнения обновлений. Взамен возможностям проекта Blazor Server.
- Компоненты Blazor, обеспечивающие *статическую визуализацию на сервере (Static Server Rendering, SSR)*, возвращая HTTP-ответ со статическим содер-

жимым, которое не взаимодействует с сервером в реальном времени. Взамен возможностям Razor Pages или Razor Views, используемых на традиционных сайтах на платформе ASP.NET Core.

- Компоненты Blazor, поддерживающие потоковую передачу данных с сервера, отображая часть контента на сайте быстрее, а оставшуюся информацию загружать в фоновом режиме. Это совершенно новая функция.
- В дальнейшем Blazor будет поддерживать выполнение в любом процессе .NET, например в консольном приложении, что откроет возможности для использования Blazor в качестве *генератора статических сайтов (Static Site Generator, SSG)*.

Знакомство с Blazor

Blazor — это фреймворк компании Microsoft для создания веб-компонентов, основанный на .NET.

Модели размещения Blazor

В Blazor доступно несколько моделей размещения.

- Blazor Server. Все компоненты выполняются на веб-сервере, а обновления пользовательского интерфейса передаются в браузер с помощью брокера SignalR. Эта модель обеспечивает ряд ключевых преимуществ, включая полную поддержку API .NET, прямой доступ ко всем серверным ресурсам, например БД, быстрое время начальной загрузки и защиту кода, так как он выполняется исключительно на сервере. Модель была представлена с .NET Core 3.0 в ноябре 2019 года.
- Blazor WebAssembly. Все компоненты выполняются в браузере, как в других фреймворках *одностраничных приложений (single page application, SPA)*, таких как React и Angular. Сборки .NET и среда выполнения .NET загружаются в браузер и кэшируются для дальнейшего использования. Эта модель позволяет запускать приложение в автономном режиме, размещать его на статическом сайте или обслуживать через *сеть доставки контента (Content Delivery Network, CDN)*, а также передавать обработку стороне клиента, что повышает масштабируемость. Модель была представлена в .NET Core 3.1 в мае 2020 года и встроена в .NET 5 в ноябре 2020 года.
- Blazor Hybrid/.NET MAUI Blazor App. Все компоненты выполняются в локальном веб-представлении, размещенном в нативном клиентском приложении. Приложение может быть создано с использованием MAUI .NET для кроссплатформенных решений или с Windows Presentation Foundation и Windows Forms, если ориентировано на ОС Windows. Основное преимущество Blazor Hybrid по сравнению с предыдущими моделями — доступ к нативным клиентским возможностям для улучшения пользовательского опыта. Модель была представлена в .NET 7 в ноябре 2022 года.

- Blazor Full Stack. Компоненты могут выполняться на сервере и генерировать статическую разметку, а каждый компонент можно переключить на один из режимов: потоковую визуализацию, интерактивный сервер с обновлениями через SignalR или интерактивный клиент с использованием WebAssembly. Эта модель, известная ранее как Blazor United, была представлена в .NET 8 в ноябре 2023 года. В будущих версиях она будет называться просто Blazor.

Дельный совет: в новых проектах рекомендуется выбирать шаблон Blazor Web App. Если нужно создать истинное SPA-приложение для размещения в службе статических веб-приложений Azure или CDN, лучше выбрать шаблон Blazor WebAssembly Standalone App, так как Blazor Web App требует наличия веб-сервера. Для статических сайтов стоит использовать шаблон Blazor WebAssembly.

Вместо нескольких моделей размещения в Blazor Full Stack появились эквивалентные режимы визуализации. Шаблон проекта Blazor Server, который раньше размещался на сервере, теперь заменен интерактивной визуализацией на сервере. Шаблоны Blazor WebAssembly, которые можно было размещать на статических сайтах и выполнять на клиенте, теперь заменены интерактивной визуализацией WebAssembly.

Фреймворк Blazor поддерживает все основные браузеры — Chrome, Firefox, Edge и Safari — на мобильных и классических платформах. Механизм Blazor Hybrid поддерживает компоненты веб-представлений на трех платформах: Chrome в Android, Safari в iOS и macOS и Edge WebView2 в Windows.

В копилку знаний: в официальной документации Blazor есть полезная таблица для выбора модели размещения. Вы найдете ее по адресу https://learn.microsoft.com/ru-ru/aspnet/core/blazor/hosting-models#which-blazor-hosting-model-should-i-choose.

Компоненты Blazor

Фреймворк Blazor состоит из *компонентов*. Компонент — это часть веб-приложения, например кнопка, сетка, форма для сбора данных или целая страница. Компоненты можно использовать многократно и комбинировать для создания более сложных элементов.

Каждый компонент Blazor обычно представлен файлом Razor с расширением `.razor`. Эти файлы, как и в MVC или Razor Pages в ASP.NET Core, содержат HTML- и C#-код. В файле также есть блоки кода для обработки событий, свойств и других операций, обеспечивающих функциональность компонента.

Например, компонент `ProgressBar.razor` позволяет реализовать индикатор выполнения с помощью фреймворка Bootstrap, задавая параметры для минимального,

максимального и текущего значения индикатора, а также возможность отображения анимации и текущего значения в виде текста:

```
<div class="progress">
  <div class="progress-bar progress-bar-striped bg-info @(IsAnimated ?
    " progress-bar-animated" : "")"
      role="progressbar" aria-label="@LabelText" style="width: @Value%"
      aria-valuenow="@Value" aria-valuemin="@Minimum" aria-valuemax="@Maximum">
    @(ShowValue ? Value + "%" : "")
  </div>
</div>

@code {
  [Parameter]
  public int Value { get; set; } = 0;

  [Parameter]
  public int Minimum { get; set; } = 0;

  [Parameter]
  public int Maximum { get; set; } = 100;

  [Parameter]
  public bool IsAnimated { get; set; } = false;

  [Parameter]
  public bool ShowValue { get; set; } = false;

  [Parameter]
  public string? LabelText { get; set; } = "Progress bar";
}
```

Чтобы внедрить компонент на страницу, нужно указать его имя в виде HTML-элемента и настроить параметры с помощью HTML-атрибутов:

```
<ProgressBar Value="25" IsAnimated="true" ShowValue="true"
  LabelText="Progress of database deletion" />
```

Маршрутизация к страницам-компонентам

В файле `App.razor` компонент `Router` отвечает за маршрутизацию к страницам-компонентам:

```
<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData" DefaultLayout="@typeof(Layout.MainLayout)" />
    <FocusOnNavigate RouteData="@routeData" Selector="h1" />
  </Found>
</Router>
```

Компонент `Router` сканирует сборку, переданную в параметре `AppAssembly`, на наличие компонентов с атрибутом `[Route]`, регистрируя их URL-пути.

Если найдено совпадение маршрута, информация о запросе сохраняется в переменной `routeData` и передается в соответствующий `.razor`-файл. По умолчанию для разметки используется класс, определенный в файле `MainLayout.razor`.

Компонент `FocusOnNavigate` позволяет установить фокус на определенный элемент после перехода. Он использует свойство `Selector`, в котором необходимо указать корректный CSS-селектор. Это может быть:

- стандартный тег (например, `h1`);
- CSS-класс (`.класс`);
- идентификатор (`#идентификатор`).

Такой параметр должен быть универсальным для всех компонентов приложения. В `.razor`-файле рекомендуется устанавливать фокус на первый заголовок `h1`. Если страница содержит форму, фокус можно задать первому полю ввода, например текстовому.

В качестве примера: в типичных проектах ASP.NET Core MVC маршруты контроллеров MVC определяются атрибутом `[Route]`:

```
[Route("customers")]
public class CustomersController
{
```

Например, HTTP-запрос `GET /customers` по относительному пути будет сопоставлен с маршрутом. В Blazor аналогичный маршрут задается с помощью директивы `@page`, которая добавляется в начало файла компонента `.razor`:

```
@page "customers"
```

Компонент может содержать несколько директив `@page`, что позволяет регистрировать несколько маршрутов.

Если просканировать сборку на предмет класса component с помощью рефлексии, можно увидеть, что директива @page автоматически добавляет атрибут [Route] в сгенерированный класс компонента.

Во время выполнения компонент страницы объединяется с указанным макетом аналогично представлениям MVC или Razor Pages. По умолчанию в проектах Blazor используется макет `MainLayout.razor`.

Дельный совет: лучше всего придерживаться следующего подхода к размещению компонентов:

- маршрутизируемые страницы помещаются в папку Components\Pages;
- обычные компоненты — в папку Components.

Передача параметров маршрута

Фреймворк Blazor поддерживает в маршрутах именованные параметры без учета регистра. Чтобы передать значение, параметр привязывается к свойству компонента с атрибутом `[Parameter]`:

```
@page "/employees/{country}"

<div>Country parameter as the value: @Country</div>

@code {
  [Parameter]
  public string Country { get; set; }
}
```

Если параметр необязателен, в маршруте указывается символ `?`, а его обработка выполняется через оператор нулевого слияния (`??`) в методе `OnParametersSet`:

```
@page "/employees/{country?}"

<div>Country parameter as the value: @Country</div>

@code {
  [Parameter]
  public string? Country { get; set; }

  protected override void OnParametersSet()
  {
    // если автоматически установленное свойство равно null,
    // в качестве его значения задается USA
    Country = Country ?? "USA";
  }
}
```

Установка параметров из строки запроса

Свойства компонента можно задавать, передавая параметры через строку запроса:

```
[Parameter]
[SupplyParameterFromQuery(Name = "country")]
public string? Country { get; set; }
```

Ограничения маршрута

Ограничения маршрута определяют соответствие переданного параметра ожидаемому типу данных. Если параметр не удовлетворяет ограничению, маршрут не совпадает и Blazor пытается сопоставить запрос с другими маршрутами. Если ни один маршрут не подходит, возвращается код ответа `404`.

Если ограничения не заданы, маршрут принимает любые значения, но при попытке преобразования параметра в ожидаемый тип C# может возникнуть исключение. Некоторые примеры ограничений маршрута приведены в табл. 15.1.

Таблица 15.1. Примеры ограничений маршрута

Пример ограничения	Описание
`{isanimated:bool}`	Свойству `IsAnimated` должно быть присвоено допустимое логическое значение, например `TRUE` или `true`
`{hiredate:datetime}`	Свойству `HireDate` должны быть присвоены корректные дата/время
`{price:decimal}`	Свойству `UnitPrice` должно быть присвоено допустимое число с плавающей точкой типа `decimal`
`{shipweight:double}`	Свойству `ShipWeight` должно быть присвоено допустимое число типа `double`
`{shipwidth:float}`	Свойству `ShipWidth` должно быть присвоено допустимое число типа `float`
`{orderid:guid}`	Свойству `OrderId` должен быть присвоен допустимый идентификатор `Guid`
`{categoryid:int}`	Свойству `CategoryId` должно быть присвоено допустимое целое число типа `int`
`{nanoseconds:long}`	Свойству `Nanoseconds` должно быть присвоено допустимое целое число типа `long`

Дельный совет: в ограничениях маршрута используется инвариантная культура, поэтому URL-адреса не должны зависеть от локализации. Например, параметры даты и времени следует передавать в инвариантном формате.

Базовые классы компонентов

Метод `OnParametersSet` определяется базовым классом `ComponentBase`, от которого по умолчанию наследуются компоненты:

```
using Microsoft.AspNetCore.Components;

public abstract class ComponentBase : IComponent, IHandleAfterRender,
IHandleEvent
{
  // члены класса не показаны
}
```

Базовый класс `ComponentBase`, от которого наследуются компоненты Blazor, содержит несколько полезных методов, которые можно вызывать или переопределять (табл. 15.2).

Таблица 15.2. Полезные методы класса ComponentBase

Метод (-ы)	Описание
`InvokeAsync`	Выполняет функцию в связанном контексте синхронизации механизма визуализации. Благодаря этому исключается необходимость писать код для синхронизации потоков при работе с общими ресурсами. Одновременный доступ нескольких потоков к механизму визуализации невозможен. Метод `InvokeAsync` гарантирует, что в каждый момент времени к компонентам обращается только один поток, устраняя необходимость блокировки и ручной синхронизации состояния
`OnAfterRender` `OnAfterRenderAsync`	Вызываются после каждой операции визуализации компонента
`OnInitialized` `OnInitializedAsync`	Вызываются после инициализации компонента, когда он получил начальные параметры от предка в механизме визуализации
`OnParametersSet` `OnParametersSetAsync`	Вызываются при обновлении параметров компонента, когда новые значения присвоены свойствам
`ShouldRender`	Определяет, должен ли компонент визуализироваться
`StateHasChanged`	Запускает повторную визуализацию компонента, обновляя интерфейс

Макеты Blazor

Компоненты Blazor поддерживают общие макеты, как и представления в MVC или Razor Pages. Для этого нужно создать компонент с расширением `.razor` и явно указать его наследование от класса `LayoutComponentBase`:

```
@inherits LayoutComponentBase

<div>
  ...
  @Body
  ...
</div>
```

Этот базовый класс содержит свойство `Body`, которое отображает содержимое компонента в нужном месте макета.

Задать макет по умолчанию можно в файле `App.razor`, в его компоненте `Router`. Для явного указания макета в конкретном компоненте используется директива `@layout`:

```
@page "/employees"

@layout AlternativeLayout

<div>
  ...
</div>
```

Настройка маршрутов Blazor к компонентам страницы

В Blazor используется встроенный механизм маршрутизации. Для работы с ним применяется разработанный в Microsoft класс `NavigationManager`, который управляет навигацией Blazor и поддерживает компонент `NavLink`. Метод `NavigateTo` выполняет переход по указанному URL-адресу.

Для создания ссылок в HTML применяется элемент `a`:

```
<a href="/employees">Employees</a>
```

В Blazor его заменяет компонент `NavLink`:

```
<NavLink href="/employees">Employees</NavLink>
```

В отличие от элемента `a` компонент `NavLink` автоматически присваивает CSS-классу значение `active`, если его параметр `href` совпадает с текущим URL-адресом. При необходимости класс можно переопределить, задав его в свойстве `ActiveClass`. Если в таблице CSS используется класс с другим именем, вы можете задать его в свойстве `NavLink.ActiveClass`.

По умолчанию алгоритм сопоставления обрабатывает `href` как префикс пути, поэтому, если компоненту `NavLink` присвоен параметр `href="/employees"`, он будет активен для всех путей, начинающихся с `/employees`. Также к ним будут применяться стили активного класса:

```
/employees
/employees/USA
/employees/UK/London
```

Чтобы алгоритм сопоставления обрабатывал путь целиком (то есть сопоставлялся весь путь, а не только некая его часть), присвойте параметру `Match` значение `NavLinkMatch.All`:

```
<NavLink href="/employees" Match="NavLinkMatch.All">Employees</NavLink>
```

Все дополнительные атрибуты (например, `target`) передаются в итоговый HTML-элемент `a`.

Изоляция CSS и JavaScript

Компоненты Blazor могут использовать собственные стили CSS и код JavaScript, которые нельзя выполнить исключительно на языке C#, например, для работы с API браузера. Чтобы избежать конфликтов с глобальными стилями и скриптами, Blazor поддерживает изоляцию CSS и JavaScript.

Для изоляции CSS достаточно создать файл `ИмяКомпонента.razor.css` (например, `Home.razor.css`). Стили из этого файла применяются (и переопределяют их) только к соответствующему компоненту и не затрагивают остальной сайт.

Изоляция JavaScript работает иначе: в Blazor используются JavaScript-модули, загружаемые через механизм *JavaScript* interop (взаимодействие с JavaScript). Далее в этой главе мы рассмотрим данный подход.

Подробнее об изоляции JavaScript-кода можно прочитать по адресу https://learn.microsoft.com/ru-ru/aspnet/core/blazor/javascript-interoperability/call-javascript-from-dotnet#javascript-isolation-in-javascript-modules.

Создание компонентов Blazor

В ASP.NET Core 8 для Blazor представлен новый шаблон проекта, который поддерживает все режимы визуализации и обеспечивает гибкость в выборе способа размещения. Этот шаблон включает базовую структуру проекта и компонент `Weather`, визуализирующий в потоке таблицу с пятью строками случайных значений температуры.

Создание проекта Blazor из шаблона

Рассмотрим процесс создания нового проекта Blazor Web App и его ключевые компоненты.

1. В редакторе кода создайте новый проект и решение на основе шаблона Blazor Web App:
   - выберите шаблон проекта Blazor Web App / `blazor --interactivity` None;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Blazor`;
   - в поле Solution name (Имя решения) укажите значение `Chapter15`;
   - в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
   - установите флажок Configure for HTTPS (Настроить для HTTPS);
   - в раскрывающемся списке Interactive render mode (Интерактивный режим визуализации) выберите пункт None (Отсутствует);
   - в раскрывающемся списке Interactivity location (Расположение интерактивных компонентов) выберите пункт Per page/component (На странице/компоненте);
   - установите флажок Include sample pages (Включить примеры страниц);
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня).

Для создания проекта в программе Visual Studio Code или JetBrains Rider откройте в терминале папку Chapter15 и выполните команду dotnet new blazor --interactivity None -o Northwind.Blazor.

Дельный совет: мы намеренно не выбрали параметры для интерактивного WebAssembly или серверных компонентов, чтобы вы могли поэтапно разобраться в работе Blazor. В реальном проекте их, скорее всего, стоит включить сразу. Кроме того, некоторые демонстрационные страницы можно будет удалить, поскольку они не предназначены для рабочей версии.

2. Выполните сборку проекта `Northwind.Blazor`.
3. Откройте файл `Northwind.Blazor.csproj` и обратите внимание, что проект основан на инструментарии Web SDK и ориентирован на .NET 8, поэтому он аналогичен другим ASP.NET Core-приложениям.
4. В файле `Program.cs` проекта `Northwind.Blazor` определены сервисы ASP.NET Core, HTTP-конвейер и специфичные для Blazor механизмы, такие как регистрация Razor-компонентов:

```
using Northwind.Blazor.Components;

var builder = WebApplication.CreateBuilder(args);

// Добавление сервисов в контейнер
builder.Services.AddRazorComponents();

var app = builder.Build();

// Настройка конвейера HTTP-запросов
if (!app.Environment.IsDevelopment())
{
  app.UseExceptionHandler("/Error", createScopeForErrors: true);
  // Значение HSTS по умолчанию составляет 30 дней.
  // Вы можете изменить это значение в рабочих сценариях,
  // см. https://aka.ms/aspnetcore-hsts
  app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles();
app.UseAntiforgery();

app.MapRazorComponents<App>();

app.Run();
```

5. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5151 для протокола HTTPS и 5152 — для HTTP:

```
"applicationUrl": "https://localhost:5151;http://localhost:5152",
```

6. Сохраните изменения в файле `launchSettings.json`.

7. В папке `Components` проекта `Northwind.Blazor` откройте файл `App.razor`. В нем должна присутствовать следующая разметка:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <base href="/" />
  <link rel="stylesheet" href="bootstrap/bootstrap.min.css" />
  <link rel="stylesheet" href="app.css" />
  <link rel="stylesheet" href="Northwind.Blazor.styles.css" />
  <link rel="icon" type="image/png" href="favicon.png" />
  <HeadOutlet />
</head>

<body>
  <Routes />
  <script src="_framework/blazor.web.js"></script>
</body>

</html>
```

Обратите внимание на следующие ключевые моменты.

- Компонент `<HeadOutlet />` фреймворка Blazor позволяет вставлять дополнительное содержимое в шапку страницы (элемент `head`). Данный встроенный компонент доступен во всех проектах Blazor.
- Компонент `<Routes />` отвечает за маршрутизацию. Он настраивается разработчиком в файле `Routes.razor`.
- Сценарий `blazor.web.js` управляет связью с сервером и динамическими функциями, включая загрузку компонентов WebAssembly в фоновом режиме и переключение между обработкой компонентов на стороне сервера и на стороне клиента.

8. В папке `Components` откройте файл `Routes.razor`. В этом файле элемент `Router` определяет маршрутизацию для всех компонентов Blazor в текущей сборке. Если найден соответствующий маршрут, метод `RouteView` загружает компонент, применяя к нему макет `MainLayout`, и передает параметры маршрута. Первый элемент `h1` в компоненте получает фокус:

```
<Router AppAssembly="@typeof(Program).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData"
      DefaultLayout="@typeof(Layout.MainLayout)" />
    <FocusOnNavigate RouteData="@routeData" Selector="h1" />
  </Found>
</Router>
```

9. В папке `Components` откройте файл `_Imports.razor`. Этот файл импортирует пространства имен, которые автоматически подключаются во всех пользовательских компонентах. Это упрощает работу с Blazor.
10. В папке `Components\Layout` откройте файл `MainLayout.razor`. Он определяет разметку с элементами `div`, формирующими боковую панель навигации, которая в этом проекте реализуется компонентом `NavMenu.razor`. Также в нем содержатся HTML5-элементы `main` и `article` для разметки содержимого:

```
@inherits LayoutComponentBase

<div class="page">
  <div class="sidebar">
    <NavMenu />
  </div>

  <main>
    <div class="top-row px-4">
      <a href="https://learn.microsoft.com/aspnet/core/" target="_blank">About</a>
    </div>

    <article class="content px-4">
      @Body
    </article>
  </main>
</div>
```

11. В папке `Components\Layout` откройте файл `NavMenu.razor`:

```
<div class="top-row ps-3 navbar navbar-dark">
  <div class="container-fluid">
    <a class="navbar-brand" href="">Northwind.Blazor</a>
  </div>
</div>

<input type="checkbox" title="Navigation menu" class="navbar-toggler" />

<div class="nav-scrollable"
     onclick="document.querySelector('.navbar-toggler').click()">
  <nav class="flex-column">
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
        <span class="bi bi-house-door-fill-nav-menu"
          aria-hidden="true"></span> Home
      </NavLink>
    </div>
```

```
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="weather">
        <span class="bi bi-list-nested-nav-menu"
          aria-hidden="true"></span> Weather
      </NavLink>
    </div>
  </nav>
</div>
```

Этот компонент отвечает за навигацию. Особенности:

- компонент `NavMenu` не содержит директиву `@page`, так как не является страницей и не использует общий макет;
- для адаптивного меню используется Bootstrap. Такое меню гибко адаптируется к ширине области просмотра. Если горизонтального пространства недостаточно, меню сворачивается в гамбургер-кнопку, с помощью которой посетитель может свернуть/развернуть навигацию;
- в шаблоне уже есть два пункта меню: Home и Weather. Мы добавим дополнительные пункты далее в этой главе.

12. В папке `Components\Pages` откройте файл `Home.razor`. Он содержит директиву `@page`, определяющую маршрут корневой страницы.

    Важно: измените заголовок страницы с `world` на `Blazor Full Stack`:

```
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, Blazor Full Stack!</h1>

Welcome to your new app.
```

13. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.
    - В Visual Studio 2022 на панели Solution Explorer (Обозреватель решений) выберите проект `Northwind.Blazor`. На панели инструментов выберите профиль `https` в раскрывающемся списке Startup Project (Запускаемый проект) и браузер Google Chrome — в раскрывающемся списке Web Browser (Браузер).
    - В Visual Studio Code в терминале выполните следующую команду:

```
dotnet run --launch-profile https
```

14. В Google Chrome обратите внимание на навигацию в левой части экрана и компонент главной страницы (рис. 15.1).

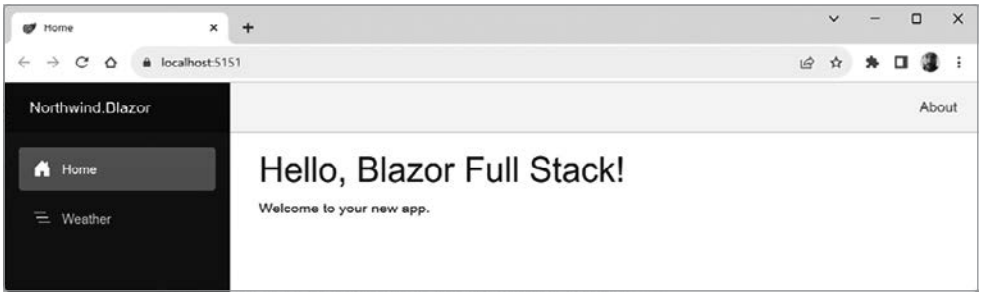


Рис. 15.1. Простая веб-страница, реализованная как Blazor-компонент

15. Закройте браузер и завершите работу веб-сервера.

Подключение значков Bootstrap

Старые шаблоны Blazor содержали полный набор значков Bootstrap. В новых шаблонах доступны только три значка в формате *масштабируемой векторной графики (Scalable Vector Graphics, SVG)*. Посмотрим, как их определили разработчики, а также добавим дополнительные значки.

1. В папке `Components\Layout` откройте файл таблицы стилей `NavMenu.razor.css`. Найдите в нем правило `bi-house`. Вы увидите код со ссылками на три SVG-значка. Фрагмент этого CSS-файла показан ниже:

```
.bi-house-door-fill-nav-menu {background-image: url("data:image/svg+xml,...");
}

.bi-plus-square-fill-nav-menu {background-image: url("data:image/svg+xml,...");
}

.bi-list-nested-nav-menu {background-image: url("data:image/svg+xml,...");
}
```

2. В браузере перейдите по адресу https://icon-sets.iconify.design/bi/. Библиотека Bootstrap Icons доступна по лицензии с открытым исходным кодом MIT и содержит свыше 2 тысяч значков.
3. В поле Search Bootstrap Icons (Поиск значков Bootstrap) введите слово `globe`. В результатах окажется шесть значков с изображением глобуса.
4. Щелкните кнопкой мыши на первом значке с изображением глобуса, прокрутите страницу вниз и перейдите на вкладку SVG. Вы можете скопировать определение этого значка и вставить в свою таблицу стилей CSS, но не торопитесь. Я подготовил специальный CSS-файл с пятью значками, которые можно использовать в проектах Blazor.
5. Перейдите по адресу https://github.com/markjprice/apps-services-net8/blob/main/code/Chapter15/Northwind.Blazor/wwwroot/icons.css, скачайте файл и сохраните его в своем проекте в папке `wwwroot`.

6. В папке `Components` откройте файл `App.razor` и в разделе заголовка (элемент `head`) добавьте элемент `link` со ссылкой на таблицу стилей `icons.css`:

```
<link rel="stylesheet" href="icons.css" />
```

7. Сохраните изменения и закройте файл.

Создание ссылок на библиотеку классов EF Core и регистрация контекста данных

Мы будем работать с моделью EF Core, созданной в главе 3.

1. В файле проекта `Northwind.Blazor.csproj` добавьте строгую обработку предупреждений и ссылку на проект контекста БД Northwind:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.DataContext.
SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
</ItemGroup>
```

Убедитесь, что путь Include не содержит символов разрыва строки.

2. В терминале выполните сборку проекта `Northwind.Blazor` с помощью команды `dotnet build`.
3. В папке `Components` откройте файл `_Imports.razor` и добавьте пространство имен, которое понадобится для работы с асинхронными методами EF Core и моделями сущностей Northwind:

```
@using Microsoft.EntityFrameworkCore @* To use ToListAsync method. *@
@using Northwind.EntityModels @* To use NorthwindContext and so on. *@
```

Поскольку мы импортируем пространства имен в этом файле, нам не нужно добавлять операторы импорта в каждый .razor-файл. Код из файла _Imports.razor применим только к .razor-файлам. В .cs-файлах с кодом (code-behind) операторы импорта нужно добавлять либо можно воспользоваться глобальным импортом для неявного использования. Важно также отметить, что для статического импорта типа визуализации следует использовать выражение @using static Microsoft.AspNetCore.Components.Web.RenderMode.

4. В файле `Program.cs` импортируйте пространство имен, необходимое для использования метода расширения `AddNorthwindContext`:

```
using Northwind.EntityModels; // Для метода AddNorthwindContext
```

5. В разделе, где происходит регистрация сервисов в контейнере, добавьте следующую строку для регистрации `NorthwindContext` в качестве сервиса:

```
builder.Services.AddNorthwindContext();
```

Статический Blazor-компонент с визуализацией данных на стороне сервера

Теперь добавим компонент, который будет выполнять ту же роль, что и страница или представление Razor на традиционном сайте ASP.NET Core. Этот компонент статичен и не требует интерактивных действий для выполнения визуализации, а лишь отображает таблицу товаров из БД Northwind.

1. В папке `Components\Pages` создайте файл `Products.razor`. В Visual Studio 2022 шаблон проекта называется Razor Component (Компонент Razor), а в JetBrains Rider — Blazor Component.

Дельный совет: имена файлов компонентов должны начинаться с прописной буквы! В противном случае могут возникнуть ошибки компиляции.

2. В файле `Products.razor` установите маршрут `/products`, добавьте контекст данных Northwind, создайте таблицу для отображения товаров и добавьте код для получения списка товаров после инициализации страницы:

```
@page "/products"
@inject NorthwindContext db
<h1>Products</h1>
<table class="table">
  <thead>
    <tr>
      <th>Product ID</th>
      <th>Product Name</th>
      <th>Unit Price</th>
    </tr>
  </thead>
  <tbody>
    @if ((products is null) || (products.Count == 0))
    {
      <tr><td colspan="4">No products found.</td></tr>
    }
    else
    {
      @foreach (Product p in products)
      {
        <tr>
          <td>@p.ProductId</td>
          <td>@p.ProductName</td>
          <td>@(p.UnitPrice.HasValue ?
            p.UnitPrice.Value.ToString("C") : "n/a")</td>
        </tr>
      }
    }
  </tbody>
</table>
```

```
@code {
  private List<Product>? products;

  protected override async Task OnInitializedAsync()
  {
    products = await db.Products.ToListAsync();
  }
}
```

3. В папке `Components\Layout` откройте файл `NavMenu.razor` и добавьте пункт меню для перехода на страницу товаров после ссылки на главную страницу:

```
<div class="nav-item px-3">
  <NavLink class="nav-link" href="products">
    <span class="bi bi-globe" aria-hidden="true"></span> Products
  </NavLink>
</div>
```

4. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
5. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.
6. На панели навигации слева щелкните на ссылке Products и убедитесь, что таблица товаров отображается корректно.
7. Закройте браузер и завершите работу веб-сервера.

Blazor-компонент с интерактивностью на стороне сервера

Далее мы добавим интерактивный компонент, поэтому применим Blazor с SignalR для динамического обновления модели DOM браузера в реальном времени.

1. В папке `Components` создайте файл `Counter.razor`.
2. Определите в нем метку для отображения текущего значения счетчика и кнопку для его увеличения, а также блок кода для хранения текущего значения счетчика с обработчиком события щелчка кнопкой мыши, который будет увеличивать значение:

```
<h3>Counter: @CounterValue</h3>
<button id="buttonIncrement" @onclick="IncrementCounter"
  class="btn btn-outline-primary">Increment</button>

@code {
  public int CounterValue { get; set; } = 0;

  public void IncrementCounter()
  {
    CounterValue++;
  }
}
```

Этот компонент не будет использоваться как страница, поэтому не нужно помечать его директивой @page и определять маршрут. Он будет встроен в другой компонент.

3. В папке `Components\Pages` откройте файл `Home.razor`. В нижней части страницы добавьте визуализируемый компонент счетчика:

```
<Counter />
```

4. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.
5. На главной странице нажмите кнопку и обратите внимание, что ничего не происходит. При создании проекта Blazor Web App с параметром `--interactivity None` и/или отключенным на сервере интерактивным режимом визуализации интерактивность компонентов будет отключена в файле проекта Blazor Web App.
6. Закройте браузер и завершите работу веб-сервера.
7. В файле `Program.cs` добавьте вызов метода для включения интерактивных компонентов на стороне сервера в конце строки кода, реализующей компоненты Razor:

```
builder.Services.AddRazorComponents()
  .AddInteractiveServerComponents();
```

8. В том же файле, в конце строки, которая сопоставляет компоненты Razor, добавьте вызов метода для включения интерактивной визуализации на стороне сервера:

```
app.MapRazorComponents<App>()
  .AddInteractiveServerRenderMode();
```

9. В начале файла `Counter.razor` (папка `Components`) добавьте директиву для включения интерактивной визуализации на сервере:

```
@rendermode InteractiveServer
```

10. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.
11. Откройте в браузере панель Developer Tools (Инструменты разработчика) и перейдите на вкладку Console (Консоль). Убедитесь, что файл `blazor.web.js` устанавливает соединение через протокол WebSocket:

```
[2023-10-20T11:25:52.498Z] Information: Normalizing '_blazor' to 'https://
localhost:5151/_blazor'.
[2023-10-20T11:25:52.675Z] Information: WebSocket connected to wss://
localhost:5151/_blazor?id=j6Fc0Mbay_jWkZTWfIqs_w.
```

12. На панели Developer Tools (Инструменты разработчика) перейдите на вкладку Network (Сеть), нажмите кнопку WS, чтобы отфильтровать соединения по протоколу WebSockets, и обновите главную страницу.

13. На главной странице нажмите кнопку Increment, обратите внимание на увеличение счетчика. Обратите также внимание на запрос `_blazor?id=...`. Выберите его и перейдите на вкладку `Initiator`. Вы увидите, что инициатором запроса является файл `blazor.web.js`, добавляемый на всех страницах через `App.razor` (рис. 15.2).

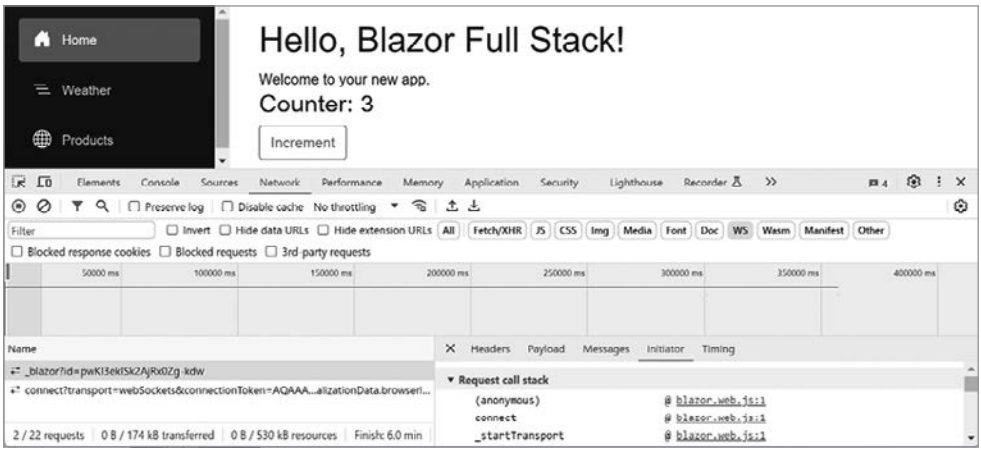


Рис. 15.2. Файл blazor.web.js запрашивает SignalR на стороне сервера для обновления модели DOM и интерактивности в реальном времени

14. Закройте браузер и завершите работу веб-сервера.

Blazor-компонент индикатора выполнения

В этом разделе мы создадим компонент для отображения индикатора выполнения. Для этого воспользуемся классами библиотеки Bootstrap для настройки светло-голубого цвета с полосками и анимацией, а также отображения текущего прогресса в процентах.

1. В папке `Components` проекта `Northwind.Blazor` создайте файл `ProgressBar.razor`.
2. В созданном файле добавьте разметку с элементами `div` и классы Bootstrap для создания индикатора выполнения с привязанными параметрами, задающими его свойства:

```
@rendermode InteractiveServer
<div class="progress">
  <div class="progress-bar progress-bar-striped bg-info
      @(IsAnimated ? " progress-bar-animated" : "")"
      role="progressbar" aria-label="@LabelText"
      style="width: @Value%" aria-valuenow="@Value"
      aria-valuemin="@Minimum" aria-valuemax="@Maximum">
    @(ShowValue ? Value + "%" : "")
  </div>
</div>
```

```
@code {
  [Parameter]
  public int Value { get; set; } = 0;

  [Parameter]
  public int Minimum { get; set; } = 0;

  [Parameter]
  public int Maximum { get; set; } = 100;

  [Parameter]
  public bool IsAnimated { get; set; } = false;

  [Parameter]
  public bool ShowValue { get; set; } = false;

  [Parameter]
  public string? LabelText { get; set; } = "Progress bar";
}
```

3. В папке `Components\Pages` откройте файл `Home.razor`. В конце файла добавьте строку с двумя одинаковыми столбцами и компонент `ProgressBar` со значением `25%`:

```
<div class="row">
  <div class="col">
    <div class="alert alert-info">
      <h4>Progress of database deletion</h4>
      <ProgressBar Value="25" IsAnimated="true" ShowValue="true"
        LabelText="Progress of database deletion" />
    </div>
  </div>
  <div class="col">
    More components coming soon.
  </div>
</div>
```

4. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.
5. Обратите внимание на индикатор выполнения, имитирующий процесс удаления базы данных.
6. Закройте браузер и завершите работу веб-сервера.

Blazor-компонент диалогового окна

В этом разделе мы создадим компонент, добавляющий всплывающее диалоговое окно для взаимодействия с посетителями сайта. С помощью классов Bootstrap мы создадим кнопку, при нажатии которой будет открываться диалоговое окно с двумя кнопками с произвольным текстом.

По умолчанию в шаблонах Blazor Web App используется локальная копия Bootstrap 5.1, но только в части CSS. Чтобы добавить необходимый JavaScript-функционал, нужно добавить соответствующий элемент `script`. Мы также можем обновить Bootstrap до последней версии, используя CDN.

CDN-ссылки на актуальную версию библиотеки опубликованы по адресу https://getbootstrap.com/docs/5.3/getting-started/introduction/#cdn-links.

Компонент будет включать два обратных вызова событий, которые родительский компонент сможет обработать для настройки кода, выполняемого при нажатии кнопок.

1. В файле `App.razor` закомментируйте элемент `link`, который подключает локальный CSS-файл, и добавьте CDN-ссылку на актуальную версию библиотеки:

```
@*<link rel="stylesheet" href="bootstrap/bootstrap.min.css" />*@
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/
dist/css/bootstrap.min.css" integrity="sha384-T3c6CoIi6uLrA9TneNEoa7
RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN" crossorigin="anonymous">
```

2. В том же файле после элемента `script` со ссылкой на Blazor добавьте элемент `script` с CDN-ссылкой на актуальную версию библиотеки и обработайте ошибку `RZ/BL9992`:

```
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/
dist/js/bootstrap.bundle.min.js" integrity="sha384-C6RzsynM9kWDr
MNeT87bh95OGNyZPhcTNXj1NW7RuBCsyN/o0jlpcV8Qyq46cDfL" crossorigin="anonymous"
suppresserror="BL9992"></script>
```

Ошибка BL9992 (или RZ9992) информирует, что «теги сценариев не следует размещать внутри компонентов, так как они не могут обновляться динамически». Подробнее об этом можно прочитать по ссылке https://learn.microsoft.com/ru-ru/aspnet/core/blazor/javascript-interoperability/.

3. В папке `Components` проекта `Northwind.Blazor` создайте файл `DialogBox.razor`.
4. В созданном файле добавьте разметку с элементами `div`, использующими классы Bootstrap для создания кнопки и модального диалогового окна с параметрами, определяющими различные свойства:

```
@rendermode InteractiveServer
<!-- Кнопка для отображения диалогового окна -->
<button type="button" class="btn btn-primary" data-bs-toggle="modal"
    data-bs-target="#dialogBox">
  @DialogTitle
</button>
```

```
<!-- Открывающееся диалоговое окно -->
<div class="modal fade" id="dialogBox" data-bs-backdrop="static"
    data-bs-keyboard="false" tabindex="-1" aria-labelledby="dialogBoxLabel"
    aria-hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="dialogBoxLabel">@DialogTitle</h5>
        <button type="button" class="btn-close" data-bs-dismiss="modal"
            aria-label="Close"></button>
      </div>
      <div class="modal-body">
        @ChildContent
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-primary" @onclick="OnClickPrimary">
          @PrimaryButtonText
        </button>
        <button type="button" class="btn btn-secondary" data-bs-dismiss="modal"
            @onclick="OnClickSecondary">
          @SecondaryButtonText
        </button>
      </div>
    </div>
  </div>
</div>

@code {
  [Parameter]
  public string? DialogTitle { get; set; }

  // ChildContent — это специальное имя, автоматически задаваемое
  // содержимым разметки в начальном и конечном элементах компонента
  [Parameter]
  public RenderFragment? ChildContent { get; set; }

  [Parameter]
  public string? PrimaryButtonText { get; set; } = "OK";

  [Parameter]
  public EventCallback<MouseEventArgs> OnClickPrimary { get; set; }

  [Parameter]
  public string? SecondaryButtonText { get; set; } = "Cancel";

  [Parameter]
  public EventCallback<MouseEventArgs> OnClickSecondary { get; set; }
}
```

Обе кнопки имеют текст по умолчанию OK и Cancel, а также параметры для обработки событий, в качестве аргументов которых передается информация о позиции указателя мыши. Обратите также внимание на кнопку с параметром class="btn-close", которая отображается в виде символа × в правом верхнем углу и позволяет закрыть диалоговое окно.

5. В папке `Components\Pages` откройте файл `Home.razor`. В начале файла добавьте оператор для выбора интерактивного режима визуализации на стороне сервера:

```
@rendermode InteractiveServer
```

6. В конце того же файла замените текст `More components coming soon` на код для добавления компонента `DialogBox`, который меняет текст кнопок на слова `Yes` и `No`. Ниже добавьте Razor-код обработчиков событий для кнопок, определяющих нажатую кнопку и позицию указателя мыши, как показано в следующем примере:

```
  <div class="col">
    <DialogBox DialogTitle="Delete Database" PrimaryButtonText="Yes"
        OnClickPrimary="Yes_Click" SecondaryButtonText="No"
        OnClickSecondary="No_Click">
      Are you sure you want to delete the entire database? Really?
    </DialogBox>
  </div>
</div>

@code {
  private void Yes_Click(MouseEventArgs e)
  {
    Console.WriteLine("User clicked 'Primary' button at ({0}, {1}).",
      arg0: e.ClientX, arg1: e.ClientY);
  }

  private void No_Click(MouseEventArgs e)
  {
    Console.WriteLine("User clicked 'Secondary' button at ({0}, {1}).",
      arg0: e.ClientX, arg1: e.ClientY);
  }
}
```

7. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.

Все содержимое между тегами <DialogBox> и </DialogBox> будет автоматически определено как свойство ChildContent.

8. Нажмите кнопку Delete Database и обратите внимание на появившееся модальное диалоговое окно (рис. 15.3).

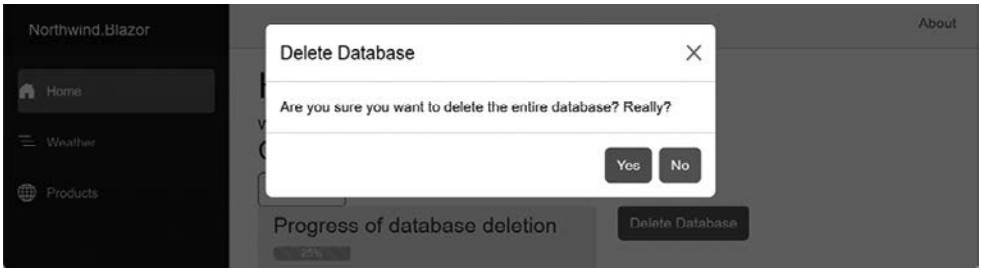


Рис. 15.3. Всплывающее модальное диалоговое окно, созданное компонентом Blazor с помощью Bootstrap

9. Расположите терминал и окно браузера так, чтобы были видны оба окна.
10. В диалоговом окне Delete Database нажимайте кнопки Yes и No (при нажатии кнопки No или × диалоговое окно закроется. Чтобы снова увидеть его, нужно вновь нажать кнопку Delete Database). Обратите внимание на сообщения, записанные в консоль (рис. 15.4).

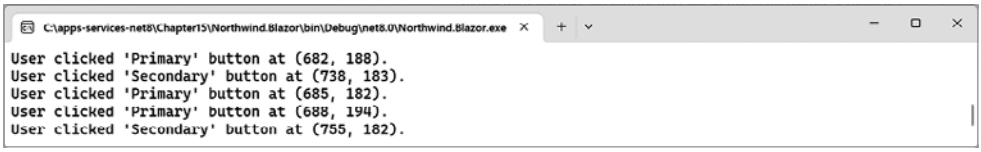


Рис. 15.4. Компонент диалогового окна, записывающий данные в консоль сервера

JavaScript-сценарий на стороне клиента управляет отображением диалогового окна Bootstrap. Нажатие кнопок инициирует соединение с брокером SignalR через WebSockets для выполнения обработки событий на сервере.

11. Закройте браузер и завершите работу веб-сервера.

Подробнее о поддерживаемых аргументах событий можно прочитать по адресу https://learn.microsoft.com/ru-ru/aspnet/core/blazor/components/event-handling#-event-arguments.

Blazor-компонент уведомлений

Теперь мы создадим компонент уведомлений, который будет отображать сообщения посетителям сайта. В компоненте используются классы Bootstrap для создания красочной области уведомления с возможностью его закрытия. Мы сможем настроить сообщение, заголовок, значок и цветовую тему.

1. В папке `Components` проекта `Northwind.Blazor` создайте файл `Bootstrap.Constants.cs`.
2. В этом файле добавьте определения статических классов с константами строк для популярных цветовых тем и значков Bootstrap:

```
namespace Northwind.Blazor.Components;

public static class BootstrapColors
{
  public const string Primary = "primary";
  public const string Secondary = "secondary";
  public const string Danger = "danger";
  public const string Warning = "warning";
  public const string Success = "success";
  public const string Info = "info";
}

public static class BootstrapIcons
{
  public const string Globe = "bi bi-globe";
  public const string GlobeEmea = "bi bi-globe-europe-africa";
  public const string Pencil = "bi bi-pencil";
  public const string Trash = "bi bi-trash";
  public const string PlusSquare = "bi bi-plus-square";
  public const string InfoCircle = "bi bi-info-circle";
  public const string ExclamationTriangleFill = "bi bi-exclamation-triangle-fill";
}
```

3. В папке `Components` создайте файл `Alert.razor`.
4. В созданном файле добавьте разметку для визуализации элементов `div`, которые будут использовать классы Bootstrap для отображения уведомлений с настраиваемыми параметрами:

```
@rendermode InteractiveServer
<div class="alert alert-@ColorTheme d-flex align-items-center
    @(IsDismissable ? " alert-dismissible fade show" : "")" role="alert">
  <div>
    <h4 class="alert-heading"><span class="@Icon" aria-hidden="true">
      </span> @Title</h4>
    @Message
    @if (IsDismissable)
    {
      <button type="button" class="btn-close" data-bs-dismiss="alert"
        aria-label="Close"></button>
    }
  </div>
</div>

@code {
  [Parameter]
  public bool IsDismissable { get; set; } = true;

  [Parameter]
  public string ColorTheme { get; set; } = BootstrapColors.Primary;
```

```
    [Parameter]
    public string Icon { get; set; } = BootstrapIcons.InfoCircle;

    [Parameter]
    public string? Title { get; set; }

    [Parameter]
    public string? Message { get; set; }
  }
```

5. В папке `Components\Pages` откройте файл `Home.razor` и добавьте в него компонент `Alert` (ниже элемента `DialogBox`):

```
<Alert IsDismissable="true"
      Icon="@(BootstrapIcons.ExclamationTriangleFill)"
      ColorTheme="@(BootstrapColors.Warning)"
      Title="Warning"
      Message="Deleting the database cannot be undone." />
```

6. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.
7. На главной странице отобразится уведомление с кнопкой × (рис. 15.5).

Рис. 15.5. Компонент уведомления с кнопкой ×

8. Нажмите кнопку ×, чтобы скрыть уведомление.
9. Закройте браузер и завершите работу веб-сервера.

Создание компонента данных Blazor

В этом разделе мы создадим компонент, который позволит перечислять, создавать и редактировать сотрудников в базе данных Northwind. Этот процесс будет состоять из нескольких этапов.

1. Создадим Blazor-компонент, отображающий имя сотрудника, переданное в качестве параметра.

2. Сделаем компонент маршрутизируемым, чтобы он мог работать как страница.
3. Реализуем веб-сервис с использованием минимальных API в ASP.NET Core.
4. В компоненте вызовем этот веб-сервис для получения и обработки данных.

Создание компонента

Добавим новый компонент в наш проект Blazor.

1. В папке `Components\Pages` проекта `Northwind.Blazor` создайте файл `Employees.razor`.
2. Добавьте в него выражения кода для вывода заголовка компонента и создайте свойство для хранения названия страны:

```
@rendermode InteractiveServer
<h1>Employees @(string.IsNullOrWhiteSpace(Country)
  ? "Worldwide" : "in " + Country)</h1>

@code {
  [Parameter]
  public string? Country { get; set; }
}
```

3. В папке `Components\Pages` откройте файл `Home.razor`. После приветственного сообщения создайте два экземпляра компонента `Employees`: один раз со значением `USA` в качестве параметра страны, а второй — без указания страны:

```
<h1>Hello, Blazor Full Stack!</h1>
Welcome to your new app.
<Employees Country="USA" />
<Employees />
```

4. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.
5. Откройте браузер Chrome и перейдите по адресу `https://localhost:5151/`. Обратите внимание, что компонент `Employees` отображает сотрудников в зависимости от параметра. Один компонент будет показывать сотрудников из США, а другой — вообще всех (рис. 15.6).
6. Закройте браузер и завершите работу веб-сервера.

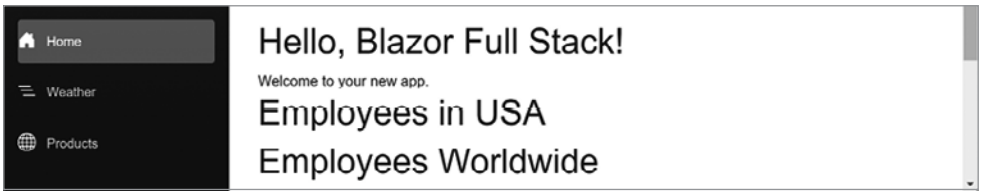


Рис. 15.6. Компоненты Employees с параметром Country="USA" и без заданных параметров

Превращение компонента в маршрутизируемую страницу

Теперь сделаем компонент маршрутизируемым, добавив параметр маршрута для страны.

1. В папке `Components\Pages` откройте файл `Home.razor` и удалите два элемента `Employee`, так как теперь они будут использоваться как страницы.
2. В той же папке добавьте в начало файла `Employees.razor` выражение для регистрации маршрута `/employees` с параметром страны:

```
@rendermode InteractiveServer
@page "/employees/{country?}"
```

3. В папке `Components\Layout` откройте файл `NavMenu.razor` и добавьте в него (после кнопки Products) новые элементы навигации, чтобы можно было переходить к списку сотрудников для различных стран, в том числе из США и Великобритании:

```
<div class="nav-item px-3">
  <NavLink class="nav-link" href="employees" Match="NavLinkMatch.All">
    <span class="bi bi-globe" aria-hidden="true"></span> Worldwide
  </NavLink>
</div>

<div class="nav-item px-3">
  <NavLink class="nav-link" href="employees/USA">
    <span class="bi bi-people" aria-hidden="true"></span> Employees in USA
  </NavLink>
</div>

<div class="nav-item px-3">
  <NavLink class="nav-link" href="employees/UK">
    <span class="bi bi-person" aria-hidden="true"></span> Employees in UK
  </NavLink>
</div>
```

4. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.
5. Откройте браузер Chrome и перейдите по адресу `https://localhost:5151/`.
6. В меню навигации выберите пункт Employees in USA. Убедитесь, что компонент страницы правильно принимает параметр страны и применяет тот же общий макет, что и другие страницы, например `Home.razor`. Также проверьте URL-адрес `https://localhost:5151/employees/USA`.
7. Закройте браузер Chrome и завершите работу веб-сервера.

Получение данных для компонента через веб-сервис

Теперь, когда вы познакомились с базовой реализацией компонента для работы с сущностями, добавим функционал для получения данных с сервера. В примере мы будем использовать контекст базы данных *Northwind* для извлечения сведений о сотрудниках и предоставим их через веб-сервис, построенный на основе минимальных API ASP.NET Core.

1. В редакторе кода создайте новый проект с необходимыми настройками и добавьте его в решение `Chapter15`:
   - выберите шаблон проекта ASP.NET Core Web API (Веб-API ASP.NET Core) / `webapi`;
   - в поле Solution name (Имя решения) укажите значение `Chapter15`;
   - в поле Project name (Имя проекта) укажите значение `Northwind.Minimal-Api.Service`;
   - в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
   - установите флажок Configure for HTTPS (Настроить для HTTPS);
   - снимите флажок Enable Docker (Включить поддержку контейнера);
   - установите флажок Enable OpenAPI support (Включить поддержку OpenAPI);
   - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
   - снимите флажок Use controllers (Использовать контроллеры).
2. Добавьте ссылку на проект контекста БД Northwind для SQL Server, созданный в главе 3:

```
<ItemGroup>
  <ProjectReference Include="..\..\Chapter03\Northwind.Common.DataContext.
  ➥ SqlServer\Northwind.Common.DataContext.SqlServer.csproj" />
</ItemGroup>
```

Путь не должен содержать символы разрыва строки. Если вы не создавали библиотеку классов в главе 3, скачайте проекты из GitHub-репозитория для этой книги.

3. В файле проекта присвойте параметру инвариантной глобализации (`Invariant-Globalization`) значение `false` и включите строгую обработку предупреждений:

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
```

```
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>false</InvariantGlobalization>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>
```

4. В терминале выполните сборку проекта `Northwind.MinimalApi.Service` для проверки, что все проекты библиотеки классов и модели сущностей компилируются корректно:

```
dotnet build
```

5. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5153 для протокола HTTPS и 5154 — для HTTP:

```
"profiles": {
  ...
  "https": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "launchUrl": "swagger",
    "applicationUrl": "https://localhost:5153;http://localhost:5154",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
```

6. В файле `Program.cs` импортируйте нужные пространства имен для работы с атрибутами минимальных API. Также настройте контекст базы данных Northwind и сериализуйте JSON-документы:

```
using Microsoft.AspNetCore.Mvc; // Для атрибута [FromServices]
using Northwind.EntityModels; // Для метода AddNorthwindContext
using System.Text.Json.Serialization; // Для класса ReferenceHandler

// Создание псевдонима для класса JsonOptions
using HttpJsonOptions = Microsoft.AspNetCore.Http.Json.JsonOptions;
```

7. В этом же файле, в конце блока настроек служб, перед вызовом метода `Build`, добавьте код для регистрации контекста базы данных Northwind и настройте механизм внедрения зависимостей для работы с JSON-форматом. Обратите внимание, что для предотвращения исключений из-за циклических ссылок нужно установить соответствующие параметры сериализации:

```
builder.Services.AddNorthwindContext();

builder.Services.Configure<HttpJsonOptions>(options =>
{
```

```
  // Если мы не сохраняем ссылки, то когда JSON-сериализатор
  // обнаружит циклические ссылки, будет выброшено исключение
  Options.SerializerOptions.ReferenceHandler = ReferenceHandler.Preserve;
});
```

Обратите внимание, что используется файл Microsoft.AspNetCore.Http.Json.JsonOptions, а не Microsoft.AspNetCore.Mvc.JsonOptions! Я добавил псевдоним, чтобы явно указать на его назначение, так как для других типов требуется импортировать пространство имен Microsoft.AspNetCore.Mvc.

8. В файле `Program.cs` перед вызовом метода `app.Run()` добавьте операторы для определения конечных точек для операций `GET` и `POST` для работы с данными сотрудников:

```
app.MapGet("api/employees", (
  [FromServices] NorthwindContext db) =>
    Results.Json(db.Employees))
  .WithName("GetEmployees")
  .Produces<Employee[]>(StatusCodes.Status200OK);

app.MapGet("api/employees/{id:int}", (
  [FromServices] NorthwindContext db,
  [FromRoute] int id) =>
  {
    Employee? employee = db.Employees.Find(id);
    if (employee == null)
    {
      return Results.NotFound();
    }
    else
    {
      return Results.Json(employee);
    }
  })
  .WithName("GetEmployeesById")
  .Produces<Employee>(StatusCodes.Status200OK)
  .Produces(StatusCodes.Status404NotFound);

app.MapGet("api/employees/{country}", (
  [FromServices] NorthwindContext db,
  [FromRoute] string country) =>
    Results.Json(db.Employees.Where(employee =>
    employee.Country == country)))
  .WithName("GetEmployeesByCountry")
  .Produces<Employee[]>(StatusCodes.Status200OK);

app.MapPost("api/employees", async ([FromBody] Employee employee,
  [FromServices] NorthwindContext db) =>
  {
```

```
    db.Employees.Add(employee);
    await db.SaveChangesAsync();
    return Results.Created($"api/employees/{employee.EmployeeId}", employee);
  })
  .Produces<Employee>(StatusCodes.Status201Created);
```

9. По желанию можно удалить настройки конечных точек, предназначенных для получения данных о погоде.

Благодаря ограничению {id:int} GET-запрос по пути api/employees/3 будет направлен на конечную точку GetEmployeesById, а GET-запрос по пути api/employees/USA — на точку GetEmployeesByCountry. При POST-запросе к api/employees ответ будет содержать URL-адрес нового сотрудника с идентификатором, присвоенным в БД.

Получение данных в компонент через вызов веб-сервиса

Теперь в компонент сущности добавим функциональность для вызова веб-сервиса.

1. В файле проекта `Northwind.Blazor` подключите пакет `QuickGrid` для отображения данных:

```
<ItemGroup>
  <PackageReference Include= "Microsoft.AspNetCore.Components.QuickGrid"
    Version="8.0.0" />
</ItemGroup>
```

2. Соберите проект, чтобы восстановить все необходимые зависимости.
3. В файле `Program.cs` импортируйте пространство имен для работы с HTTP-заголовками:

```
using System.Net.Http.Headers; // Для класса MediaTypeWithQualityHeaderValue
```

4. В том же файле перед вызовом метода `builder.Build()` добавьте код для настройки фабрики HTTP-клиентов, которая будет использоваться для вызова веб-сервиса:

```
builder.Services.AddHttpClient(name: "Northwind.Blazor.Service",
  configureClient: options =>
  {
    options.BaseAddress = new("https://localhost:5153/");
    options.DefaultRequestHeaders.Accept.Add(
      new MediaTypeWithQualityHeaderValue(
        "application/json", 1.0));
  });
```

5. В папке `Components` откройте файл `_Imports.razor`. Добавьте в него операторы импорта необходимых пространств имен для работы с пакетом `QuickGrid`

и сериализацией JSON. Благодаря этому создаваемым компонентам Blazor не нужно будет импортировать эти пространства имен по отдельности:

```
@using Microsoft.AspNetCore.Components.QuickGrid
@using System.Text.Json @* To use JsonSerializerOptions. *@
@using System.Text.Json.Serialization @* To use ReferenceHandler. *@
```

6. В папке `Components\Pages` откройте файл `Employees.razor` и добавьте в него код для внедрения фабрики HTTP-клиентов. Используйте ее для загрузки и отображения данных либо всех сотрудников, либо сотрудников из конкретной страны:

```
@rendermode InteractiveServer
@page "/employees/{country?}"
@inject IHttpClientFactory httpClientFactory
<h1>
  Employees @(string.IsNullOrWhiteSpace(Country) ? "Worldwide" : "in " +
Country)
</h1>

<QuickGrid Items="@employees" Class="table table-striped table-bordered">
  <PropertyColumn Property="@(emp => emp.EmployeeId)" Title="ID" />
  <PropertyColumn Property="@(emp => emp.FirstName)" />
  <PropertyColumn Property="@(emp => emp.LastName)" />
  <PropertyColumn Property="@(emp => emp.City)" />
  <PropertyColumn Property="@(emp => emp.Country)" />
  <PropertyColumn Property="@(emp => emp.HireDate)" Format="yyyy-MM-dd" />
</QuickGrid>

@code {
  [Parameter]
  public string? Country { get; set; }

  // Пакет QuickGrid работает лучше всего при привязке
  // к последовательности IQueryable<T>
  private IQueryable<Employee>? employees;

  protected override async Task OnParametersSetAsync()
  {
    Employee[]? employeesArray = null;

    // Сущность Employee содержит циклическую ссылку на себя, поэтому
    // мы должны контролировать, как обрабатываются ссылки
    JsonSerializerOptions jsonOptions = new()
      {
        ReferenceHandler = ReferenceHandler.Preserve,
        PropertyNameCaseInsensitive = true
      };

    HttpClient client = httpClientFactory.CreateClient(
      "Northwind.Blazor.Service");
    string path = "api/employees";
```

```
        try
        {
          employeesArray = (await client.GetFromJsonAsync<Employee[]?>(
            path, jsonOptions));
        }
        catch (Exception ex)
        {
          Console.WriteLine($"{ex.GetType()}: {ex.Message}");
        }

        if (employeesArray is not null)
        {
          employees = employeesArray.AsQueryable();

          if (!string.IsNullOrWhiteSpace(Country))
          {
            employees = employees.Where(emp => emp.Country == Country);
          }
        }
      }
    }
```

Хотя веб-сервис предоставляет конечную точку для получения сотрудников из определенной страны, в этой реализации мы сначала получим все данные о сотрудниках, а фильтрацию будем проводить на стороне клиента, используя сетку данных.

7. Если сервер БД не запущен (например, размещен в контейнере, на виртуальной машине или в облаке), обязательно запустите его.
8. Запустите проект `Northwind.MinimalApi.Service` в режиме `https` без отладки.
9. Запустите проект `Northwind.Blazor` в режиме `https` без отладки.
10. Откройте браузер Chrome и перейдите на сайт `https://localhost:5151/`.
11. В левой части страницы выберите пункт Employees in USA. Обратите внимание, что сетка данных о сотрудниках загружается через веб-сервис и отображается на веб-странице (рис. 15.7).

Рис. 15.7. Сетка данных о сотрудниках из США

12. В левой части страницы выберите пункт Employees in UK. Сетка отфильтруется, и будут показаны только те, кто работает в Великобритании.
13. Закройте браузер Chrome и завершите работу веб-сервера.

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 15.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. В чем заключается преимущество новой модели размещения Blazor Full Stack в .NET 8 по сравнению с устаревшими моделями наподобие Blazor Server?
2. Поддерживает ли Blazor WebAssembly все возможности новейших API .NET?
3. Какое расширение у файлов Blazor-компонентов?
4. Как для всех компонентов страницы Blazor применить макет по умолчанию?
5. Как зарегистрировать маршрут для компонента страницы Blazor?
6. Когда свойству `Match` компонента `NavLink` необходимо присваивать значение `NavLinkMatch.All`?
7. В файле `_Imports.razor` вы импортировали пользовательское пространство имен, но, когда пытаетесь использовать класс из него в файле отделенным кодом (code-behind) для компонента Blazor, класс не обнаруживается. Почему? Как решить эту проблему?
8. Что нужно сделать со свойством в классе компонента, чтобы оно автоматически заполнялось из строки запроса?
9. Что такое `QuickGrid`?
10. Как Blazor-компоненту получить доступ к таким функциям браузера, как локальное хранилище?

Упражнение 15.2. Создание компонентов Blazor

Создайте Blazor-компонент `Carousel` с классами библиотеки Bootstrap для работы с каруселями. Этот компонент должен отображать восемь категорий из БД Northwind, включая изображения.

Дополнительную информацию о карусели Bootstrap можно найти на странице https://getbootstrap.com/docs/5.3/components/carousel/.

Упражнение 15.3. Создание сервиса для работы с IndexedDB

Для хранения небольших объемов данных в браузере можно использовать локальное и сессионное хранилище. Однако если нужно более надежное и функциональное решение, стоит использовать API IndexedDB.

Создайте Blazor-сервис `IndexedDbService` и добавьте JavaScript-модуль для взаимодействия с API IndexedDB. Этот сервис будет использоваться для кэширования данных сотрудников.

Более подробную информацию о методах объекта window.indexedDB можно найти по адресу https://developer.mozilla.org/ru/docs/Web/API/IndexedDB_API.

Упражнение 15.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-15---building-web-components-using-blazor.

Резюме

Из этой главы вы узнали:

- о ключевых концепциях Blazor, таких как модели размещения, компоненты, маршрутизация и передача параметров;
- как создавать компоненты Blazor с настраиваемыми параметрами, дочерним содержимым и пользовательскими событиями;
- как создавать компоненты Blazor, которые получают данные через веб-сервис.

В следующей главе мы рассмотрим создание кросс-платформенных приложений для мобильных и классических устройств с использованием .NET MAUI.

16 Создание мобильных и классических приложений с помощью .NET MAUI

В этой главе мы разберемся, как создавать кросс-платформенные мобильные и классические приложения с *графическим интерфейсом пользователя (Graphical User Interface, GUI)* для систем iOS, Android, macOS Catalyst и Windows с помощью *.NET MAUI (Multi-platform App UI)*. По словам команды разработчиков .NET MAUI, между версиями .NET 7 и .NET 8 в API значимой разницы нет — основные улучшения касаются исправления ошибок и повышения производительности.

Вы увидите, как с помощью языка *eXtensible Application Markup Language (XAML)* можно легко определить *пользовательский интерфейс (User Interface, UI)* для GUI-приложений. Название XAML произносится как «замл» или «зэмл».

Разработка кросс-платформенного интерфейса — обширная тема, и ее невозможно охватить за сотню страниц. Тем не менее эта глава познакомит вас с основными возможностями .NET MAUI. Рассматривайте этот материал как введение, которое может вас заинтересовать и вдохновить, а затем переходите к специализированным книгам по разработке мобильных или классических приложений.

Рассматриваемое мобильное приложение позволит управлять списком клиентов, хранящихся в БД Northwind. Оно будет взаимодействовать с веб-сервисом, построенным на минимальных API ASP.NET Core.

Для создания проекта .NET MAUI можно использовать:

- Windows с программой Visual Studio 2022 версии 17.8 или более поздней;
- любую операционную систему с программой Visual Studio Code, `dotnet` CLI или JetBrains Rider

Однако для компиляции приложений WinUI 3 потребуется компьютер под управлением операционной системы Windows, а для сборки проектов под macOS Catalyst и iOS необходим Mac с установленным ПО Xcode.

Обзор XAML

Начнем с изучения XAML — языка разметки, используемого в .NET MAUI.

В 2006 году компания Microsoft представила *Windows Presentation Foundation (WPF)* — первую технологию, в которой использовался XAML. Позже появилась Silverlight — платформа для веб- и мобильных приложений; в настоящее время она больше не поддерживается. Система WPF и сегодня используется при разработке классических приложений Windows; например, программа Visual Studio 2022 частично построена с использованием WPF.

Сегодня XAML применяется в разработке множества типов приложений:

- *.NET MAUI* — кросс-платформенные приложения для Android, iOS, Windows и macOS (развитие технологии `Xamarin.Forms`);
- *WinUI 3* — современные приложения для Windows 10 и 11;
- *Universal Windows Platform (UWP)* — приложения для Windows 10 и 11, Xbox, устройств смешанной реальности, гарнитур виртуальной реальности Meta Quest;
- *WPF* — классические приложения для Windows, включая Windows 7 и новее;
- *Avalonia* и *Uno* — кросс-платформенные решения сторонних разработчиков.

Оптимизация кода с помощью XAML

Язык XAML значительно упрощает код на C#, особенно при создании пользовательского интерфейса.

Допустим, вам нужно создать панель инструментов с несколькими розовыми кнопками, расположенными горизонтально. При нажатии каждая из них должна вызывать определенный метод.

На языке C# задачу можно реализовать следующим образом:

```
HorizontalStackPanel toolbar = new();

Button newButton = new();
newButton.Content = "New";
newButton.Background = new SolidColorBrush(Colors.Pink);
newButton.Clicked += NewButton_Clicked;
toolbar.Children.Add(newButton);

Button openButton = new();
openButton.Content = "Open";
openButton.Background = new SolidColorBrush(Colors.Pink);
openButton.Clicked += OpenButton_Clicked;
toolbar.Children.Add(openButton);
```

А в XAML тот же результат можно получить всего несколькими строками:

```
<HorizontalStackPanel x:Name="toolbar">
  <Button x:Name="newButton" Background="Pink"
          Clicked="NewButton_Clicked">New</Button>
  <Button x:Name="openButton" Background="Pink"
          Clicked="OpenButton_Clicked">Open</Button>
</StackPanel>
```

При обработке XAML устанавливаются эквивалентные свойства и вызываются методы, чтобы достичь того же эффекта, что и в коде C#.

XAML — это удобный альтернативный способ объявления и создания экземпляров объектов .NET, особенно когда речь идет о пользовательском интерфейсе и связанных с ним ресурсах.

С его помощью можно:

- объявлять и многократно использовать ресурсы, такие как кисти, стили и темы. Их можно задать локально (на уровне отдельного элемента пользовательского интерфейса или страницы) или глобально — для всего приложения, чтобы обеспечить совместное использование ресурсов;
- настраивать привязку данных, связывая элементы пользовательского интерфейса между собой или с объектами и коллекциями данных.

Если вы определяете пользовательский интерфейс и связанные с ним компилируемые ресурсы в XAML, то в файле кода нужно вызвать метод `InitializeComponent` в конструкторе страницы:

```
public partial class MainPage : ContentPage
{
  public MainPage()
  {
    InitializeComponent(); // Обработка разметки XAML
  }

  private void NewButton_Clicked(object sender, EventArgs e)
  {
    ...
  }

  private void OpenButton_Clicked(object sender, EventArgs e)
  {
    ...
  }
}
```

Этот вызов загружает XAML-код страницы, создает объявленные в нем элементы управления, применяет их свойства и подключает обработчики событий.

XAML делает код более читаемым, декларативным и гибким, упрощая создание интерфейсов и работу с ресурсами.

Пространства имен .NET MAUI

В .NET MAUI используется несколько ключевых пространств имен, в которых определены основные типы (табл. 16.1).

Таблица 16.1. Ключевые пространства имен .NET MAUI

Пространство имен	Описание
`Microsoft.Maui`	Включает утилиты, такие как `FlowDirection`, `IButton`, `IImage` и `Thickness`
`Microsoft.Maui.Controls`	Содержит элементы управления, страницы и связанные с ними типы, например: `Application`, `Brush`, `Button`, `CheckBox`, `ContentPage`, `Image` и `VerticalStackPanel`
`Microsoft.Maui.Graphics`	Определяет графические объекты, такие как `Color`, `Font`, `ImageFormat`, `PathBuilder`, `Point` и `Size`

Для работы с пространствами имен в *XAML* используются атрибуты `xmlns`, которые добавляются в корневой элемент. Одно пространство имен подключается автоматически, а остальные импортируются с префиксом.

Например:

- *типы .NET MAUI* импортируются по умолчанию, поэтому их можно использовать без префикса;
- *общий синтаксис XAML* подключается с префиксом `x` и используется, например, для задания имен элементов управления или указания имени класса, под которым будет скомпилирован XAML-код;
- *типы, определенные в самом проекте*, обычно импортируются с префиксом `local`:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MyMauiApp.Controls"
             x:Class="MyMauiApp.MainPage"
             ...>

  <Button x:Name="NewFileButton" ...>New File</Button>
  <local:CustomerList x:Name="CustomerList" ... />
  ...
</ContentPage>
```

В этом примере проект `MyMauiApp` содержит элемент управления `CustomerList` в пространстве имен `MyMauiApp.Controls`. Это пространство имен зарегистрировано с префиксом `local`, поэтому его элементы объявляются как `<local:CustomerList>`.

Вы можете импортировать любое количество пространств имен с разными префиксами.

Преобразователи типов

Преобразователи типов используются для приведения строковых значений атрибутов XAML к другим типам. Например, строковое значение `"Pink"` свойства `Background` показанной ниже кнопки задано строкой:

```
<Button x:Name="newButton" Background="Pink" ...
```

При разборе XAML это значение преобразуется в экземпляр класса `SolidColorBrush`:

```
newButton.Background = new SolidColorBrush(Colors.Pink);
```

В .NET MAUI уже есть встроенные преобразователи типов, но можно создавать и регистрировать собственные. Они особенно полезны для визуализации данных.

Выбор элементов управления .NET MAUI

В .NET MAUI есть множество готовых элементов управления, которые можно использовать в различных сценариях в процессе создания пользовательского интерфейса. .NET MAUI (и большинство диалектов XAML) поддерживают элементы управления, перечисленные в табл. 16.2.

Таблица 16.2. Элементы управления пользовательских интерфейсов MAUI

Элемент управления	Описание
`Button`, `ImageButton`, `MenuItem`, `ToolbarItem`	Выполняют действия
`CheckBox`, `RadioButton`, `Switch`	Позволяют выбирать варианты
`DatePicker`, `TimePicker`	Используются для выбора даты и времени
`CollectionView`, `ListView`, `Picker`, `TableView`	Отображают списки и таблицы
`CarouselView`, `IndicatorView`	Реализуют анимированные прокручиваемые представления
`AbsoluteLayout`, `BindableLayout`, `FlexLayout`, `Grid`, `HorizontalStackLayout`, `StackLayout`, `VerticalStackLayout`	Различные контейнеры для организации дочерних элементов
`Border`, `BoxView`, `Frame`, `ScrollView`	Дополнительные визуальные элементы
`Ellipse`, `Line`, `Path`, `Polygon`, `Polyline`, `Rectangle`, `RoundRectangle`	Графические фигуры
`ActivityIndicator`, `Label`, `ProgressBar`, `RefreshView`	Используются для отображения данных, недоступных для редактирования

Продолжение ⇨

Таблица 16.2 *(продолжение)*

Элемент управления	Описание
`Editor`, `Entry`	Предназначены для ввода и редактирования текста
`GraphicsView`, `Image`	Работают с изображениями, видео и аудио
`Slider`, `Stepper`	Позволяют выбирать значения в числовом диапазоне
`SearchBar`	Реализует функцию поиска
`BlazorWebView`, `WebView`	Позволяют встраивать *Blazor* и веб-компоненты
`ContentView`	Используется для создания пользовательских элементов управления

В .NET MAUI элементы управления определяются как свои в пространстве имен `Microsoft.Maui.Controls`. Кроме стандартных элементов управления, в .NET MAUI есть несколько особых.

- `Application` — управляет параметрами кросс-платформенного GUI-приложения: корневой страницей, окнами, темами, ресурсами, а также событиями уровня приложения (`PageAppearing`, `ModalPushing`, `RequestedThemeChanged`). Можно переопределять методы `OnStart`, `OnSleep`, `OnResume` и `CleanUp` для обработки жизненного цикла приложения.
- `Shell` — элемент управления объектом `Page`, который предоставляет удобные механизмы навигации, такие как меню вкладок или боковая панель, а также следит за историей переходов и событиями навигации.

Большинство элементов управления .NET MAUI наследуются от класса View. Это позволяет вкладывать элементы друг в друга и создавать сложные пользовательские интерфейсы.

Расширения разметки в XAML

Расширения разметки добавляют в XAML дополнительные возможности, такие как привязка данных и совместное использование ресурсов.

- `{Binding}` — связывает элемент со значением из другого элемента или источника данных.
- `{OnPlatform}` — настраивает свойства в зависимости от платформы.
- `{StaticResource}` и `{DynamicResource}` — связывают элемент с общим ресурсом.
- `{AppThemeBinding}` — применяется для работы с темами.

В .NET MAUI предусмотрено расширение разметки `OnPlatform`, позволяющее адаптировать разметку к разным платформам. Например, на смартфоне iPhone X и в более поздних моделях есть выемка в верхней части экрана. Вместо добавления дополнительной полоски для всех устройств можно использовать расширение `{OnPlatform}` и настроить этот отступ только для iOS:

```
<VerticalStackLayout>
  <VerticalStackLayout.Padding>
    <OnPlatform x:TypeArguments="Thickness">
      <On Platform="iOS" Value="30,60,30,30" />
      <On Platform="Android" Value="30" />
      <On Platform="WinUI" Value="30" />
    </OnPlatform>
  </VerticalStackLayout.Padding>
```

А вот так выглядит упрощенный синтаксис:

```
<VerticalStackLayout Padding"{OnPlatform iOS='30,60,30,30', Default='30'}">
```

Обзор .NET MAUI

Для создания мобильного приложения, которое должно работать только на iPhone, можно воспользоваться языками Objective-C или Swift и библиотеками UIKit в Xcode.

Если приложение предназначено исключительно для устройств под управлением Android, подойдут Java или Kotlin с библиотеками Android SDK в Android Studio.

Но что делать, если требуется создать мобильное приложение сразу и для iPhone, и для Android? Или если хочется разработать один проект под два типа устройств, используя уже знакомый язык программирования и платформу? А что, если, адаптировав интерфейс под устройства с большим экраном, можно будет запустить приложение еще и на компьютерах с macOS и Windows?

.NET MAUI позволяет создавать кросс-платформенные мобильные приложения для iOS (iPhone), iPadOS, macOS (с помощью Catalyst), Windows (WinUI 3) и Google Android. Код пишется на C# и .NET, затем компилируется в нативные API и выполняется на целевых платформах мобильных устройств и классических компьютеров.

Бизнес-логика разрабатывается один раз и используется на всех поддерживаемых устройствах. Однако интерфейс и взаимодействие с API различаются на мобильных и классических платформах, поэтому иногда слой UI адаптируется отдельно для каждой из них.

Как и в приложениях WPF и UWP, в .NET MAUI можно описывать интерфейс с помощью XAML, используя абстракции UI-компонентов, соответствующих

нативным элементам конкретной платформы. При этом .NET MAUI не рисует интерфейс самостоятельно — он использует системные виджеты, обеспечивая естественный внешний вид и привычное поведение приложений на каждой платформе.

Хотя нативные инструменты позволяют добиться идеальной адаптации под конкретную ОС, для большинства мобильных и настольных приложений это некритично. Приложив немного усилий, можно создать красивый и удобный интерфейс. Microsoft продемонстрировала возможности .NET MAUI в своем UI-челлендже — ознакомиться с результатами можно по адресу https://devblogs.microsoft.com/dotnet/announcing-dotnet-maui-beautiful-ui-challenge/.

Поддержка .NET MAUI и Xamarin

Основные версии платформы .NET MAUI поставляются вместе с .NET, начиная с версии .NET 7, но как *дополнительная рабочая нагрузка*. Это означает, что MAUI не следует *стандартной схеме краткосрочной (Short Term Support, STS)* и *долгосрочной (Long Term Support, LTS) поддержки*, принятой в основной платформе .NET.

Каждая версия .NET MAUI поддерживается лишь 18 месяцев, что фактически превращает их в STS-релизы. Это касается и версии .NET MAUI, поставляемой в качестве рабочей нагрузки в составе .NET 8.

Кроме того, .NET MAUI зависит от других операционных систем, таких как iOS и macOS, что добавляет сложности. Основные обновления iOS традиционно выходят в сентябре, а новые версии iPadOS и macOS — в октябре или ноябре. Так что у разработчиков .NET MAUI остается совсем немного времени на адаптацию платформы перед релизом основной версии .NET, который обычно приходится на ноябрь.

Внимание! Поддержка платформы Xamarin завершилась 1 мая 2024 года.

Все проекты, созданные на Xamarin и Xamarin.Forms, необходимо перенести на .NET MAUI или альтернативные платформы (Avalonia, Uno и др.).

Средства разработки для мобильных и облачных проектов

Мобильные приложения часто работают в связке с облачными сервисами. Сатья Наделла, генеральный директор Microsoft, выразил это так: *«Когда мы говорим о подходе mobile first, речь идет не о мобильности устройства, а о мобильности индивидуального опыта. [...] Единственный способ организовать мобильность этих приложений и данных — облако»*.

При установке Visual Studio 2022 необходимо выбрать рабочую нагрузку .NET Multi-platform App UI development (Разработка с помощью .NET Multi-platform App UI), которая находится в разделе Desktop & Mobile (Классические и мобильные приложения) (рис. 16.1).

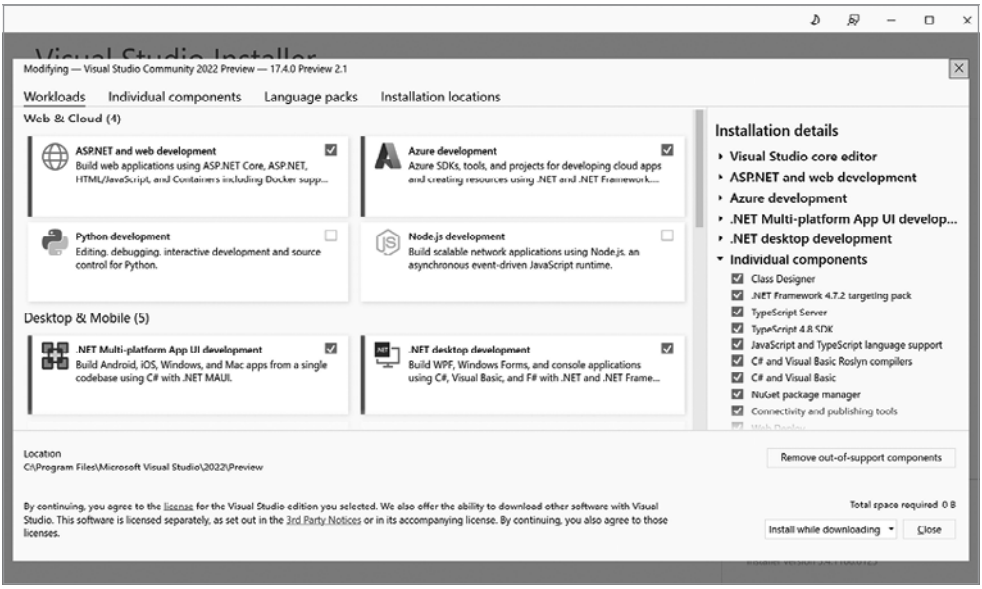


Рис. 16.1. Выбор рабочей нагрузки .NET Multi-platform App UI development в Visual Studio 2022

Установка рабочих нагрузок .NET MAUI вручную

В процессе установки Visual Studio 2022 выбранные рабочие нагрузки устанавливаются автоматически. Если рабочие нагрузки не были выбраны при инсталляции, их можно добавить вручную.

Проверить, какие рабочие нагрузки установлены, поможет следующая команда:

```
dotnet workload list
```

Вывод будет содержать список установленных компонентов, например:

```
Installed Workload Ids Manifest Version Installation Source
----------------------------------------------------------------------
maui-maccatalyst 6.0.486/6.0.400 SDK 7.0.100
```

Чтобы вывести доступные для установки рабочие нагрузки, выполните следующую команду:

```
dotnet workload search
```

Доступные к установке рабочие нагрузки будут отображены в таблице:

```
Workload ID                Description
--------------------------------------------------------------------------------
---
android                    .NET SDK Workload for building Android applications.
ios                        .NET SDK Workload for building iOS applications.
maccatalyst                .NET SDK Workload for building MacCatalyst
applications.
macos                      .NET SDK Workload for building macOS applications.
maui                       .NET MAUI SDK for all platforms
maui-android               .NET MAUI SDK for Android
maui-desktop               .NET MAUI SDK for Desktop
maui-ios                   .NET MAUI SDK for iOS
maui-maccatalyst           .NET MAUI SDK for Mac Catalyst
maui-mobile                .NET MAUI SDK for Mobile
maui-tizen                 .NET MAUI SDK for Tizen
maui-windows               .NET MAUI SDK for Windows
tvos                       .NET SDK Workload for building tvOS applications.
wasi-experimental          workloads/wasi-experimental/description
wasm-experimental          workloads/wasm-experimental/description
wasm-experimental-net7     .NET WebAssembly experimental tooling for net7.0
wasm-tools                 .NET WebAssembly build tools
wasm-tools-net6            .NET WebAssembly build tools for net6.0
wasm-tools-net7            .NET WebAssembly build tools for net7.0
```

Чтобы установить рабочую нагрузку .NET MAUI для всех платформ, выполните:

```
dotnet workload install maui
```

Обновить существующие рабочие нагрузки можно следующим образом:

```
dotnet workload update
```

Добавить недостающие рабочие нагрузки в проект поможет следующая команда:

```
dotnet workload restore <projectname>
```

Со временем может накопиться несколько версий рабочих нагрузок для разных версий .NET SDK. До .NET 8 их приходилось удалять вручную, что иногда вызывало проблемы. В .NET 8 появилась новая команда для безопасного удаления неиспользуемых рабочих нагрузок:

```
dotnet workload clean
```

В копилку знаний: если вы планируете разрабатывать в программе Visual Studio 2022 мобильные приложения под iOS или классические приложения под macOS Catalyst, можно подключиться к сетевому компьютеру Mac для их сборки. Инструкции вы найдете по адресу https://learn.microsoft.com/ru-ru/dotnet/maui/ios/pair-to-mac.

Категории компонентов пользовательского интерфейса .NET MAUI

В состав .NET MAUI входят элементы управления общего назначения для создания пользовательских интерфейсов. Они делятся на четыре категории.

- *Страницы* — представляют собой экраны кросс-платформенных приложений. К ним относятся `Shell`, `ContentPage`, `NavigationPage`, `FlyoutPage` и `TabbedPage`.
- *Макеты* — обеспечивают структуру и компоновку других элементов интерфейса. Примеры: `Grid`, `StackLayout`, `FlexLayout`.
- *Представления* — отдельные компоненты пользовательского интерфейса, такие как `CarouselView`, `CollectionView`, `Label`, `Entry`, `Editor` и `Button`.
- *Ячейки* — элементы списка или таблицы, например `TextCell`, `ImageCell`, `SwitchCell` и `EntryCell`.

Элемент управления Shell

Элемент `Shell` упрощает создание приложений, предоставляя стандартные механизмы навигации и поиска. В проекте создается класс, наследуемый от `Shell` и включающий следующие компоненты навигации.

- `TabBar` — содержит вкладки (`Tab`). Подходит для небольшого числа страниц (до 4–5).
- `FlyoutItem` — используется для списка навигации, если страниц больше. Отображается в виде вертикального прокручиваемого списка.
- `ShellContent` — содержит страницы (`ContentPage`), используемые внутри вкладок или списка.

Можно комбинировать `TabBar` и `FlyoutItem`, обеспечивая единый механизм навигации. Элемент `Shell` будет поддерживать их синхронизацию.

Всплывающее меню навигации (flyout menu) представляет собой выезжающий из левой части экрана мобильного устройства или главного окна классического приложения список элементов. Оно вызывается нажатием на значок «гамбургер» с тремя горизонтальными линиями. При выборе элемента создается новая страница, если она еще не загружена.

На верхней панели интерфейса автоматически отображается кнопка «Назад» для возврата на предыдущую страницу.

Элемент управления ListView

Элемент управления `ListView` используется для отображения длинных списков однотипных данных. Он поддерживает:

- заголовки и нижние колонтитулы;
- группировку элементов;
- встроенные типы ячеек (текстовые и с изображениями);
- настраиваемые типы ячеек.

Для ячеек `ListView` доступны контекстные действия — они появляются:

- при смахивании на iPhone;
- при долгом нажатии на Android;
- при щелчке правой кнопкой мыши на ПК.

Деструктивные действия, например удаление, могут выделяться красным цветом:

```
<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>
```

Элементы управления Entry и Editor

Оба элемента используются для ввода/редактирования текста и часто привязываются к свойству модели сущности:

- `Entry` — ввод одной строки текста;
- `Editor` — ввод многострочного текста:

```
<Editor Text="{Binding CompanyName, Mode=TwoWay}" />
```

Дельный совет: используйте элемент управления Entry для одной строки текста, а Editor — для многострочного текста.

Обработчики .NET MAUI

Элементы управления XAML в .NET MAUI определены в пространстве имен `Microsoft.Maui.Controls`. Они сопоставляются с нативными элементами управления через *обработчики (handlers)*:

- iOS: `Button` → `UIButton (UIKit)`;
- macOS: `Button` → `NSButton (AppKit)`;
- Android: `Button` → `AppCompatButton`.

Обработчики имеют свойство `NativeView`, через которое можно работать с платформенно-специфичными свойствами, методами и событиями и настраивать все экземпляры нативного элемента управления.

Разработка платформенно-специфичного кода

Для выполнения кода на конкретной платформе используются предопределенные *константы компилятора*.

Например, в Android поле `Entry` по умолчанию отображает нижнее подчеркивание. Чтобы его скрыть, можно использовать платформенно-специфичный код для Android, чтобы получить обработчик элемента управления `Entry`, использовать его свойство `NativeView` для доступа к базовому нативному элементу управления и затем присвоить соответствующему свойству значение `false`:

```
#if __ANDROID__
  Handlers.EntryHandler.EntryMapper[nameof(IEntry.BackgroundColor)] = (h, v) =>
  {
    (h.NativeView as global::Android.Views.Entry).UnderlineVisible = false;
  };
#endif
```

Используются следующие предопределенные константы компилятора:

- `__ANDROID__`
- `__IOS__`
- `WINDOWS`

Важно! Синтаксис `#if` в C# отличается от стандартного оператора `if`:

```
#if __IOS__
  // платформенно-специфичный код для iOS
#elif __ANDROID__
  // платформенно-специфичный код для Android
#elif WINDOWS
  // платформенно-специфичный код для Windows
#endif
```

Теперь, когда мы рассмотрели основные концепции .NET MAUI и установили компоненты, необходимые для работы платформы MAUI, давайте создадим первый проект.

Разработка мобильных и классических приложений с помощью .NET MAUI

Создадим мобильное и классическое приложение для управления клиентами в Northwind.

Дельный совет: если вы работаете на компьютере Mac и никогда не запускали Xcode, сделайте это прямо сейчас. Просто откройте инструмент и дождитесь появления окна запуска. Так вы убедитесь, что все необходимые компоненты установлены и зарегистрированы. Если вы этого не сделаете, в будущем при работе с проектами могут возникнуть ошибки.

Виртуализация Android-устройства для локального тестирования приложений

Для работы с ОС Android нужно установить хотя бы один инструмент Android SDK. Стандартная установка программы Visual Studio 2022 с рабочей нагрузкой для мобильной разработки включает Android SDK, но часто это устаревшая версия, предназначенная для поддержки большинства Android-устройств. Чтобы воспользоваться последними возможностями платформы .NET MAUI, необходимо настроить виртуальное устройство с актуальной версией Android.

1. Запустите программу Visual Studio 2022 в ОС Windows. Если появилось диалоговое окно со списком последних проектов, щелкните кнопкой мыши на ссылке Continue without code (Продолжить без кода).
2. Выберите команду меню Tools ▸ Android ▸ Android Device Manager (Средства ▸ Android ▸ Диспетчер устройств Android). Если появится запрос системы контроля учетных записей пользователей на разрешение этому приложению вносить изменения в устройство, нажмите кнопку Yes (Да).
3. В диалоговом окне Android Device Manager (Диспетчер устройств Android) нажмите кнопку + New (+ Новинка), чтобы создать новое устройство.
4. Далее выберите следующие параметры (рис. 16.2):
   - в раскрывающемся списке Base Device (Базовое устройство) выберите пункт Pixel 5;
   - в раскрывающемся списке Processor (Процессор) выберите пункт x86_64;
   - в раскрывающемся списке OS (ОС) выберите пункт Android 13.0 — API 33;
   - установите флажок Google APIs;
   - снимите флажок Google Play Store.
5. Нажмите кнопку Create (Создать).
6. Примите все лицензионные соглашения.
7. Дождитесь загрузки всех необходимых файлов.

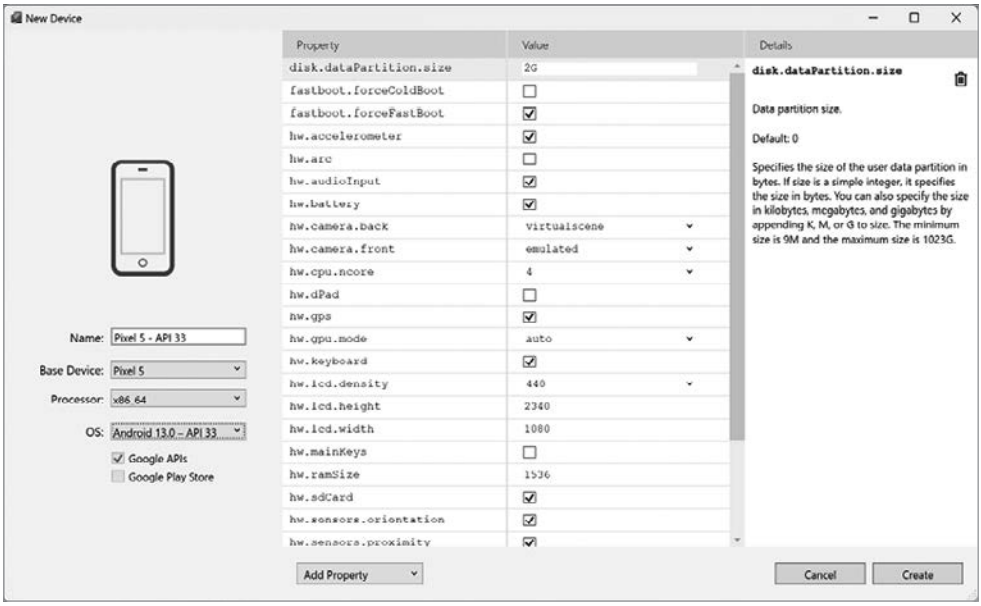


Рис. 16.2. Выбор оборудования и ОС для виртуального устройства Android

8. В диалоговом окне Android Device Manager (Диспетчер устройств Android) найдите созданное устройство и нажмите кнопку Start (Запустить).
9. Будьте терпеливы! Запуск эмулятора может занять несколько минут.
10. После загрузки устройства откройте браузер Chrome и проверьте доступ к сети, зайдя на сайт https://russian.rt.com/.
11. Закройте эмулятор.
12. Закройте диалоговое окно Android Device Manager (Диспетчер устройств Android).
13. Перезапустите среду разработки Visual Studio 2022, чтобы оповестить ее о новом эмуляторе.

Включение режима разработчика Windows

Чтобы создавать приложения для Windows, необходимо включить режим разработчика.

1. В меню Start (Пуск) выберите пункт Settings (Настройки). В открывшемся диалоговом окне выберите пункт Update & security (Обновление и безопасность) и перейдите в раздел For developers (Для разработчиков). Установите переключатель Developer Mode (Режим разработчика) в активное положение.
2. Примите предупреждение о том, что это «может ослабить безопасность вашего устройства и личных данных или нанести ему вред», нажав кнопку Yes (Да).
3. Закройте приложение Settings (Настройки).

Создание проекта .NET MAUI

Теперь создадим проект кросс-платформенного приложения для мобильных и классических компьютеров.

1. В программе Visual Studio 2022 создайте проект со следующими настройками:
   - выберите шаблон проекта .NET MAUI App (Приложение .NET MAUI) / `maui`;

Вы можете отфильтровать шаблоны, выбрав язык C# и тип проекта MAUI.

   - в поле Project name (Имя проекта) укажите значение `Northwind.Maui.Client`;
   - в поле Solution name (Имя решения) укажите значение `Chapter16`.
2. В операционной системе Windows, если появится предупреждение о блокировке брандмауэром Windows Defender некоторых функций Broker во всех публичных и частных сетях, установите флажок Private networks (Частные сети) и сбросьте флажок Public networks (Публичные сети), а затем нажмите кнопку Allow access (Разрешить доступ).
3. В файле проекта обратите внимание на элементы, которые определяют целевой фреймворк для iOS, Android и Mac Catalyst, а также на элемент для добавления целевого фреймворка для Windows. Убедитесь, что проект настроен как единый проект MAUI. Фрагмент разметки показан ниже:

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

  <PropertyGroup>
    <TargetFrameworks>net8.0-ios;net8.0-android;net8.0-maccatalyst
    ➥ </TargetFrameworks>
    <TargetFrameworks Condition="$([MSBuild]::IsOSPlatform('windows'))">
      $(TargetFrameworks);net8.0-windows10.0.19041.0</TargetFrameworks>
    ...
    <OutputType>Exe</OutputType>
    <RootNamespace>Northwind.Maui.Client</RootNamespace>
    <UseMaui>true</UseMaui>
    <SingleProject>true</SingleProject>
```

Если возникает ошибка Error NU1012 Platform version is not present для одного или нескольких целевых фреймворков, хотя в них указана платформа net8.0-ios, net8.0-maccatalyst, выполните восстановление рабочих нагрузок для проекта в терминале, находясь в папке проекта:

```
dotnet workload restore
```

4. Справа от кнопки Run (Запуск) на панели инструментов присвойте параметру Framework (Фреймворк) значение net8.0-android и выберите образ эмулятора Pixel 5 — API 33 (Android 13.0 — API 33), созданный ранее (рис. 16.3).

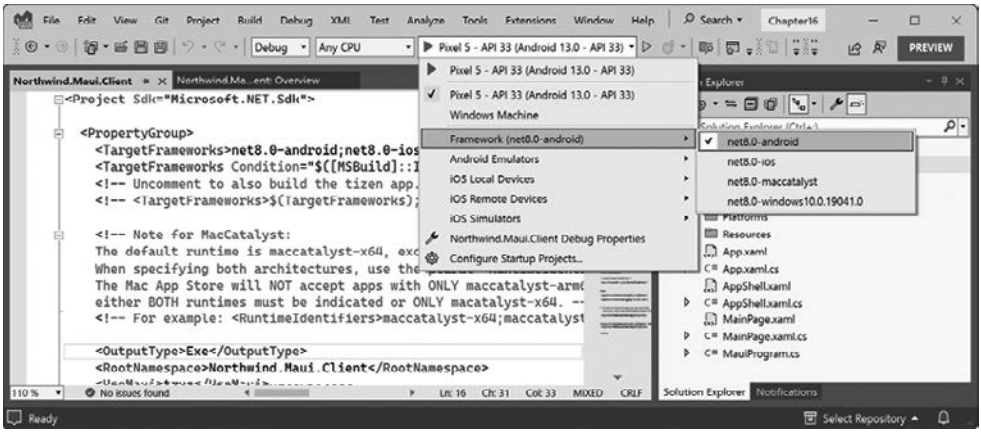


Рис. 16.3. Выбор виртуального устройства Android для запуска

5. Нажмите кнопку Run (Запуск) на панели инструментов и дождитесь, пока эмулятор загрузит Android и развернет мобильное приложение. Это может занять больше 5 минут, особенно если вы запускаете проект MAUI впервые. Следите за строкой состояния в Visual Studio 2022 (рис. 16.4).

Рис. 16.4. Строка состояния отражает ход развертывания приложения .NET MAUI

При первом запуске, возможно, потребуется подтвердить лицензионное соглашение Google.

6. В приложении .NET MAUI нажмите кнопку Click me, чтобы счетчик увеличился трижды (рис. 16.5).
7. Закройте эмулятор устройства Android. Можно оставить его включенным.
8. Справа от кнопки Run (Запуск) на панели инструментов присвойте параметру Framework (Фреймворк) значение net8.0-windows10.0.19041.0.
9. Выберите конфигурацию Debug (Отладка), а затем нажмите кнопку Start (Старт) с зеленым треугольником и надписью Windows Machine. Если появляется предупреждение об отсутствии пакетов, установите их, нажав Start (Старт) второй раз.

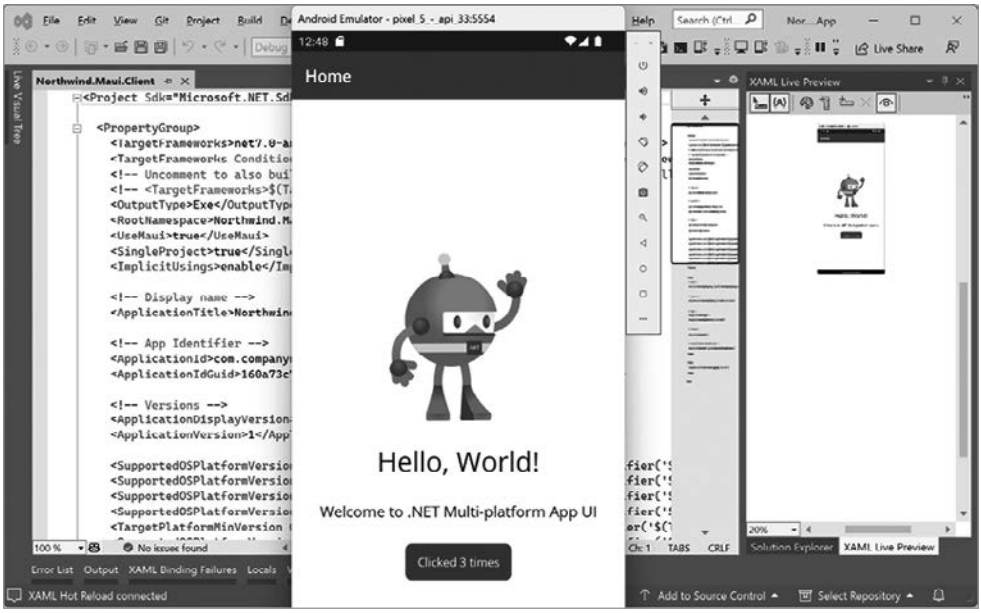


Рис. 16.5. Кнопка счетчика в приложении .NET MAUI на Android

10. Через несколько секунд приложение откроется с теми же функциями и кнопкой Click me, как в мобильной версии (рис. 16.6).
11. Закройте приложение.

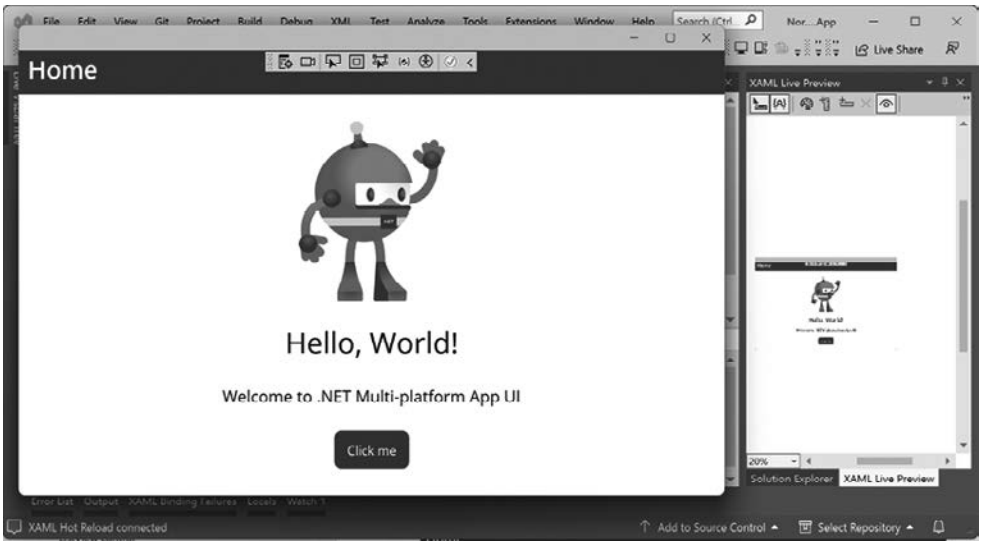


Рис. 16.6. То же приложение .NET MAUI в Windows

Дельный совет: вы должны тестировать приложение .NET MAUI на всех устройствах, на которых оно будет работать. В этой главе, даже если я не указываю об этом прямо, рекомендую проверять приложение на эмулированных устройствах Android и в Windows после каждой задачи. Так вы его увидите на мобильном устройстве с узким портретным форматом и на настольном устройстве с крупным ландшафтным форматом. Приложения для Mac тестируйте в эмуляторах iOS, Android и как классическое приложение Mac Catalyst.

Добавление навигационной оболочки и страниц с дополнительным содержимым

Теперь рассмотрим структуру приложения .NET MAUI, а затем добавим в проект новые страницы и навигацию.

1. В папке проекта `Northwind.Maui.Client` откройте файл `MauiProgram.cs`. Обратите внимание, что объект `builder` вызывает метод `UseMauiApp`, указывая `App` в качестве общего типа:

```
using Microsoft.Extensions.Logging;

namespace Northwind.Maui.Client;

public static class MauiProgram
{
  public static MauiApp CreateMauiApp()
  {
    var builder = MauiApp.CreateBuilder();
    builder
      .UseMauiApp<App>()
      .ConfigureFonts(fonts =>
      {
        fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
        fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
      });

#if DEBUG
    builder.Logging.AddDebug();
#endif

    return builder.Build();
  }
}
```

2. На панели Solution Explorer (Обозреватель решений) разверните пункт `App.xaml`, откройте файл `App.xaml.cs` и проверьте, что свойству `MainPage` назначен экземпляр `AppShell` (выделено ниже):

```
namespace Northwind.Maui.Client;

public partial class App : Application
{
  public App()
```

```
    {
      InitializeComponent();

      MainPage = new AppShell();
    }
}
```

3. В файле `AppShell.xaml` у оболочки должен быть отключен режим всплывающих подсказок и она должна содержать только одну страницу — `MainPage`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  x:Class="Northwind.Maui.Client.AppShell"
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:Northwind.Maui.Client"
  Shell.FlyoutBehavior="Disabled"
  Title="Northwind.Maui.Client">

  <ShellContent
    Title="Home"
    ContentTemplate="{DataTemplate local:MainPage}"
    Route="MainPage" />

</Shell>
```

Оболочка с одной страницей содержимого не отображает навигацию. Чтобы навигация работала, необходимо добавить минимум две страницы.

4. Добавьте в папку `Resources/Images` изображения значков для всплывающих элементов навигации, которую мы создадим чуть позже.

Готовые изображения можно скачать из репозитория GitHub по адресу https://github.com/markjprice/apps-services-net8/tree/main/code/Chapter16/Northwind.Maui.Client/Resources/Images.

5. В файле `AppShell.xaml` включите режим всплывающих подсказок, настройте бледно-голубой (`AliceBlue`) цвет фона, добавьте значок и всплывающий заголовок для `MainPage`, а также несколько всплывающих элементов с содержимым оболочки:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
  x:Class="Northwind.Maui.Client.AppShell"
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:Northwind.Maui.Client"
  Shell.FlyoutBehavior="Flyout"
  Title="Northwind.Maui.Client"
  FlyoutBackgroundColor="AliceBlue">
```

```
  <Shell.FlyoutHeader>
    <HorizontalStackLayout Spacing="10" HorizontalOptions="Start">
      <Image Source="wind_face_3d.png" WidthRequest="80"
        HeightRequest="80" />
      <Label Text="Northwind" FontFamily="OpenSansSemibold" FontSize="32"
        VerticalOptions="Center" />
    </HorizontalStackLayout>
  </Shell.FlyoutHeader>

  <ShellContent Title="Home"
    Icon="file_cabinet_3d.png"
    ContentTemplate="{DataTemplate local:MainPage}"
    Route="MainPage" />

  <ShellContent Title="Categories"
    Icon="delivery_truck_3d.png"
    ContentTemplate="{DataTemplate local:CategoriesPage}"
    Route="Categories" />

  <ShellContent Title="Products"
    Icon="cityscape_3d.png"
    ContentTemplate="{DataTemplate local:ProductsPage}"
    Route="Products" />

  <ShellContent Title="Customers"
    Icon="card_index_3d.png"
    ContentTemplate="{DataTemplate local:CustomersPage}"
    Route="Customers" />

  <ShellContent Title="Employees"
    Icon="identification_card_3d.png"
    ContentTemplate="{DataTemplate local:EmployeesPage}"
    Route="Employees" />

  <ShellContent Title="Settings"
    Icon="gear_3d.png"
    ContentTemplate="{DataTemplate local:SettingsPage}"
    Route="Settings" />

</Shell>
```

В некоторых строках свойства ContentTemplate могут появляться предупреждения об отсутствии страниц — их мы создадим позже. Выбранный цвет хорошо смотрится в светлом режиме. Если ваш интерфейс в темном режиме, цвет #75858a может подойти лучше, чем AliceBlue.

6. В Visual Studio 2022 щелкните правой кнопкой мыши на папке `Northwind.Maui.Client` и в контекстном меню выберите пункт Add ▸ New Item (Добавить ▸ Новый элемент) или нажмите сочетание клавиш Ctrl+Shift+A. Выберите вариант .NET MAUI ContentPage (XAML), введите имя `SettingsPage` и нажмите кнопку Add (Добавить).

В Visual Studio Code и JetBrains Rider нет шаблонов элементов проекта для MAUI. Страницы можно создать в терминале с помощью следующей команды:

```
dotnet new maui-page-xaml --name SettingsPage.xaml
```

7. Повторите предыдущий шаг, добавив страницы:
   - `CategoriesPage`;
   - `CustomersPage`;
   - `CustomerDetailPage`;
   - `EmployeesPage`;
   - `ProductsPage`.
8. На панели Solution Explorer (Обозреватель решений) дважды щелкните мышью по файлу `CategoriesPage.xaml`, чтобы открыть его для редактирования. В программе Visual Studio 2022 пока нет графического дизайнера XAML.
9. В элементе `ContentPage` присвойте параметру `Title` значение `Categories`. Такое же значение присвойте параметру `Text` в элементе `Label`:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Northwind.Maui.Client.CategoriesPage"
             Title="Categories">
  <VerticalStackLayout>
    <Label
      Text="Categories"
      VerticalOptions="Center"
      HorizontalOptions="Center" />
  </VerticalStackLayout>
</ContentPage>
```

10. Выберите команду меню View ▸ Toolbox (Вид ▸ Панеаль инструментов) или нажмите Ctrl+W, X. На панели инструментов доступны разделы Controls (Элементы управления), Layouts (Макеты), Cells (Ячейки) и General (Общие). Если панель не используется, разметку можно ввести вручную.
11. В строке поиска в верхней части панели инструментов введите запрос `b`, отфильтруйте список и найдите элементы `Button`, `ProgressBar` и `AbsoluteLayout`.
12. Перетащите элемент управления `Button` (Кнопка) с панели инструментов в XAML-разметку и разместите после элемента `Label` перед закрывающим тегом элемента `VerticalStackLayout`. Присвойте свойству `Text` значение `Hello!`:

```
<Button Text="Hello!" />
```

13. Выберите режим запуска Windows Machine, а затем запустите проект `Northwind.Maui.Client` с отладкой. В строке состояния Visual Studio появится сообщение о подключении XAML Hot Reload.

14. В левом верхнем углу приложения щелкните на кнопке меню «гамбургер» и убедитесь, что заголовки и значки всплывающих элементов отображаются правильно (рис. 16.7).

Рис. 16.7. Всплывающее окно со значками

15. Во всплывающем меню выберите пункт Categories и убедитесь, что кнопка с текстом Hello! растягивается на всю ширину окна приложения.
16. Не закрывая приложение, в Visual Studio 2022 присвойте свойству `Text` значение `Click Me`, добавьте атрибут `WidthRequest = "100"` и убедитесь, что функция XAML Hot Reload автоматически применяет изменения в приложении (рис. 16.8).
17. Закройте приложение.

Рис. 16.8. Функция XAML Hot Reload автоматически обновляет изменения в XAML в запущенном приложении

18. Измените элемент `Button`, присвоив ему имя `ClickMeButton`, и создайте обработчик события `Clicked` (рис. 16.9).

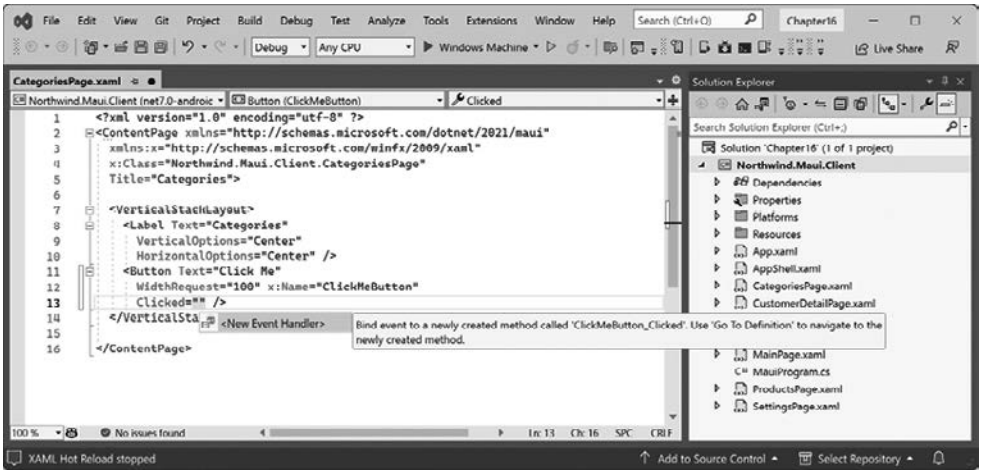


Рис. 16.9. Добавление обработчика событий к элементу управления

19. Щелкните правой кнопкой мыши на имени обработчика события и в контекстном меню выберите пункт Go To Definition (Перейти к определению) или нажмите клавишу F12.
20. Добавьте в метод обработчика события оператор, настраивающий на кнопке текущее время:

```
private void ClickMeButton_Clicked(object sender, EventArgs e)
{
  ClickMeButton.Text = DateTime.Now.ToString("hh:mm:ss");
}
```

21. Запустите проект `Northwind.Maui.Client` на мобильном устройстве и компьютере.

Дельный совет: при развертывании в эмуляторе Android/iOS старая версия приложения может быть все еще запущена. Дождитесь развертывания новой версии приложения перед тестированием. Ход развертывания можно отслеживать в строке состояния программы Visual Studio с сообщением Hot Reload connected.

22. Перейдите в раздел Categories, нажмите кнопку и убедитесь, что текст на ней изменился на текущее время.
23. Закройте приложение.

Внедрение дополнительных страниц с содержимым

Добавим несколько новых страниц и реализуем на одной из них простой калькулятор.

1. В файле `EmployeesPage.xaml` присвойте параметру `Title` значение `Employees` и добавьте разметку пользовательского интерфейса для калькулятора:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Northwind.Maui.Client.EmployeesPage"
             Title="Employees">

  <VerticalStackLayout>

    <Grid Background="DarkGray" Margin="10"
        Padding="5" x:Name="GridCalculator"
        ColumnDefinitions="Auto,Auto,Auto,Auto"
        RowDefinitions="Auto,Auto,Auto,Auto">
      <Button Grid.Row="0" Grid.Column="0" Text="X" />
      <Button Grid.Row="0" Grid.Column="1" Text="/" />
      <Button Grid.Row="0" Grid.Column="2" Text="+" />
      <Button Grid.Row="0" Grid.Column="3" Text="-" />
      <Button Grid.Row="1" Grid.Column="0" Text="7" />
      <Button Grid.Row="1" Grid.Column="1" Text="8" />
      <Button Grid.Row="1" Grid.Column="2" Text="9" />
      <Button Grid.Row="1" Grid.Column="3" Text="0" />
      <Button Grid.Row="2" Grid.Column="0" Text="4" />
      <Button Grid.Row="2" Grid.Column="1" Text="5" />
      <Button Grid.Row="2" Grid.Column="2" Text="6" />
      <Button Grid.Row="2" Grid.Column="3" Text="." />
      <Button Grid.Row="3" Grid.Column="0" Text="1" />
      <Button Grid.Row="3" Grid.Column="1" Text="2" />
      <Button Grid.Row="3" Grid.Column="2" Text="3" />
      <Button Grid.Row="3" Grid.Column="3" Text="=" />
    </Grid>
    <Label x:Name="Output" FontSize="24" VerticalOptions="Center"
      HorizontalOptions="Start" />

  </VerticalStackLayout>
</ContentPage>
```

2. Добавьте обработчик события `Loaded` для страницы:

```
Title="Employees"
Loaded="ContentPage_Loaded">
```

3. В файле `EmployeesPage.xaml.cs` напишите код, который изменяет размер каждой кнопки в сетке и подключает обработчик события `Clicked`:

```
private void ContentPage_Loaded(object sender, EventArgs e)
{
  foreach (Button button in GridCalculator.Children.OfType<Button>())
  {
    button.FontSize = 24;
    button.WidthRequest = 54;
    button.HeightRequest = 54;
    button.Clicked += Button_Clicked;
  }
}
```

4. Создайте метод `Button_Clicked`, который будет добавлять текст нажатой кнопки к выходной метке:

```
private void Button_Clicked(object sender, EventArgs e)
{
  string operationChars = "+-/X=";

  Button button = (Button)sender;

  if (operationChars.Contains(button.Text))
  {
    Output.Text = string.Empty;
  }
  else
  {
    Output.Text += button.Text;
  }
}
```

Это пока не полноценный калькулятор — математические операции не обрабатываются. Основная цель — разобраться с созданием интерфейсов в .NET MAUI. В качестве дополнительного упражнения можно изучить, как реализовать логику вычислений.

5. Запустите проект `Northwind.Maui.Client` на компьютере и мобильном устройстве.
6. Перейдите в раздел Employees, нажимайте кнопки калькулятора и следите за обновлением выходной метки. Процесс на эмулированном устройстве Android показан на рис. 16.10.
7. Закройте приложение.

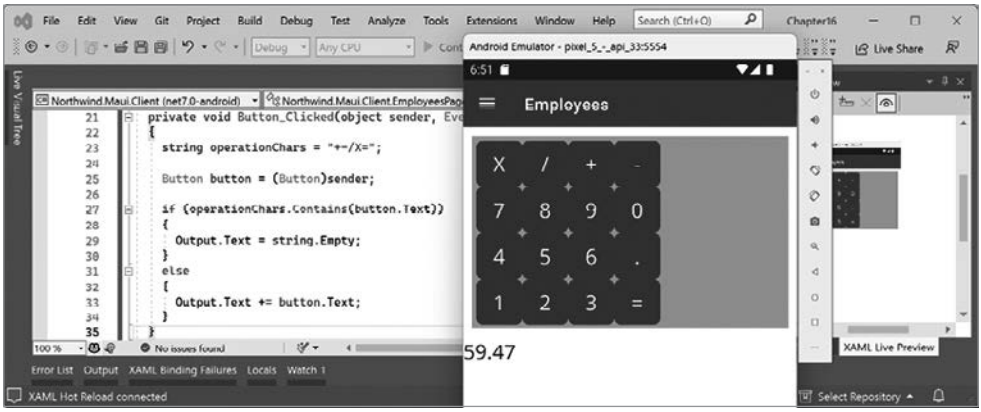


Рис. 16.10. Интерфейс калькулятора на эмулированном устройстве Android

На этом этапе создано несколько простых пользовательских интерфейсов с применением XAML и платформы MAUI. Далее рассмотрим, как управлять ресурсами, чтобы улучшить приложение.

Управление общими ресурсами

При создании графических пользовательских интерфейсов часто требуется задействовать общие ресурсы, например кисти для фона или объекты с пользовательскими параметрами. Ресурсы можно определять на разных уровнях. Они могут использоваться совместно всеми элементами на текущем уровне или ниже.

- *Приложение* — ресурсы доступны во всех страницах приложения.
- *Страница* — ресурсы применяются только к конкретной странице.
- *Элемент управления* — ресурсы доступны только внутри элемента.

Определение общих ресурсов в приложении

Удобнее всего задавать общие ресурсы на уровне приложения. Давайте сделаем это.

1. В папке `Resources/Styles` создайте файл `Northwind.xaml`, выбрав шаблон проекта .NET MAUI Resource Dictionary (XAML).

В Visual Studio Code и JetBrains Rider нет шаблонов элементов проекта для MAUI. Вы можете создать их в терминале с помощью следующей команды:

```
dotnet new maui-dict-xaml --name Northwind.xaml
```

2. Добавьте в этом файле в элемент `ResourceDictionary` разметку, определяющую линейную градиентную кисть с ключом `Rainbow`:

```
<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
                    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                    x:Class="Northwind.Maui.Client.Resources.Styles.Northwind">

  <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>

</ResourceDictionary>
```

3. В файле `App.xaml` добавьте новую запись в объединенные словари ресурсов, чтобы подключить файл ресурсов `Northwind.xaml`, расположенный в папке `Styles`:

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<Application xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:Northwind.Maui.Client"
             x:Class="Northwind.Maui.Client.App">

  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Resources/Styles/Colors.xaml" />
        <ResourceDictionary Source="Resources/Styles/Styles.xaml" />
        <ResourceDictionary Source="Resources/Styles/Northwind.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Обращение к общим ресурсам

Теперь можно использовать созданную кисть на страницах приложения.

1. В файле `CategoriesPage.xaml` измените элемент `ContentPage`, чтобы установить его фон на ресурс `Rainbow`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Northwind.Maui.Client.CategoriesPage"
             Background="{StaticResource Rainbow}"
             Title="Categories">
```

Расширение StaticResource означает, что ресурс загружается один раз при запуске приложения. Если он изменится позже, обновление не произойдет.

2. Запустите проект `Northwind.Maui.Client` с отладкой.
3. Перейдите в раздел Categories и убедитесь, что фон страницы оформлен как градиент.
4. Закройте приложение.

Динамическое изменение общих ресурсов

Теперь создадим страницу настроек, позволяющую пользователю переключать цветовую тему интерфейса (светлую, темную или системную) во время работы приложения.

1. В папке `Resources\Styles` создайте файл `LightDarkModeColors.xaml`, выбрав шаблон проекта .NET MAUI Resource Dictionary (XAML).
2. Откройте его и в элемент `ResourceDictionary` добавьте разметку, определяющую набор цветов для светлой и темной тем:

```
<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
           xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
           x:Class="Northwind.Maui.Client.Resources.Styles.LightDarkModeColors">

  <Color x:Key="LightPageBackgroundColor">White</Color>
  <Color x:Key="LightNavigationBarColor">AliceBlue</Color>
  <Color x:Key="LightPrimaryColor">WhiteSmoke</Color>
  <Color x:Key="LightSecondaryColor">Black</Color>
  <Color x:Key="LightPrimaryTextColor">Black</Color>
  <Color x:Key="LightSecondaryTextColor">White</Color>
  <Color x:Key="LightTertiaryTextColor">Gray</Color>
  <Color x:Key="DarkPageBackgroundColor">Black</Color>
  <Color x:Key="DarkNavigationBarColor">Teal</Color>
  <Color x:Key="DarkPrimaryColor">Teal</Color>
  <Color x:Key="DarkSecondaryColor">White</Color>
  <Color x:Key="DarkPrimaryTextColor">White</Color>
  <Color x:Key="DarkSecondaryTextColor">White</Color>
  <Color x:Key="DarkTertiaryTextColor">WhiteSmoke</Color>

</ResourceDictionary>
```

3. В той же папке создайте еще один словарь ресурсов, `DarkModeTheme.xaml`.
4. Откройте его и в элемент `ResourceDictionary` добавьте разметку, определяющую набор цветов для темной темы:

```
<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
                 xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                 x:Class="Northwind.Maui.Client.Resources.Styles.DarkModeTheme">
```

```
  <Style TargetType="Shell">
      <Setter Property="FlyoutBackgroundColor"
                Value="{StaticResource DarkNavigationBarColor}" />
  </Style>

  <Style TargetType="ContentPage">
    <Setter Property="BackgroundColor"
      Value="{StaticResource DarkPageBackgroundColor}" />
  </Style>

  <Style TargetType="Button">
    <Setter Property="BackgroundColor"
      Value="{StaticResource DarkPrimaryColor}" />
    <Setter Property="TextColor"
      Value="{StaticResource DarkSecondaryColor}" />
    <Setter Property="HeightRequest" Value="45" />
    <Setter Property="WidthRequest" Value="190" />
    <Setter Property="CornerRadius" Value="18" />
  </Style>

</ResourceDictionary>
```

5. В той же папке создайте еще один словарь ресурсов, `LightModeTheme.xaml`.
6. Откройте его и в элемент `ResourceDictionary` добавьте разметку, определяющую набор цветов для светлой темы:

```
<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
                xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                x:Class="Northwind.Maui.Client.Resources.Styles.LightModeTheme">

  <Style TargetType="Shell">
    <Setter Property="FlyoutBackgroundColor"
      Value="{StaticResource LightNavigationBarColor}" />
  </Style>

  <Style TargetType="ContentPage">
    <Setter Property="BackgroundColor"
      Value="{StaticResource LightPageBackgroundColor}" />
  </Style>

  <Style TargetType="Button">
    <Setter Property="BackgroundColor"
      Value="{StaticResource LightPrimaryColor}" />
    <Setter Property="TextColor"
      Value="{StaticResource LightSecondaryColor}" />
    <Setter Property="HeightRequest" Value="45" />
    <Setter Property="WidthRequest" Value="190" />
    <Setter Property="CornerRadius" Value="18" />
  </Style>

</ResourceDictionary>
```

7. В той же папке создайте еще один словарь ресурсов, `SystemModeTheme.xaml`.
8. Откройте его и в элемент `ResourceDictionary` добавьте стили, которые будут автоматически адаптироваться под системные настройки операционной системы:

```
<?xml version="1.0" encoding="utf-8" ?>
<ResourceDictionary xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
                    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                    x:Class="Northwind.Maui.Client.Resources.Styles.SystemModeTheme">

  <Style TargetType="Shell">
    <Setter Property="FlyoutBackgroundColor"
      Value="{AppThemeBinding Light={StaticResource LightNavigationBarColor},
        Dark={StaticResource DarkNavigationBarColor}}" />
  </Style>

  <Style TargetType="ContentPage">
    <Setter Property="BackgroundColor"
      Value="{AppThemeBinding Light={StaticResource LightPageBackgroundColor},
        Dark={StaticResource DarkPageBackgroundColor}}" />
  </Style>

  <Style TargetType="Button">
    <Setter Property="BackgroundColor"
      Value="{AppThemeBinding Light={StaticResource LightPrimaryColor},
        Dark={StaticResource DarkPrimaryColor}}" />
    <Setter Property="TextColor"
      Value="{AppThemeBinding Light={StaticResource LightSecondaryColor},
        Dark={StaticResource DarkSecondaryColor}}" />
    <Setter Property="HeightRequest" Value="45" />
    <Setter Property="WidthRequest" Value="190" />
    <Setter Property="CornerRadius" Value="18" />
  </Style>

</ResourceDictionary>
```

В этом словаре используется расширение {AppThemeBinding} для динамической привязки к двум предопределенным темам — Light и Dark. Оно автоматически подстраивается под настройки системы.

9. В файле `App.xaml` добавьте определенные ранее цвета и ссылки на ресурсы тем:

```
<ResourceDictionary.MergedDictionaries>
  <ResourceDictionary Source="Resources/Styles/Colors.xaml" />
  <ResourceDictionary Source="Resources/Styles/Styles.xaml" />
  <ResourceDictionary Source="Resources/Styles/Northwind.xaml" />
  <ResourceDictionary Source="Resources/Styles/LightDarkModeColors.xaml" />
  <ResourceDictionary Source="Resources/Styles/SystemModeTheme.xaml" />
</ResourceDictionary.MergedDictionaries>
```

Дельный совет: файл ресурсов SystemModeTheme.xaml использует цвета из файла LightDarkModeColors.xaml, поэтому порядок их объявления в разметке имеет значение.

10. В корне проекта создайте папку `Controls`.
11. В этой папке создайте файл класса `ThemeEnum.cs` и определите в нем перечисление с тремя значениями: `System`, `Light`, `Dark` — для выбора темы:

```
namespace Northwind.Maui.Client.Controls;

public enum Theme
{
  System,
  Light,
  Dark
}
```

12. В той же папке создайте файл `EnumPicker.cs` и реализуйте класс, наследующий элемент управления `Picker`. Он должен поддерживать привязку к перечислению и отображать раскрывающийся список возможных тем:

```
using System.Reflection; // Для метода GetTypeInfo

namespace Northwind.Maui.Client.Controls;

public class EnumPicker : Picker
{
  public Type EnumType
  {
    set => SetValue(EnumTypeProperty, value);
    get => (Type)GetValue(EnumTypeProperty);
  }

  public static readonly BindableProperty EnumTypeProperty =
    BindableProperty.Create(
      propertyName: nameof(EnumType),
      returnType: typeof(Type),
      declaringType: typeof(EnumPicker),
      propertyChanged: (bindable, oldValue, newValue) =>
    {
      EnumPicker picker = (EnumPicker)bindable;

      if (oldValue != null)
      {
        picker.ItemsSource = null;
      }

      if (newValue != null)
      {
        if (!((Type)newValue).GetTypeInfo().IsEnum)
          throw new ArgumentException(
            "EnumPicker: EnumType property must be enumeration type");
```

```
        picker.ItemsSource = Enum.GetValues((Type)newValue);
      }
    });
}
```

13. В файле `SettingsPage.xaml`:
    - импортируйте локальное пространство имен, чтобы использовать элемент управления `EnumPicker`;
    - добавьте пространство имен `ios`, необходимое для настройки специфичных для iOS параметров;
    - измените заголовок страницы на Settings;
    - добавьте элемент `EnumPicker` для выбора темы:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:Northwind.Maui.Client.Controls"
             xmlns:ios="clr-namespace:Microsoft.Maui.Controls.
➥ PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"
             x:Class="Northwind.Maui.Client.SettingsPage"
             Title="Settings">

  <VerticalStackLayout HorizontalOptions="Center">
    <local:EnumPicker ios:Picker.UpdateMode="WhenFinished"
                      EnumType="{x:Type local:Theme}"
                      Title="Select Theme"
                      SelectedIndexChanged="ThemePicker_SelectionChanged"
                      Loaded="ThemePicker_Loaded"
                      x:Name="ThemePicker" />
  </VerticalStackLayout>
</ContentPage>
```

14. В файле `SettingsPage.xaml.cs` добавьте обработчик событий для `Picker`, чтобы менять тему при выборе пользователем:

```
using Northwind.Maui.Client.Controls; // Для перечисления Theme
using Northwind.Maui.Client.Resources.Styles; // Для темы DarkModeTheme

namespace Northwind.Maui.Client;

public partial class SettingsPage : ContentPage
{
  public SettingsPage()
  {
    InitializeComponent();
  }

  private void ThemePicker_SelectionChanged(object sender, EventArgs e)
  {
    Picker picker = sender as Picker;
    Theme theme = (Theme)picker.SelectedItem;
```

```
    ICollection<ResourceDictionary> resources =
      Application.Current.Resources.MergedDictionaries;

    if (resources is not null)
    {
      resources.Clear();
      resources.Add(new Resources.Styles.Northwind());
      resources.Add(new LightDarkModeColors());

      ResourceDictionary themeResource = theme switch
      {
        Theme.Dark  => new DarkModeTheme(),
        Theme.Light => new LightModeTheme(),
        _           => new SystemModeTheme()
      };

      resources.Add(themeResource);
    }
  }

  private void ThemePicker_Loaded(object sender, EventArgs e)
  {
    ThemePicker.SelectedItem = Theme.System;
  }
}
```

15. Запустите проект `Northwind.Maui.Client` как минимум на одном компьютере и мобильном устройстве.
16. Убедитесь, что интерфейс приложения отображается в светлой теме (пример для Windows — на рис. 16.11).

Рис. 16.11. Кнопка в светлом режиме в Windows

17. Оставьте приложение работающим, запустите в Windows приложение Settings (Настройки), перейдите в раздел Personalization ▸ Colors (Персонализация ▸ Цвета) и в раскрывающемся списке Choose your mode (Выбор цвета) выберите вариант Dark (Темный) (рис. 16.12).

Рис. 16.12. Переключение цветового оформления системы в настройках Windows

18. Не закрывая окно Settings (Настройки) открытым, переключитесь на приложение и убедитесь, что оно автоматически изменило цветовую схему на темную (рис. 16.13).
19. Закройте приложение.
20. В окне Settings (Настройки) снова переключитесь в режим Light (Светлый).

Рис. 16.13. Темный режим в приложении

Дельный совет: ресурсом может быть любой объект. Чтобы использовать его в приложении, добавьте его в файл App.xaml и присвойте уникальный ключ.

- Для установки значения один раз при запуске приложения используйте расширение {StaticResource key}.
- Для динамического обновления ресурса во время работы приложения используйте расширение {DynamicResource key}.
- Чтобы получить ресурс в коде, вызовите метод TryGetValue через свойство Resources.
- Если обратиться к Resources[key] напрямую, будут доступны только ресурсы, определенные в этом словаре, но не в объединенных.

Ресурсы можно определять в любом элементе XAML, а не только на уровне всего приложения. Например, если ресурс нужен только в `MainPage`, его можно объявить прямо там. Вы также можете динамически загружать файлы XAML на этапе выполнения программы.

В копилку знаний: подробную информацию о словарях ресурсов в .NET MAUI можно найти в официальной документации по адресу https://learn.microsoft.com/ru-ru/dotnet/maui/fundamentals/resource-dictionaries. В частности, обратите внимание на раздел о поиске ресурсов.

Привязка данных

При создании графических пользовательских интерфейсов часто возникает необходимость привязать свойство одного элемента управления к другому или к каким-либо данным.

Привязка к элементам

Простейший способ связывания — привязка значений между двумя элементами. Один элемент выступает в качестве источника данных, а другой — в качестве их получателя.

Выполним следующие действия.

1. В файле `CategoriesPage.xaml` под существующей кнопкой в макете вертикального стека добавьте:
   - метку для инструкций;
   - еще одну метку для отображения текущей степени поворота;

- ползунок для выбора угла поворота;
- радужный квадрат, который будет вращаться:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Northwind.Maui.Client.CategoriesPage"
             Background="{StaticResource Rainbow}"
             Title="Categories">

  <VerticalStackLayout>

    <Button Text="Click Me" WidthRequest="100"
            x:Name="ClickMeButton"
            Clicked="ClickMeButton_Clicked" />

    <Label Margin="10">
      Use the slider to rotate the square:
    </Label>

    <Label BindingContext="{x:Reference Name=SliderRotation}"
           Text="{Binding Path=Value, StringFormat='{0:N0} degrees'}"
           FontSize="30" HorizontalTextAlignment="Center" />

    <Slider Value="0" Minimum="0" Maximum="180"
            x:Name="SliderRotation" Margin="10,0" />

    <Rectangle HeightRequest="200" WidthRequest="200"
               Fill="{StaticResource Rainbow}"
               BindingContext="{x:Reference Name=SliderRotation}"
               Rotation="{Binding Path=Value}" />

  </VerticalStackLayout>

</ContentPage>
```

Обратите внимание, что текст метки и угол поворота прямоугольника привязаны к положению ползунка с помощью контекста привязки и расширения разметки {Binding}.

2. Запустите проект `Northwind.Maui.Client`.
3. Перейдите на страницу Categories.
4. Нажав и удерживая кнопку мыши, перетащите ползунок, чтобы вращать радужный квадрат (рис. 16.14).
5. Закройте приложение.

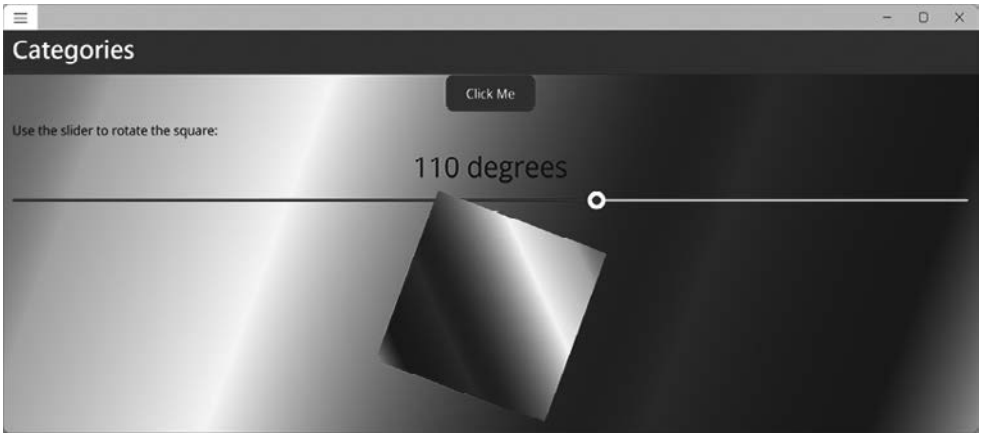


Рис. 16.14. Привязка положения ползунка к метке и углу поворота прямоугольника в Windows

Практика и исследование

Давайте проверим ваши знания! Ответьте на приведенные далее вопросы и изучите ресурсы с дополнительной информацией по темам из этой главы, перейдя по указанным ссылкам.

Упражнение 16.1. Проверьте свои знания

Ответьте на следующие вопросы.

1. Опишите четыре категории компонентов пользовательского интерфейса .NET MAUI.
2. В чем преимущество компонента `Shell` и какие виды пользовательского интерфейса он реализует?
3. Как разрешить пользователю выполнять действие с ячейкой в представлении списка?
4. В каких случаях используется компонент `Entry`, а в каких — `Editor`?
5. Какое влияние оказывает установка свойства `IsDestructive = true` для пункта меню в контекстных действиях ячейки?
6. Почему оболочка с одной страницей содержимого не отображает навигацию?
7. Чем отличаются свойства `Margin` и `Padding` у элемента, например `Button`?
8. Как прикрепить обработчики событий к объекту через XAML?
9. Для чего используются стили XAML?
10. Где можно определить ресурсы?

Упражнение 16.2. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md#chapter-16---building-mobile-and-desktop-apps-using-net-maui.

Резюме

Из этой главы вы узнали:

- как создать кросс-платформенное мобильное и классическое приложение с помощью платформы .NET MAUI;
- как определять общие ресурсы и ссылаться на них;
- как выполнять привязку данных к стандартным элементам управления.

В послесловии рассмотрены дальнейшие шаги в изучении приложений и сервисов .NET.

Послесловие

Я хотел, чтобы эта книга отличалась от других, представленных на рынке. Я надеюсь, что она покажется вам свежей, увлекательной, полной практических разборов каждой темы.

Облачно-нативная разработка с .NET Aspire

В книге рассмотрены многие технологии, используемые при разработке облачных нативных приложений, включая gRPC, кэширование различных типов и отказоустойчивость с помощью Polly.

Чем больше технологий применяется, тем сложнее управлять средой разработки. Ошибки в строках подключения, секретных ключах, номерах портов, ограничении скорости и настройках кэширования могут приводить к сбоям или неожиданным результатам, затрудняя их диагностику.

Четырнадцатого ноября 2023 года на конференции .NET Conf команда .NET анонсировала .NET Aspire. И уже в 2024 году состоялся выпуск версии Aspire 1.0.

Чтобы познакомиться с .NET Aspire SDK, выберите соответствующую рабочую нагрузку на вкладке Individual components (Отдельные компоненты) программы установки Visual Studio 2022.

.NET Aspire — это стек для создания отказоустойчивых, наблюдаемых и настраиваемых облачных приложений на .NET. Он включает в себя следующее.

- *Компоненты Aspire* — пакеты, основанные на зрелых технологиях (Redis, OpenTelemetry), но адаптированные для Aspire и упрощающие настройку.

Они предоставляют стандартизированную функциональность, в том числе обнаружение сервисов, телеметрию, восстановление и проверку работоспособности.

- *Шаблоны Aspire Starter и Empty project* — дают возможность быстро начать работу с решениями, включающими Blazor-фронтенд для мониторинга всех созданных вами сервисов и приложений, веб-API и сервис управления. Общий проект позволяет корректировать настройки по умолчанию, которые будут применяться ко всем проектам, с управлением из хоста Aspire.
- *Aspire Orchestration* — инструменты для запуска многопроектных решений с учетом зависимостей. Пользовательские методы расширения позволяют настраивать все компоненты в таких проектах в соответствии с лучшими практиками, и вы можете изменить их под свои особые требования.

Подробнее о .NET Aspire см. в документации по адресу https://learn.microsoft.com/ru-ru/dotnet/aspire/.

Главу о .NET Aspire я напишу в третьей книге моей трилогии о .NET 8, «.NET 8: лучшие практики и паттерны проектирования, отладки и тестирования».

Ознакомление с проектом Survey Project Challenge

Для освоения современных технологий .NET, которыми должны владеть разработчики .NET в наши дни, полезно создать реальный проект, сочетающий разные технологии и навыки. Он должен быть как можно более реалистичным, крутым, веселым и практичным, а также реализованным другими пользователями, благодаря которым мы можем черпать вдохновение.

Survey Project Challenge — это учебный проект с реальными задачами, который позволяет потренироваться в создании комплексных решений.

Подробнее о нем вы можете прочитать по адресу https://github.com/markjprice/apps-services-net8/blob/main/docs/ch17-survey-project.md.

Следующие шаги на пути изучения C# и .NET

Эта книга не исчерпывающее руководство. Надеюсь, что примечания, советы и ссылки на GitHub помогут вам углубить знания: https://github.com/markjprice/apps-services-net8/blob/main/docs/book-links.md.

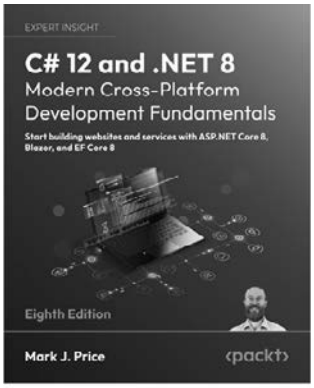
Работа над следующей версией книги уже идет. Ее выпуск планируется через месяц после выхода .NET 10 GA (ноябрь 2025 года). Хотя я не ожидаю появления новых возможностей на уровне унификации моделей размещения Blazor, все же очень надеюсь, что .NET 9 и .NET 10 действительно улучшат платформу по всем фронтам. Если у вас есть предложения по совершенствованию книги или исправлению ошибок, оставляйте их на Discord-канале или в репозитории по адресу https://github.com/markjprice/apps-services-net8.

Сопутствующие книги для продолжения обучения

Эта книга — вторая из трилогии, посвященной платформе .NET 8.

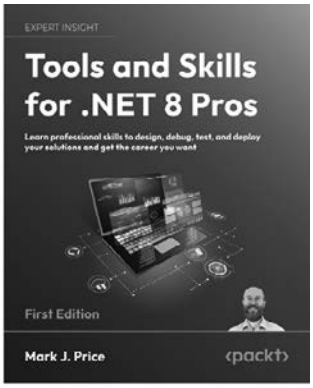
1. В первой книге рассматриваются основы языка C# и библиотеки .NET и ASP.NET Core для разработки веб-приложений. Она предназначена для последовательного чтения, поскольку навыки и знания, полученные в предыдущих главах, дополняются и необходимы для понимания материала последующих глав.
2. Во второй книге рассмотрены более узкие темы, такие как интернационализация, а также популярные пакеты сторонних разработчиков, в том числе Serilog и NodaTime. Вы научитесь разрабатывать нативные сервисы с AOT-компиляцией при помощи минимальных API ASP.NET Core и узнаете, как повысить производительность, масштабируемость и надежность посредством кэширования данных, очередей и фоновых сервисов. С технологиями GraphQL, gRPC, SignalR и Azure Functions вы реализуете дополнительные сервисы. Наконец, вы узнаете, как с помощью Blazor и .NET MAUI создавать графические пользовательские интерфейсы для сайтов, классических и мобильных приложений.
3. В третьей книге рассматриваются важные инструменты и навыки, которыми должен обладать каждый профессиональный разработчик .NET. Сюда входят паттерны проектирования и архитектуры решений, отладка, анализ памяти, все важные виды тестирования — модульное, производительности, системное, веб- и мобильное, а также инструменты хостинга и развертывания, такие как Docker и Azure Pipelines. Наконец, мы рассмотрим, как подготовиться к собеседованию на замещение вакансии разработчика .NET.

Краткое описание трилогии .NET 8 и ее наиболее важных тем показано на рис. П.1.



- **Язык C#**, в том числе новые возможности версии C# 12, объектно-ориентированное программирование (ООП), отладка и модульное тестирование.
- **Библиотеки .NET**, в том числе касающиеся работы с числами, текстом, регулярными выражениями, коллекциями, файловым вводом-выводом и данными с помощью EF Core и SQLite.
- **Разработка сайтов и веб-сервисов** с помощью ASP.NET Core и Blazor

- **Дополнительные библиотеки:** интернационализация, многозадачность и пакеты сторонних разработчиков.
- **Дополнительная информация об обработке данных:** SQL Server и Cosmos DB.
- **Дополнительная информация о сервисах:** кэширование, очереди, минимальные API, GraphQL, gRPC, SignalR и Azure Functions.
- **Дополнительная информация о пользовательских интерфейсах:** ASP.NET Core и .NET MAUI



- **Инструменты:** интегрированные среды разработки, отладка, анализ памяти и ИИ-ассистенты.
- **Тестирование:** модульное, интеграционное, производительности, системное и веб-, в том числе тесты внедрения зависимостей и инверсий контроля.
- **Проектирование:** паттерны и архитектуры.
- **Развертывание:** непрерывная интеграция/непрерывное развертывание, хостинг в Azure, конвейеры развертывания и автоматизация.
- **Карьера:** подготовка к собеседованию

Рис. П.1. Трилогия для изучения C# 12 и .NET 8

Желаю вам удачи во всех ваших проектах .NET!

Приложение. Ответы к разделам «Проверьте свои знания»

В этом приложении содержатся ответы на вопросы раздела «Проверьте свои знания» в конце каждой главы.

Глава 1

1. Для чего в файлы проекта добавляется показанный ниже код? А когда его не следует приводить?

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

Ответ. Компилятор обрабатывает предупреждения как ошибки, которые необходимо исправить. По умолчанию такие предупреждения компилятор игнорирует. Активация параметра `TreatWarningsAsErrors` вынуждает разработчика устранить все проблемы в проекте, прежде чем его можно будет скомпилировать и запустить.

В проектах .NET 7 с текущими инструментами gRPC данный параметр недоступен, поскольку они генерируют типы с именами из строчных букв. В версии .NET 7 и более поздней такая ситуация приводит к выводу предупреждений. Подробнее об этом типе предупреждений можно узнать по адресу https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/compiler-messages/warning-waves#cs8981---the-type-name-only-contains-lower-cased-ascii-characters.

Если после активации данного параметра будет выводиться излишне много уведомлений об ошибках, определенные предупреждения можно отключить, указав параметр `<WarningsNotAsErrors>` со списком кодов предупреждений через запятую:

```
<WarningsNotAsErrors>0219,CS8981</WarningsNotAsErrors>
```

2. Для какой технологии сервиса требуется протокол HTTP/2 или версии выше?

   Ответ. Для gRPC-сервиса.

3. В 2010 году в вашей компании был создан сервис на базе .NET Framework и WCF. На какую технологию лучше всего его перенести и почему?

   Ответ. Два варианта переноса сервиса и клиента WCF: 1) проект Core WCF с открытым исходным кодом или 2) повторная реализация сервиса и клиента с помощью gRPC. Если функционала Core WCF для проекта достаточно и вы хотите затратить минимум усилий, остановите свой выбор на нем. Если же Core WCF не поддерживает все необходимые вам функциональные возможности и у вас есть время и ресурсы для повторной реализации сервиса, переходите на gRPC.

4. Какой редактор кода или среду разработки следует установить для программирования .NET-проектов?

   Ответ. Большинству разработчиков подойдут программы Visual Studio 2022 и Visual Studio Code. Так вы сможете для разных типов проектов выбирать наиболее подходящую программу. Опытные разработчики часто оценивают преимущества сторонних решений, таких как JetBrains Rider.

5. Чего следует остерегаться при создании ресурсов Azure?

   Ответ. Многие ресурсы Azure предоставляются за определенную плату. В целях экономии необходимо оперативно удалять неиспользуемые ресурсы.

Глава 2

1. На какой NuGet-пакет следует ссылаться в проекте .NET, чтобы добиться максимальной производительности при работе с БД SQL Server?

   Ответ. Microsoft.Data.SqlClient.

2. Какой способ конфигурации строки подключения к БД наиболее безопасен?

   Ответ. Создайте экземпляр класса `SqlConnectionStringBuilder`, настройте его свойства, такие как `DataSource`, `InitialCatalog`, `UserID` и `Password`, а затем считайте его свойство `ConnectionString`.

3. Что необходимо указывать в качестве префикса параметров и переменных T-SQL?

   Ответ. У параметров и переменных T-SQL должен быть префикс в виде символа `@`.

4. Что необходимо сделать перед считыванием выходного параметра команды, выполненной с помощью метода `ExecuteReader`?

   Ответ. Перед считыванием выходного параметра необходимо закрыть объект `DataReader`.

5. К какому типу Dapper добавляет свои методы расширения?

   Ответ. Инструмент Dapper добавляет свои методы расширения к любому типу, реализующему интерфейс `IDbConnection`.

6. Какие два наиболее часто используемых метода расширения предоставляет Dapper?

   Ответ. Методы расширения `Query<T>` и `Execute`.

Глава 3

1. Для чего применяется инструмент `dotnet-ef`?

   Ответ. Инструмент `dotnet-ef` используется для выполнения задач разработки на этапе проектирования, таких как создание или применение миграций и генерация модели из существующей БД.

2. Какой тип данных применяется для свойства, представляющего таблицу, например для свойства `Products` в контексте данных?

   Ответ. `DbSet<T>`, где `T` — это тип модели сущности в таблице, например `Product`.

3. Какой тип данных используется для свойства, представляющего отношение «один ко многим», например для свойства `Products` в сущности `Category`?

   Ответ. `ICollection<T>`, где `T` — это тип модели сущности в связанной таблице, например `Product`.

4. Расскажите о соглашении первичных ключей EF Core.

   Ответ. За первичный ключ принимается свойство с именем `ID`, `Id`, `ClassNameID` или `ClassNameId`. Если тип этого свойства `tinyint`, `smallint`, `int`, `bigint` или `guid`, то это свойство также помечается как столбец `IDENTITY`.

5. В чем заключаются преимущества Fluent API перед атрибутами аннотаций?

   Ответ. Технология Fluent API будет лучше атрибутов аннотаций, если требуется уменьшить количество избыточного кода в классах сущностей, который нужен лишь в определенных случаях. Например, при создании библиотеки классов .NET Standard 2.0 для сущностей можно использовать лишь атрибуты валидации, чтобы метаданные считывались не только Entity Framework Core, но и инструментами проверки привязки модели в ASP.NET Core или классическими и мобильными приложениями .NET MAUI. Также Fluent API может понадобиться для реализации специфического функционала EF Core, например для сопоставления с таблицей или именами столбцов.

6. Для чего в типе сущности реализуется интерфейс `IMaterializationInterceptor`?

   Ответ. Перехватчики EF Core позволяют вмешиваться в операции, выполняемые с EF Core, модифицировать их или подавлять. Интерфейс `IMaterializationInterceptor` предоставляет возможность перехватывать данные

до/после создания экземпляра сущности, а также до/после инициализации ее свойств. Это удобно для настройки свойств, выполнения вычислений, проверки значений/флагов, использования фабрик для создания экземпляров, а также для управления экземплярами сущностей, отличными от стандартных, например извлеченных из кэша.

Глава 4

1. Какие пять API поддерживаются Azure Cosmos DB?

   Ответ. API, поддерживаемые Azure Cosmos DB:

   - API Core (SQL);
   - API MongoDB;
   - API Cassandra;
   - API Gremlin (графовый);
   - API Azure Table.

2. На каком уровне выбирается API: учетная запись, база данных, контейнер или раздел?

   Ответ. API выбирается на уровне учетной записи и не может быть изменен после создания. Данный выбор распространяется на все БД, контейнеры и разделы в этой учетной записи.

3. Что означает «встраивание» в контексте моделирования данных в Cosmos DB?

   Ответ. Встраивание (денормализация) означает, что связанные данные хранятся в одном документе, а не в отдельных таблицах, как в реляционной БД. Такой подход уменьшает количество операций чтения и записи, улучшая производительность.

4. В каких единицах измеряется пропускная способность Cosmos DB и что такое одна единица?

   Ответ. Пропускная способность измеряется в единицах запросов в секунду (request units per second, RU/s). Одна единица ЕЗ/с соответствует выполнению простого запроса `GET` для документа размером 1 Кб с уникальным идентификатором.

5. Какой пакет нужен для работы с ресурсами Cosmos DB программным способом?

   Ответ. `Azure.Data.Cosmos` для .NET.

6. Какой язык используется для написания пользовательских функций и хранимых процедур в Cosmos DB API Core (SQL)?

   Ответ. JavaScript.

Глава 5

1. Какие сведения можно получить о процессе?

   Ответ. Можно узнать идентификатор процесса, его имя, процент использования процессора и памяти, приоритет, статус (запущен, приостановлен и т. д.), время запуска, список потоков и загруженных модулей.

2. Насколько точен класс `Stopwatch`?

   Ответ. Класс `Stopwatch` использует высокоточную системную тактовую частоту, что позволяет измерять время с точностью до наносекунд (в зависимости от аппаратной поддержки).

3. Какой суффикс должен иметь метод, возвращающий объект `Task` или `Task<T>`?

   Ответ. К имени метода добавляется суффикс `Async`. Например, для синхронного метода `Open` будет использоваться его асинхронный аналог `OpenAsync`, который возвращает объект `Task` или `Task<T>`.

4. Какое ключевое слово должно быть указано в объявлении метода, если внутри него используется оператор `await`?

   Ответ. В объявлении метода должно быть использовано ключевое слово `async`.

5. Как создать дочернюю задачу?

   Ответ. Для создания дочерней задачи нужно вызвать метод `Task.Factory.StartNew` с параметром `TaskCreationOptions.AttachToParent`.

6. Почему рекомендуется избегать ключевого слова `lock`?

   Ответ. Ключевое слово `lock` не позволяет задать тайм-аут, что может привести к взаимным блокировкам. Вместо него рекомендуется использовать метод `Monitor.Enter` с указанием тайм-аута (параметр `TimeSpan`), а затем явно вызвать `Monitor.Exit` для снятия блокировки после выполнения работы.

7. В каких случаях следует использовать класс `Interlocked`?

   Ответ. Класс `Interlocked` используется для безопасного изменения целочисленных значений и чисел с плавающей точкой, которые могут использоваться одновременно несколькими потоками.

8. Когда класс `Mutex` предпочтительнее класса `Monitor`?

   Ответ. Класс `Mutex` используется, когда необходимо синхронизировать доступ к ресурсу между несколькими процессами. В то время как класс `Monitor` работает только в пределах одного процесса.

9. Каковы преимущества применения ключевых слов `async` и `await` на стороне сервера в веб-приложениях и веб-сервисах?

   Ответ. Использование ключевых слов `async` и `await` на сервере улучшает масштабируемость веб-приложений и сервисов, позволяя эффективно управлять

потоками и не блокировать их во время выполнения длительных операций. Однако производительность каждого отдельного запроса в этом случае не увеличивается из-за дополнительной нагрузки на процессор и оперативную память, связанную с переключением контекста между потоками.

10. Можно ли отменить задачу? Если да, то каким образом?

    Ответ. Да, задачу можно отменить. См. инструкцию по адресу https://learn.microsoft.com/ru-ru/dotnet/csharp/asynchronous-programming/cancel-an-async-task-or-a-list-of-tasks.

Глава 6

1. Какой пакет сторонних разработчиков скачивали в NuGet чаще всего (за все время)?

   Ответ. `Newtonsoft.Json`.

2. Какой метод в классе `Image` из библиотеки ImageSharp позволяет, помимо прочего, изменить размер изображения и преобразовать его в черно-белое?

   Ответ. `Mutate`.

3. В чем главное преимущество библиотеки Serilog в рамках журналирования?

   Ответ. Библиотека `Serilog` позволяет записывать сериализованные структурированные данные в журнал. Символ `@` перед параметром указывает `Serilog` на сериализацию переданного объекта, а не просто на вызов метода `ToString`. В дальнейшем этот объект можно использовать для поиска и сортировки в журналах.

4. Что означает термин *«приемник»* в Serilog?

   Ответ. В `Serilog` *приемник* представляет собой «пункт назначения» журнала. Например, есть приемник для вывода в консоль, записи в файл, передачи на SQL Server и в Azure Application Insights.

5. Всегда ли следует использовать пакет AutoMapper для сопоставления объектов?

   Ответ. Нет, не всегда. Ведутся споры о его целесообразности, подробнее об этом можно прочитать по адресу https://www.anthonysteele.co.uk/AgainstAutoMapper.html.

6. Какой метод библиотеки FluentAssertions следует вызвать для простого утверждения значения?

   Ответ. `Should`.

7. Какой метод библиотеки FluentAssertions проверяет, все ли элементы последовательности соответствуют условию, например тому, что строки должны содержать не более шести символов?

   Ответ. `OnlyContain`.

8. От какого класса библиотеки FluentValidation нужно наследовать пользовательский валидатор?

   Ответ. `AbstractValidator<T>`.

9. Можно ли в FluentValidation задать правило, которое будет применяться только при определенных условиях?

   Ответ. В FluentValidation для применения правила при определенных условиях вызывается метод `When` с лямбда-выражением, возвращающим значение `true`. При необходимости можно также использовать метод `Otherwise`, если лямбда-выражение возвращает значение `false`.

10. Какой интерфейс и с какими методами в QuestPDF нужно реализовать для создания PDF-документа?

    Ответ. Необходимо реализовать интерфейс `IDocument`, а также его методы `Compose` и `GetMetadata`.

Глава 7

1. В чем разница между локализацией, глобализацией и интернационализацией?

   Ответ

   - Локализация касается пользовательского интерфейса приложения. Она определяется нейтральной (только язык) или специфической (язык и регион) культурой. Например, текст может быть представлен на разных языках: First name на английском и Prénom на французском.
   - Глобализация влияет на данные приложения. Например, для правильного отображения денежных единиц важно учитывать не только язык, но и регион: для британского английского это будет `en-GB`, а для канадского французского — `fr-CA`. Культура должна быть специфической, потому что для корректного отображения десятичных значений, отформатированных как денежная единица, важно учитывать, какая именно используется: канадские доллары для культуры `fr-CA` или евро для французской культуры.
   - Интернационализация включает в себя и локализацию, и глобализацию и предполагает подготовку приложения к поддержке множества языков и регионов.

2. Какое наименьшее измерение времени доступно в .NET?

   Ответ. Наносекунды.

3. Сколько времени длится один тик в .NET?

   Ответ. Один тик равен 100 наносекундам.

4. В каких случаях следует использовать значение `DateOnly` вместо `DateTime`?

   Ответ. Используйте значение `DateOnly`, когда нужно работать только с датой, без учета времени или часового пояса. Это полезно, например, при работе со столбцами с датами в SQL Server.

5. Что обозначает свойство `BaseUtcOffset` для часового пояса?

   Ответ. Свойство `BaseUtcOffset` представляет собой разницу во времени между текущим часовым поясом и UTC, без учета возможных изменений, связанных с переходом на летнее время.

6. Как получить информацию о локальном часовом поясе, в котором выполняется код?

   Ответ. Для этого нужно обратиться к свойству `TimeZoneInfo.Local`.

7. Что означает свойство `Kind` у значения `DateTime`?

   Ответ. Свойство `Kind` определяет тип значения `DateTime` — `Unspecified`, `Utc` или `Local`.

8. Как изменить текущую культуру для исполняемого кода?

   Ответ. Для этого можно установить свойства `CultureInfo.CurrentCulture`, `CultureInfo.CurrentUICulture`, `Thread.CurrentThread.CurrentCulture` или `Thread.CurrentThread.CurrentUICulture` нужного экземпляра `CultureInfo`.

9. Каков ISO-код валлийского языка?

   Ответ. ISO-код валлийского языка в Великобритании — `cy-GB`.

10. Как используются резервные файлы ресурсов локализации?

    Ответ. Файлы ресурсов имеют расширение `.resx` и содержат информацию о культуре, например `PacktResources.en-GB.resx` для британского английского или `PacktResources.da-DK.resx` для датского языка. Поиск резервных файлов происходит по порядку от конкретной культуры к нейтральной и инвариантной культуре. Например, если культура потока — `en-AU`, поиск будет осуществляться в следующем порядке:

    - `PacktResources.en-AU.resx` (австралийский английский);
    - `PacktResources.en.resx` (английский без привязки к региону);
    - `PacktResources.resx` (инвариантная культура).

Глава 8

1. Назовите шесть методов, которые могут быть указаны в HTTP-запросе.

   Ответ. В HTTP-запросе могут быть указаны следующие методы:

   - `GET` — запрашивает данные с сервера;
   - `HEAD` – запрашивает только заголовки ресурса без тела ответа, полезен для проверки доступности или обновлений;
   - `POST` — отправляет данные на сервер;
   - `PUT` — обновляет ресурс на сервере;
   - `DELETE` — удаляет ресурс;
   - `PATCH` — частично обновляет ресурс.

А также:

- `TRACE` — используется для диагностики, возвращает клиенту запрос в том виде, в котором он был получен сервером;
- `OPTIONS` — запрашивает информацию о возможностях сервера;
- `CONNECT` — устанавливает туннельное соединение через HTTP-прокси, применяется для HTTPS-запросов.

2. Перечислите шесть кодов HTTP-ответа и их описания.

   Ответ. Шесть кодов ответа HTTP:

   - `200 OK` — успешный запрос;
   - `201 Created` — ресурс успешно создан;
   - `301 Moved Permanently` — перемещено навсегда;
   - `400 Bad Request` — некорректный запрос;
   - `404 Not Found` — ресурс не найден;
   - `500 Internal Server Error` — внутренняя ошибка сервера.

   А также:

   - `101 Switching Protocols` — переключение протоколов (например, с HTTP на WebSocket);
   - `202 Accepted` — принято;
   - `204 No Content` — нет содержимого;
   - `304 Not Modified` — не изменялось;
   - `401 Unauthorized` — требуется авторизация;
   - `403 Forbidden` — доступ запрещен;
   - `406 Not Acceptable` — неприемлемо (например, запрос формата ответа, который не поддерживается сайтом);
   - `503 Service Unavailable` — сервис недоступен.

3. Чем технология минимальных API в ASP.NET Core отличается от технологии веб-API в ASP.NET Core?

   Ответ. Сервису ASP.NET Core веб-API требуется класс контроллера для определения конечных точек. В сервисах, построенных на минимальных API ASP.NET Core, контроллеры не используются. Минимальные API позволяют создавать более простые сервисы, требующие меньше кода, что и отражено в их названии.

4. Как сопоставить HTTP-запрос `PUT` к `api/customers` с блоком лямбда-выражения, используя минимальные API в ASP.NET Core?

Ответ. Для этого используется метод `MapPut`, которому передается шаблон маршрута и блок лямбда-выражения:

```
app.MapPut("api/customers", () =>
  {
    // Выполнение каких-либо задач
  });
```

5. Как сопоставить метод/параметр лямбды со значением из маршрута, строки запроса или тела запроса в минимальных API ASP.NET Core?

   Ответ. Пометить параметр атрибутом `[FromRoute]`, `[FromQuery]` или `[FromBody]`.

6. Улучшает ли механизм CORS безопасность веб-сервиса?

   Ответ. Нет. Механизм CORS ослабляет ограничения браузера на междоменные запросы, позволяя определенным клиентам обращаться к ресурсу. Он не усиливает защиту веб-сервиса и не влияет на клиентов, работающих вне браузера.

7. Вы добавили в файл `Program.cs` код для включения HTTP-журналирования, но запросы и ответы не протоколируются. Почему возникла эта проблема и как ее устранить?

   Ответ. Скорее всего, уровень журналирования установлен на `Warning` (2) или `Error` (1). Для протоколирования HTTP-запросов требуется уровень `Information` (3) или выше. Проверьте файл `appsettings.json` и убедитесь, что указана корректная настройка:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.HttpLogging": "Information"
    }
  }
}
```

8. Как ограничить количество запросов для конкретного клиента с помощью пакета `AspNetCoreRateLimit`?

   Ответ. Во-первых, укажите заголовок HTTP-запроса, по которому будет идентифицироваться клиент:

```
"ClientRateLimiting": {
  "ClientIdHeader": "X-Client-Id",
```

   Во-вторых, присвойте каждому клиенту уникальный идентификатор, который тот должен указывать в заголовке своих запросов, например GUID соответствующей учетной записи.

В-третьих, настройте правила ограничения запросов для данного идентификатора в файле конфигурации:

```
"ClientRateLimitPolicies": {
  "ClientRules": [
    {
      "ClientId": "abc123-...",
      "Rules": [
```

9. Как ограничить количество запросов для определенной конечной точки с помощью пакета `Microsoft.AspNetCore.RateLimiting`?

   Ответ. Сначала создайте объект `RateLimiterOptions`, определив в нем политику ограничения. Зарегистрируйте политику в методе `UseRateLimiter()`. Примените ограничение к конечной точке с помощью метода `RequireRateLimiting()`.

10. Как расшифровывается аббревиатура JWT?

    Ответ. JWT (JSON Web Token) — открытый стандарт, описывающий компактный и самодостаточный способ передачи информации в формате JSON. Чаще всего используется для авторизации и безопасного обмена данными между клиентом и сервером.

Глава 9

1. Какова разница в скорости считывания 1 Мбайт данных с SSD и из оперативной памяти по результатам исследования Джеффа Дина?

   Ответ. Считывание 1 Мбайт данных с SSD-диска занимает в четыре раза больше времени, чем из оперативной памяти.

2. В чем разница между абсолютными и скользящими сроками хранения?

   Ответ. Абсолютный срок хранения определяет точный момент времени, после которого объект должен быть удален, вне зависимости от даты создания. Скользящий срок хранения начинается с момента последнего обращения к объекту и обновляется с каждым доступом, так что потенциально он может длиться вечно.

3. В каких единицах измеряется параметр `Size` кэша?

   Ответ. Параметр `Size` для кэша в памяти обычно измеряется в байтах, но могут использоваться и другие единицы.

4. В следующем операторе вы пробуете получить информацию о кэшировании в оперативной памяти, но параметр `stats` возвращает `null`. Как решить эту проблему?

   ```
   MemoryCacheStatistics? stats = _memoryCache.GetCurrentStatistics();
   ```

   Ответ. Необходимо присвоить значение `true` свойству `TrackStatistics` в объекте `MemoryCacheOptions` при регистрации класса `MemoryCache` в качестве механизма внедрения зависимостей:

```
builder.Services.AddSingleton<IMemoryCache>(new MemoryCache(
  new MemoryCacheOptions
  {
    TrackStatistics = true,
    // ...установка остальных параметров
  }));
```

5. Какие типы данных могут храниться: а) в кэше оперативной памяти и б) в распределенном кэше?

   Ответ. В кэше оперативной памяти могут храниться любые типы данных. В распределенном кэше могут храниться только байтовые массивы.

6. Чем различаются шаблоны Retry и Circuit Breaker?

   Ответ. Шаблон Retry позволяет клиентам автоматически повторять неудачные операции с расчетом на то, что после небольшой задержки следующая попытка будет успешной. Шаблон Circuit Breaker предотвращает выполнение дальнейших вызовов при достижении определенного порога ошибок. Полезный способ определить, что ошибка постоянна/достаточно длительная, и прекратить попытки.

7. В случае прямого обмена RabbitMQ какой ключ маршрутизации должен использоваться для очереди с именем `product`?

   Ответ. Ключ маршрутизации должен совпадать с именем очереди, поэтому для очереди с названием `product` ключ маршрутизации будет `product`.

8. Чем отличается веерный обмен от обмена через топики?

   Ответ. Веерный обмен отправляет сообщение всем подписанным очередям, игнорируя ключ маршрутизации. Обмен через топики, наоборот, доставляет сообщения на основе ключа маршрутизации и условий, заданных в привязке между обменом и очередью.

9. На каком порте RabbitMQ прослушивает соединения по умолчанию?

   Ответ. `5672`.

10. При наследовании от класса `BackgroundService` какой метод необходимо переопределить, чтобы хост мог автоматически запустить ваш сервис?

    Ответ. `ExecuteAsync`.

Глава 10

1. В чем разница между моделями `in-process` и `isolated` для Azure Functions?

   Ответ. Модель `in-process` требует, чтобы функция Azure загружалась вместе с другим кодом и работала в заранее определенной версии .NET LTS, например .NET 6 или .NET 8.

   Модель `isolated` позволяет функции работать в отдельном процессе, поддерживая любую версию .NET на выбор.

2. Какой атрибут используется для срабатывания функции при поступлении сообщения в очередь?

   Ответ. В методе `Run` используйте с параметром, представляющим полученное сообщение, атрибут `[QueueTrigger]`. Укажите имя очереди, например:

```
public static async Task Run(
  [QueueTrigger("checksQueue")] QueueMessage message,
  ILogger log)
{
```

3. Какой атрибут используется для поддержки очередью отправки сообщений?

   Ответ. Используйте атрибут `[Queue]` для параметра метода `Run`, представляющего очередь. Укажите ее имя, например:

```
public static async Task<IActionResult> Run(
  [HttpTrigger(AuthorizationLevel.Anonymous,
    "get", "post", Route = null)] HttpRequest req,
  [Queue("checksQueue")] ICollector<string> collector,
  ILogger log)
{
  ...
  collector.Add("This is the message."); // Отправка сообщения в очередь
  ...
```

4. Что означает показанное ниже NCRONTAB-выражение?

```
0 0 */6 * 6 6
```

   Ответ. Если речь идет о 2023 годе, то это расписание определяет выполнение задачи каждые 6 часов (`*/6`) по субботам (последнее число `6`) независимо от дня недели (последний символ `*`) в июне (предпоследнее число `6`). Событие запускается в 0 минут 0 секунд (`0 0`). Пример вывода:

```
Sat, 03 Jun 2023 00:00:00
Sat, 03 Jun 2023 06:00:00
Sat, 03 Jun 2023 12:00:00
Sat, 03 Jun 2023 18:00:00
Sat, 10 Jun 2023 00:00:00
Sat, 10 Jun 2023 06:00:00
Sat, 10 Jun 2023 12:00:00
Sat, 10 Jun 2023 18:00:00
Sat, 17 Jun 2023 00:00:00
Sat, 17 Jun 2023 06:00:00
Sat, 17 Jun 2023 12:00:00
Sat, 17 Jun 2023 18:00:00
Sat, 24 Jun 2023 00:00:00
Sat, 24 Jun 2023 06:00:00
Sat, 24 Jun 2023 12:00:00
Sat, 24 Jun 2023 18:00:00
```

5. Как зарегистрировать зависимости для использования в функции?

   Ответ. Создайте класс, наследуемый от `FunctionsStartup`, и переопределите его метод `Configure`. Добавьте атрибут сборки `[FunctionsStartup]`, укажите имя класса, зарегистрированного для запуска. В методе `Configure` добавьте сервисы через экземпляр `IfunctionsHostBuilder`.

Глава 11

1. Какие протоколы передачи используются в SignalR? А какой из них по умолчанию?

   Ответ. В SignalR рекомендуется использовать транспортный протокол WebSockets. Если он недоступен, выполняется возврат к событиям на стороне сервера, а при отсутствии поддержки используется запрос типа Long polling.

2. Каковы принципы проектирования сигнатур методов RPC?

   Ответ. При разработке сигнатур методов RPC рекомендуется определять один параметр сложного типа. Благодаря этому в будущем вы сможете добавлять к нему новые свойства без нарушения контракта между клиентом и сервером.

3. Как загрузить JavaScript-библиотеку SignalR?

   Ответ. В терминале можно воспользоваться консольным инструментом CLI Library Manager. В программе Visual Studio 2022 для этого предусмотрен графический интерфейс: щелкните на названии проекта правой кнопкой мыши и выберите в контекстном меню команду Add ▸ Client Side Libraries (Добавить ▸ Клиентская библиотека).

4. Что произойдет при отправке сообщения SignalR клиенту с несуществующим идентификатором соединения?

   Ответ. Ничего.

5. Каковы преимущества отделения сервиса SignalR от других компонентов ASP.NET Core?

   Ответ. Отделение хостинга SignalR позволяет использовать все возможности Azure SignalR, обеспечивающего глобальный охват, центр обработки данных и сеть мирового класса, масштабируемость до миллионов подключений, высокую безопасность и соответствие требованиям SLA, таких как обеспечение соответствия требованиям и высокая безопасность.

Глава 12

1. Какой транспортный протокол используется в GraphQL?

   Ответ. GraphQL работает через HTTP, а также поддерживает WebSockets и другие протоколы.

2. Какой формат передачи данных используется в GraphQL-запросах?

   Ответ. В GraphQL используется собственный формат данных, `application/graphql`.

3. Как передавать параметры в GraphQL-запросах?

   Ответ. Параметры указываются с префиксом `$` и типом данных. После этого к ним можно обращаться из очереди, например, так:

```
query getOrdersByDateAndCountry($country: String, $orderDate: String) {
  order(orderDate: $orderDate {
    orderId
    orderDate
    customer(country: $country) {
      companyName
      country
    }
  }
}
```

4. Каковы преимущества Strawberry Shake перед обычным HTTP-клиентом в рамках отправки GraphQL-запросов?

   Ответ. Инструмент Strawberry Shake генерирует строго типизированные прокси, которые упрощают вызовы GraphQL-сервисов в .NET-клиентах по сравнению с запросами вручную.

5. Как добавить новый товар в БД Northwind?

   Ответ. Нужно определить мутацию `addProduct`, включая тип входных данных, тип полезной нагрузки и класс с методом для вставки. После этого можно отправить запрос, как показано ниже:

```
mutation AddProduct {
  addProduct(
    input: {
      productName: "Tasty Burgers"
      supplierId: 1
      categoryId: 2
      quantityPerUnit: "6 per box"
      unitPrice: 40
      unitsInStock: 0
      unitsOnOrder: 0
      reorderLevel: 0
      discontinued: false
    }
  )
  {
    product {
      productId
      productName
    }
  }
}
```

Глава 13

1. Благодаря каким трем преимуществам gRPC прекрасно подходит для реализации сервисов?

   Ответ. Три ключевых преимущества gRPC, благодаря которым этот фреймворк отлично подходит для создания сервисов:

   1) бинарная сериализация Protobuf, значительно снижающая объем передаваемых данных и уменьшающая нагрузку на сеть;
   2) использование протокола HTTP/2, который повышает производительность за счет мультиплексирования потоков и сжатия заголовков;
   3) широкая поддержка различных языков программирования и платформ, что упрощает интеграцию сервисов в разнородных системах.

2. Как в gRPC определяются контракты?

   Ответ. В gRPC контракты определяются с помощью `.proto`-файлов.

3. Каким типам .NET (`int`, `double`, `DateTime`) требуются расширения для импорта?

   Ответ. Тип данных `DateTime` не поддерживается по умолчанию и требует импорта расширения.

4. Почему при вызове метода gRPC важно ограничивать срок выполнения?

   Ответ. Ограничение срока выполнения вызова gRPC — это важная мера, позволяющая контролировать максимальное время работы запроса. Это предотвращает чрезмерное потребление серверных ресурсов и помогает избежать ситуации, когда длительные или зависшие запросы негативно сказываются на производительности системы.

5. В чем заключаются преимущества транскодирования JSON для gRPC-сервиса, размещенного в ASP. NET Core?

   Ответ. gRPC предъявляет строгие требования к клиентам, и не все из них могут с ним работать нативно. В отличие от него HTTP-запросы и передача JSON-объектов поддерживаются практически всеми клиентами. Включение транскодирования JSON в gRPC-сервисе делает его доступным для большего числа пользователей, сочетая преимущества обоих подходов: удобство и универсальность JSON с высокой производительностью и надежностью gRPC.

Глава 14

1. В чем преимущество объявления строго типизированного представления Razor и как это сделать?

   Ответ. Преимущество объявления строго типизированного представления Razor заключается в поддержке IntelliSense, что помогает при написании разметки и кода, а также позволяет компилятору обнаруживать ошибки на этапе проектирования. Если тип модели не указан, то представление Razor будет использовать тип `dynamic` и IntelliSense не сможет показать использование класса

`Model`, что приведет к ошибкам только на этапе выполнения. Чтобы указать тип модели, добавьте директиву `@model` с желаемым типом в верхнюю часть представления Razor. Например:

```
@model Customer
...
<h1>@Model.CompanyName</h1>
...
```

2. Как использовать тег-хелперы в представлении?

   Ответ. Для использования тег-хелперов в представлении Razor или в файле `Views/_ViewImports.cshtml` добавьте следующее выражение:

   ```
   @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
   ```

3. В чем преимущества и недостатки HTML-хелперов по сравнению с тег-хелперами?

   Ответ. Тег-хелперы обычно проще в использовании, и их следует применять в большинстве случаев. Однако бывают ситуации, когда их нельзя использовать, например в компонентах Razor. В таких случаях лучше использовать HTML-хелперы, которые предоставляют больше возможностей, но программируются сложнее.

4. Как браузер запрашивает предпочтительный язык для локализации?

   Ответ. Браузер может указать предпочитаемую культуру несколькими способами:

   - добавив параметр строки запроса (например, `?culture=en-US&ui-culture=en-US`);
   - отправив cookie-файл (например, `c=en-US|uic=en-US`);
   - установив HTTP-заголовок (например, `Accept-Language: en-US,en;q=0.9, fr-FR;q=0.8,fr;q=0.7,en-GB;q=0.6`).

5. Как локализовать текст в представлении?

   Ответ. Создайте файлы ресурсов (`*.resx`) для каждого языка, в которых будут храниться текстовые значения для локализации. Внедрите в представление Razor службу `IViewLocalizer` для доступа к локализованным строкам. В файле `Program.cs` вызовите метод `AddLocalization` и укажите папку с `.resx`-файлами. Затем вызовите метод `AddViewLocalization` после метода `AddControllersWithViews`. При построении HTTP-конвейера вызовите метод `UseRequestLocalization`, указав поддерживаемые культурные коды ISO.

6. Какой префикс у атрибутов, распознаваемых тег-хелперами?

   Ответ. Префикс — `asp-`. Например, `asp-action` и `asp-controller` в элементе `a`:

   ```
   <a asp-action="Privacy" asp-controller="Home">View our privacy policy.</a>
   ```

7. Как передать сложный объект в качестве параметра строки запроса?

   Ответ. Создайте метод, который возвращает словарь с ключами и значениями в виде строк. Затем передайте результат метода в атрибут `asp-all-route-data` элемента `a`. Пример разметки:

```
<a asp-controller="Home" asp-action="Shipper"
   asp-all-route-data="await GetShipperData()"
   class="btn btn-outline-primary">Shipper</a>
```

8. Как управлять сроком кэширования с помощью элемента `cache`?

   Ответ. Элемент `cache` поддерживает следующие атрибуты для настройки сроков хранения кэша:

   - `expires-after`: задает время в формате `TimeSpan`, через которое содержимое кэша будет удалено. По умолчанию это 20 минут (`00:20:00`);
   - `expires-on`: задает конечную дату хранения кэша в формате `DateTimeOffset`. Значения по умолчанию нет;
   - `expires-sliding`: задает время в формате `TimeSpan`, через которое кэш будет удален, если в течение этого времени к нему не обращались. Значения по умолчанию нет.

9. Для каких целей используется элемент `environment`?

   Ответ. Элемент `environment` позволяет отображать содержимое только в тех средах, которые указаны в его атрибутах. Это полезно для того, чтобы отображать разное содержимое в зависимости от среды, например показывать только данные для разработки или рабочего сайта.

10. Как выполняется кэширование с помощью тег-хелперов?

    Ответ. Если атрибуту `asp-append-version` присвоено значение `true` в элементах `link`, `img` или `script`, то используется соответствующий тег-хелпер. В результате будет вызвана кэшированная версия файла, на который ведет ссылка. При этом автоматически формируется строка запроса, генерируемая на основе SHA256-хеша исходного файла, на который указывает ссылка.

Глава 15

1. В чем заключается преимущество новой модели размещения Blazor Full Stack в .NET 8 по сравнению с устаревшими моделями наподобие Blazor Server?

   Ответ. Устаревшие модели, например Blazor Server, ограничивают возможности работы с компонентами. В отличие от них новая модель Blazor Full Stack предоставляет большую гибкость и дополнительные возможности. Компоненты могут работать на сервере, генерировать статическую разметку, а также переключаться на один из следующих режимов: потоковый рендеринг,

интерактивный сервер с постоянным обновлением через SignalR или интерактивный клиент с использованием WebAssembly. Модель Blazor WebAssembly по-прежнему актуальна для создания автономных приложений, размещенных на статических сайтах.

2. Поддерживает ли Blazor WebAssembly все возможности новейших API .NET?

   Ответ. Нет. Из-за ограничений, связанных с работой внутри браузера, Blazor WebAssembly не поддерживает все возможности .NET. Использование неподдерживаемых функций вызовет исключение `PlatformNotSupportedException`. Анализатор совместимости браузеров предупреждает о возможных проблемах еще на этапе разработки.

3. Какое расширение файлов Blazor-компонентов?

   Ответ. `.razor`.

4. Как для всех компонентов страницы Blazor применить макет по умолчанию?

   Ответ. Для установки макета по умолчанию в файле `App.razor` необходимо задать свойство `DefaultLayout` элемента `RouteView`:

```
<Router AppAssembly="@typeof(App).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    ...
  </Found>
  ...
</Router>
```

5. Как зарегистрировать маршрут для компонента страницы Blazor?

   Ответ. В начале `.razor`-файла компонента страницы необходимо добавить директиву `@page` с указанием маршрута:

```
@page "my-route"
```

6. Когда свойству `Match` компонента `NavLink` необходимо присваивать значение `NavLinkMatch.All`?

   Ответ. Это необходимо, когда несколько экземпляров компонента `NavLink` могут частично совпадать с путями. Например, маршрут `/customers` будет соответствовать как `/customers`, так и `/customers/USA`. В таком случае в компоненте `NavLink` для маршрута `/customers` нужно установить значение `NavLinkMatch.All`.

7. В файле `_Imports.razor` вы импортировали пользовательское пространство имен, но, когда пытаетесь использовать класс из него в файле отделенным кодом (code-behind) для компонента Blazor, класс не обнаруживается. Почему? Как решить эту проблему?

   Ответ. Файл `_Imports.razor` применяется только к `.razor`-файлам. Для `.cs`-файлов с отделенным кодом пространство имен необходимо импортировать

отдельно. Чтобы решить эту проблему, можно использовать глобальные директивы `using` для неявного импорта пространства имен в файлах с кодом компонента.

8. Что нужно сделать со свойством в классе компонента, чтобы оно автоматически заполнялось из строки запроса?

   Ответ. Нужно пометить свойство атрибутами `[Parameter]` и `[SupplyParameterFromQuery]`, как показано ниже:

```
[Parameter]
[SupplyParameterFromQuery(Name = "country")]
public string? Country { get; set; }
```

9. Что такое `QuickGrid`?

   Ответ. `QuickGrid` — это компонент с открытым исходным кодом, предназначенный для отображения сетки данных в Blazor и разработанный компанией Microsoft.

10. Как Blazor-компоненту получить доступ к таким функциям браузера, как локальное хранилище?

    Ответ. Для доступа к функциям браузера, таким как локальное хранилище, компонент Blazor использует механизм JavaScript interop через интерфейс `IJSRuntime`. С помощью метода `InvokeAsync<IJSObjectReference>` можно динамически загружать JavaScript-модуль, который затем взаимодействует с функциями браузера.

Глава 16

1. Опишите четыре категории компонентов пользовательского интерфейса .NET MAUI.

   Ответ. Компоненты пользовательского интерфейса .NET MAUI делятся на четыре категории:

   - *страницы* — представляют экраны мобильных приложений;
   - *шаблоны* — задают структуру, объединяющую несколько компонентов;
   - *представления* — отдельные элементы пользовательского интерфейса;
   - *ячейки* — элементы списка или таблицы.

2. В чем преимущество компонента `Shell` и какие виды пользовательского интерфейса он реализует?

   Ответ. Компонент `Shell` упрощает разработку приложений, предоставляя стандартные механизмы навигации и поиска. Он позволяет организовать пользовательский интерфейс с вкладками или всплывающими окнами для навигации.

3. Как разрешить пользователю выполнять действие с ячейкой в представлении списка?

   Ответ. Можно добавить контекстные действия — пункты меню, при выборе которых вызывается определенное событие:

```
<TextCell Text="{Binding CompanyName}"
          Detail="{Binding Location}"
          TextColor="{DynamicResource PrimaryTextColor}"
          DetailColor="{DynamicResource PrimaryTextColor}" >
  <TextCell.ContextActions>
    <MenuItem Clicked="Customer_Phoned" Text="Phone" />
    <MenuItem Clicked="Customer_Deleted" Text="Delete"
              IsDestructive="True" />
  </TextCell.ContextActions>
</TextCell>
```

4. В каких случаях используется компонент `Entry`, а в каких — `Editor`?

   Ответ. Компонент `Entry` применяется для ввода одной строки текста, а `Editor` — нескольких строк текста.

5. Какое влияние оказывает установка свойства `IsDestructive = true` для пункта меню в контекстных действиях ячейки?

   Ответ. Пункт меню окрашивается в красный цвет, предупреждая пользователя.

6. Почему оболочка с одной страницей содержимого не отображает навигацию?

   Ответ. Навигация отсутствует, поскольку переходить некуда, — требуется как минимум две страницы.

7. Чем отличаются свойства `Margin` и `Padding` у элемента, например `Button`?

   Ответ. Свойство `Margin` определяет внешний отступ (за границами элемента), а `Padding` — внутренний отступ (внутри границ элемента).

8. Как прикрепить обработчики событий к объекту через XAML?

   Ответ. Указать в XAML-атрибуте имя события и метод в коде, например:

```
<Button Clicked="SaveButton_Clicked">
```

9. Для чего используются стили XAML?

   Ответ. Стили XAML позволяют задать одно или несколько свойств.

10. Где можно определить ресурсы?

    Ответ. Ресурсы можно определить в любом элементе в зависимости от области видимости:

    - в элементе `Application.Resources` — для всего приложения;
    - в элементе `Page.Resources` — для конкретной страницы;
    - в элементе `Button.Resources` — для отдельного элемента.

Алексей Сошин

KOTLIN. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ И ЛУЧШИЕ ПРАКТИКИ

3-е издание

Руководство позволит разработчикам, интересующимся паттернами проектирования в Kotlin, применить свои знания на практике. В книге рассматриваются классические и современные паттерны проектирования, демонстрируются практические подходы к их реализации и соответствующие методологии.

Третье издание содержит актуальные обновления Kotlin, начиная с версии 1.6, и предлагает углубленное обсуждение таких тем, как структурированная конкурентность и контекстные приемники. Книга начинается с описания базового синтаксиса Kotlin и важности паттернов проектирования, в том числе классических порождающих, структурных и поведенческих паттернов. Затем она переходит к представлению средств функционального программирования и паттернов реактивной и конкурентной обработки, включая корутины и структурированную конкурентность. Знакомясь с этими продвинутыми концепциями, вы будете совершенствовать навыки программирования на Kotlin. В книге также рассматриваются последние архитектурные тенденции, особое внимание уделяется паттернам проектирования микросервисов, которые помогут вам выбрать ту или иную архитектуру.

Вы хорошо разберетесь в этих передовых концепциях и сумеете применить их в своих проектах.

SPRINT book

Натан Лифтинг, Брайан ван Бакел

ZABBIX 7: МОНИТОРИНГ ИТ-ИНФРАСТРУКТУРЫ

Обновленное третье издание «Zabbix 7: мониторинг ИТ-инфраструктуры» познакомит вас со всеми возможностями Zabbix. Эта книга, включающая различные примеры и описывающая обновленные функциональные возможности Zabbix, охватывает все, что нужно знать для настройки высокой доступности серверов, расширенной аутентификации пользователей с помощью LDAP, реализации низкоуровневого обнаружения хостов и создания продвинутых триггеров.

В этом практическом руководстве вы найдете простые и понятные рецепты применения Zabbix для мониторинга устройств и приложений в сети. Сначала вы узнаете, как установить Zabbix, и познакомитесь с его наиболее важными функциями, что поможет вам принимать правильные решения при создании масштабируемых и легкоуправляемых окружений. Прорабатывая каждый пример, вы научитесь создавать элементы данных и триггеры для различных типов мониторинга, разрабатывать шаблоны и использовать прокси-серверы Zabbix. Вы также научитесь использовать Zabbix API для настройки сервера и базы данных Zabbix и управления ими. Наконец, вы найдете быстрые решения распространенных и не очень проблем, возникающих в процессе мониторинга с применением Zabbix.

Прочитав всю книгу, вы получите полное представление о том, как использовать Zabbix для мониторинга, и сможете построить надежную систему Zabbix, используя ее ключевые функции.

SPRINT book

Марк Прайс

.NET 8: ИНСТРУМЕНТЫ И НАВЫКИ. ЛУЧШИЕ ПРАКТИКИ И ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ, ОТЛАДКИ И ТЕСТИРОВАНИЯ

Раскройте богатство возможностей разработки .NET-приложений с новой книгой Марка Прайса! Здесь вы узнаете, как управлять исходным кодом и проектами с помощью Git, обеспечивая при этом контроль версий. Откроете для себя передовые методы отладки и стратегии выявления и устранения проблем, а также увидите, как правильно документировать код, API и сервисы, чтобы повысить удобство обслуживания проекта. Погрузитесь в мир криптографии и освоите приемы обеспечения конфиденциальности и целостности на протяжении всего цикла разработки.

Вы сможете расширить свои навыки, познакомившись с такими продвинутыми темами, как создание интеллектуальных приложений с поддержкой пользовательских чат-сервисов на основе LLM, внедрение зависимостей, оптимизация производительности с помощью тестирования и запуск сервисов в контейнерах Docker. В книге описаны возможности разработки облачных сервисов с применением .NET Aspire и преимущества современных облачных платформ. А знакомство с особенностями передовых архитектур программного обеспечения позволит вам создавать надежные и масштабируемые приложения, удобные в обслуживании.

SPRINT book

Дэнис Ротман

RAG И ГЕНЕРАТИВНЫЙ ИИ. СОЗДАЕМ СОБСТВЕННЫЕ RAG-ПАЙПЛАЙНЫ С ПОМОЩЬЮ LLAMAINDEX, DEEP LAKE И PINECON

В книге описываются приемы создания эффективных больших языковых моделей, систем компьютерного зрения и генеративного ИИ, показывающих высокую производительность при относительно невысоких затратах. В ней приводится подробное исследование технологии RAG, а также подходов к проектированию мультимодальных пайплайнов ИИ и управлению ими. Связывая вывод с исходными документами, RAG повышает точность и контекстную релевантность результатов, предлагая динамический подход к управлению большими объемами информации.

Узнайте, как построить инфраструктуру RAG, попутно разобравшись с векторными хранилищами, фрагментацией, индексацией и ранжированием. Познакомьтесь с методами оптимизации производительности и приемами более глубокого изучения данных, включая использование адаптивного RAG и обратной связи от человека для уточнения поиска, тонкую настройку RAG, реализацию динамических RAG для поддержки принятия решений в реальном времени и визуализацию сложных данных с помощью графов знаний. Вы также увидите, как на практике объединить такие фреймворки, как LlamaIndex и Deep Lake, векторные базы данных наподобие Pinecone и Chroma и модели, предлагаемые компаниями Hugging Face и OpenAI. Приобретите навыки внедрения интеллектуальных решений, что повысит вашу конкурентоспособность в различных областях: от продакшна до обслуживания клиентов в любом проекте.

SPRINT book