
Федор Сазанавец
C# 11 и .NET 7.0
Создаем кросс-платформенные
приложения на базе .NET Core

Implementing C# 11 and .NET 7.0

Learn how to build cross-platform apps with .NET Core

Fiodar Sazanavets



www.bpbonline.com

Фёдор Сазанавец

C# 11 и .NET 7.0

Санкт-Петербург
«БХВ-Петербург»
2024

УДК 004.43
ББК 32.973.26-018.1
С14

Сазанавец Ф.

С14 С# 11 и .NET 7.0: Пер. с англ. — СПб.: БХВ-Петербург, 2024. — 288 с.: ил.

ISBN 978-5-9775-1966-3

Книга о разработке нативных, кросс-платформенных и веб-приложений на платформе .NET с использованием языка C#. Подробно рассмотрена экосистема платформы .NET, средства связи в распределённых системах (gRPC и SignalR), инструменты современной веб-разработки — Blazor, библиотека машинного обучения ML.NET, фреймворк для веб-разработки ASP.NET Core, инструментарий Entity Framework Core 7. Объяснены возможности подготовки микросервисной архитектуры, а также развертывание контейнеров и управление ими с применением Docker Swarm и Kubernetes.

Для разработчиков

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Олег Сивченко*
Зав. редакцией	*Людмила Гауль*
Перевод с английского	*Дмитрия Мешкова*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Зои Канторович*

Copyright 2023 BPB Publications, India. All rights reserved.
First published in the English language under the title ***Implementing C# 11 and .NET 7.0: Learn how to build cross-platform apps with .NET Core***, ISBN 9789355513281 by BPB Publications India (sales@bpbonline.com).
Russian translation rights arranged with BPB Publications, India.

© 2023 BPB Publications, Индия. Все права защищены. Впервые опубликовано на английском языке под названием ***Implementing C# 11 and .NET 7.0: Learn how to build cross-platform apps with .NET Core***, ISBN 9789355513281 издательством BPB Publications India (sales@bpbonline.com).
Права на перевод на русский язык предоставлены издательством BPB Publications, Индия.

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

ISBN 978-93-55513-281 (англ.)
ISBN 978-5-9775-1966-3 (рус.)

© BPB Publications, India, 2023
© Перевод на русский язык, оформление. ООО "БХВ-Петербург", ООО "БХВ", 2024

Оглавление

Об авторе **15**

О рецензентах **16**

Благодарности **17**

Предисловие **18**
Комплект исходного кода и цветные иллюстрации 22

Глава 1. Знакомство со структурой приложений .NET 7 ..... **23**
Введение 23
Структура 24
Цели 24
Настройка среды разработки 24
Компьютеры, подходящие для разработки ПО 24
.NET 7 SDK 24
Установка редактора кода или среды IDE 25
Установка подходящего редактора кода 26
Установка подходящей IDE 28
Microsoft Visual Studio 2022 28
Microsoft Visual Studio 2022 for Mac 29
JetBrains Rider 30
Создание приложения .NET 7 31
Создание приложения с помощью CLI 32
Создание приложения с помощью графического интерфейса IDE 33
Обзор структуры проекта .NET 7 35
Добавление объекта *struct* 35
Добавление интерфейса и класса 36
Изменение точки входа в приложение 39
Основы C# 11 и встроенные типы данных 41
Встроенные типы данных 41
Поток управления 43
Пользовательские типы в C# 44
Модификаторы доступа 44
Заключение 45

Что следует помнить 46
Контрольные вопросы 46
Ответы 47
Основные термины 47

Глава 2. Обзор возможностей C# 11 48
Введение 48
Структура 48
Цели 48
Предварительные условия 49
Автоматическая инициализация структур 49
Обобщенные атрибуты 50
Пример обобщенного атрибута 51
Сопоставление последовательности с образцом 53
Демонстрация сопоставления последовательности с образцом 54
Сопоставление последовательности с образцом типа *char span* 54
Новые операции со строками 56
Заключение 59
Что следует помнить 60
Контрольные вопросы 60
Ответы 61
Основные термины 61

Глава 3. Что нового в .NET 7? 62
Введение 62
Структура 62
Цели 62
Предварительные условия 63
Улучшения в SDK и инструментах сборки 63
Улучшения инструментов командной строки 63
NativeAOT и функция обрезки библиотек 64
Центральный диспетчер пакетов 65
Обновления библиотек System и Microsoft 66
Поддержка микросекунд и наносекунд 66
Новые возможности для работы с JSON 68
Свойство *MaxDepth* класса *JsonWriterOptions* 69
Конфигурация *JsonSerializerOptions* по умолчанию 70
Вызовы *HTTP PATCH* для JSON 70
Полиморфизм JSON 71
Тестирование возможностей JSON 73
Новые возможности класса *Stream* 73
Улучшения RegEx 75
Улучшения в криптографии 76
Новый TAR API 78
Улучшения наблюдаемости 80
Новые способы мониторинга действий 80
Мониторинг остановленных действий 80
Событие изменения текущего действия 81
Перечисление свойств действия 81
Показатель *UpDownCounter* 83

Критические изменения в .NET 7 ... 83
Допустимость неопределенных значений в пространстве имен Microsoft.Extensions ... 83
Конечные точки: устаревшие и не допускающие неопределенных значений ... 84
Ограничение *PatternContext* ... 84
Многоуровневый поиск в Windows отключен ... 84
Сериализация пользовательских типов в *MSBuild* ... 84
Заключение ... 85
Что следует помнить ... 85
Контрольные вопросы ... 86
Ответы ... 86
Основные термины ... 87

Глава 4. MAUI и кросс-платформенные нативные приложения ... 88
Введение ... 88
Структура ... 88
Цели ... 89
Предварительные условия ... 89
Общие сведения о MAUI ... 89
Включение среды разработки MAUI ... 90
Создание простых MAUI-приложений ... 91
Справочные данные по MAUI XAML ... 94
Работа с Blazor на платформе .NET MAUI ... 97
Архитектурные паттерны MAUI ... 99
Model-View-ViewModel ... 99
Паттерны, поддерживаемые MAUI через сторонние платформы ... 101
ReactiveUI ... 101
Model-View-Update ... 101
Использование MAUI для создания настольных приложений ... 102
Подготовка среды разработки настольных приложений ... 102
Запуск настольного приложения в режиме отладки ... 102
Публикация настольного приложения ... 103
Использование MAUI для создания мобильных приложений ... 104
Подготовка среды разработки мобильных приложений ... 105
Запуск мобильного приложения на эмуляторе ... 106
Публикация мобильного приложения ... 106
Ограничения разработки для Mac OS и iOS ... 107
Дополнительные инструменты, необходимые для публикации приложений для iOS ... 107
Менее строгие требования к приложениям для Mac OS ... 108
Заключение ... 108
Что следует помнить ... 109
Контрольные вопросы ... 109
Ответы ... 110
Основные термины ... 110

Глава 5. Доступ к базам данных с помощью Entity Framework 7 ... 111
Введение ... 111
Структура ... 111
Цели ... 112
Предварительные условия ... 112

Знакомство с основами реляционных БД ... 112
Обзор систем управления реляционными БД и SQL ... 112
Таблицы, отношения и нормализация ... 113
Введение первичных ключей ... 115
Нормализация и отношения внешних ключей ... 115
Введение в Entity Framework 7 ... 117
Подход "сначала код" в EF7 ... 119
Добавление кода Entity Framework ... 119
Добавление объектов-сущностей ... 120
Добавление контекста базы данных ... 122
Добавление скрипта создания базы данных ... 124
Создание базы данных путем запуска приложения ... 128
Подход "сначала база данных" в EF7 ... 129
Создание моделей EF7 из существующей БД ... 129
Просмотр автоматически сгенерированного кода ... 130
Новейшие функции EF7 ... 131
Подход "сначала база данных" и управление с помощью шаблонов T4 ... 131
Защищенный ключ ... 131
Отображение "одна таблица на конкретный тип" (TPC) ... 133
Перехватчики ... 136
Заключение ... 137
Что следует помнить ... 137
Контрольные вопросы ... 138
Ответы ... 139
Основные термины ... 139

Глава 6. Типы веб-приложений в .NET ... 140
Введение ... 140
Структура ... 140
Цели ... 141
Предварительные условия ... 141
Основы ASP.NET Core ... 141
Структура базового приложения ASP.NET Core ... 141
Web API в ASP.NET Core ... 144
Web API с контроллерами ... 144
Конечные точки Minimal API ... 149
Добавление метаданных Open API ... 150
Улучшение передачи параметров в Minimal API ... 150
Minimal API и типизированные результаты ... 151
Выгрузка файлов в Minimal API ... 153
Новое в промежуточном ПО для обработки запросов ... 153
MVC в ASP.NET Core ... 155
Razor Pages в ASP.NET Core ... 167
Заключение ... 169
Что следует помнить ... 170
Контрольные вопросы ... 170
Ответы ... 171
Основные термины ... 171

Глава 7. Blazor и WebAssembly в .NET 172
Введение 172
Структура 173
Цели 173
Предварительные условия 173
Общие сведения о Blazor 173
Примеры Razor-компонентов 174
@page 175
@onclick 175
@code 175
Ключевые слова Razor в Razor-компонентах 175
@using 176
@implements 176
@inherits 176
@inject 176
@layout 176
@namespace 176
@preservewhitespace 177
@attributes 177
@bind 177
@ref 177
@typeparam 178
Обзор Blazor WebAssembly 178
Отделение кода программной части 179
Взаимодействие с JavaScript 181
Передача параметров в Razor-компоненты 182
Предварительная компиляция 183
Пустой шаблон Blazor WebAssembly 184
Хостинг приложений Blazor WebAssembly в ASP.NET Core 185
Добавление хостируемого приложения Blazor WebAssembly к существующему приложению ASP.NET Core 185
Проверка форм в Blazor 186
NavigationManager и передача состояния между страницами 188
Настройка Blazor Server 189
Пользовательские элементы в Blazor 190
Жизненный цикл Razor-компонента 191
Пустой шаблон Blazor Server 192
Заключение 193
Что следует помнить 193
Контрольные вопросы 194
Ответы 194
Основные термины 195

Глава 8. SignalR и двусторонняя связь 196
Введение 196
Структура 197
Цели 197
Предварительные условия 197

Общие сведения о библиотеке SignalR....197
WebSocket....198
Server-Sent Events....198
Длительный опрос....199
Создание хаба SignalR на сервере....199
Строго типизированный хаб....202
Внедрение зависимостей в хаб SignalR....203
JSON и MessagePack....204
JavaScript-клиент для SignalR....204
Добавление HTML-разметки для клиента SignalR....205
Применение функций SignalR в JavaScript....208
.NET-клиент для SignalR....211
Заключение....214
Что следует помнить....214
Контрольные вопросы....214
Ответы....215
Основные термины....215

Глава 9. gRPC в ASP.NET Core....216
Введение....216
Структура....216
Цели....217
Предварительные условия....217
Общие сведения о технологии gRPC....217
Protobuf как основной протокол сериализации сообщений....218
Настройка сервера gRPC....218
Структура проекта ASP.NET Core gRPC....219
Типы вызовов и типы данных gRPC....221
JSON-перекодирование в gRPC....224
Настройка клиента gRPC....227
Использование фабрики клиентов gRPC и внедрение зависимостей....230
Обзор типов данных gRPC....232
Перечисления Protobuf....233
Использование коллекций с помощью ключевого слова *repeated*....234
Функциональность словаря в Protobuf....234
Использование ключевого слова *oneof* в Protobuf....234
Общеизвестные типы данных....235
Заключение....236
Что следует помнить....237
Контрольные вопросы....237
Ответы....238
Основные термины....238

Глава 10. Машинное обучение с помощью ML.NET....239
Введение....239
Структура....240
Цели....240
Предварительные условия....240
Основы ML.NET....240
Типы машинного обучения....242

Обучение с учителем 242
Обучение без учителя 242
Обучение с подкреплением 242
Начало работы с ML.NET 243
Установка инструментов ML.NET 243
Использование ML.NET для создания первой модели МО 244
Выбор задачи для МО 246
Бинарная классификация 246
Многоклассовая классификация 248
Регрессия 249
Рекомендации 250
Прогнозирование 251
Классификация изображений 253
Кластеризация 254
Обнаружение отклонений 254
Ранжирование 255
Обучение и оценка модели 255
Показатели бинарной классификации 256
Показатели многоклассовой классификации 256
Классификация изображений 256
Прогнозирование 257
Регрессия и рекомендации 257
Показатели кластеризации 257
Показатели обнаружения отклонений 258
Показатели ранжирования 258
Использование малокодового построителя моделей 258
Заключение 261
Что следует помнить 262
Контрольные вопросы 262
Ответы 263
Основные термины 263

Глава 11. Микросервисы и контейнеризация в .NET 7 265
Введение 265
Структура 266
Цели 266
Предварительные условия 266
Контейнеры Docker 266
Контейнеры и образы контейнеров 267
Базовые образы и уровни 267
Изоляция сети и сопоставление портов 268
Изоляция файловой системы и монтирование каталогов хоста 268
Установка Docker в Linux 269
Установка Docker на Mac 269
Установка Docker в Windows 269
Базовый образ Docker для .NET 7 270
Создание приложения с поддержкой Docker 271
Добавление поддержки Docker в существующее приложение 272
Структура файла Dockerfile 273
Создание и запуск контейнера Docker 275

Оркестрация приложений с помощью Docker Swarm 275
Базовая оркестрация помощью Docker Compose 276
Запуск Docker Swarm 278
Оркестрация приложений с помощью Kubernetes 279
Установка Kubernetes в Linux 279
Установка Kubernetes на Mac 280
Добавление сервисов в кластер Kubernetes 281
Заключение 282
Что следует помнить 282
Контрольные вопросы 282
Ответы 283
Основные термины 283

Предметный указатель 285

Посвящается

Моей матери, ***Лилии Сазанавец****,*

и памяти моего отца, ***Дмитрия Сазанавеца****,*
которого, к сожалению, больше нет с нами.

Моей жене, ***Ольге Сазанавец***

Об авторе

Фёдор Сазанавец является участником программы Microsoft MVP (Most Valuable Professionals — Самые ценные специалисты) и старшим инженером-программистом с более чем десятилетним профессиональным опытом. Основные области его специализации — .NET и стек технологий Microsoft. Он с энтузиазмом создает качественное программное обеспечение, полностью отвечающее потребностям бизнеса. Он с удовольствием обучает начинающих программистов и делится своими знаниями с сообществом разработчиков.

На протяжении своей карьеры Фёдор создавал программное обеспечение различных типов и уровней сложности в различных отраслях. Сюда входит система управления информацией о пассажирах железных дорог, распределенные интеллектуальные кластеры устройств Интернета вещей (IoT), системы электронной коммерции, системы обработки финансовых транзакций и многое другое.

О рецензентах

- **Ваче Чек** — в настоящее время старший инженер-программист со специализацией в области серверных и облачных вычислений. Наука и технологии были для него постоянным предметом увлечения начиная с 13 лет, когда он в качестве хобби начал писать программы для своего Commodore 64. Со временем его страсть к программированию росла, и в конце концов он сделал это своей карьерой. Несмотря на то что он самостоятельно изучил предмет, он обнаружил, что самый эффективный способ улучшить свои навыки — делиться своими знаниями с другими.
- **Кратика Джайн** — старший разработчик программного обеспечения, она полна энергии и стремится внести свой вклад в коллективный успех посредством усердной работы, внимания к деталям и отличных организационных навыков в руководстве и управлении несколькими проектами. Кратика обеспечивает высокое качество кода, безопасность, следование паттернам проектирования и подготовку тестовых примеров, сочетая все это с непрерывными процессами интеграционной сборки. Она участвовала в управлении проектами по методике Agile и разрабатывала серверные приложения с использованием Asp.NET, MVC, .NET CORE, Entity Framework, SQL-сервера, а также обладает глубокими знаниями паттернов проектирования и методик разработки программного обеспечения.

Благодарности

Я хочу поблагодарить всех тех людей, которые поддерживали и наставляли меня на протяжении моей карьеры. Среди них Дикайос Пападогконас, Вече Чек, Ян Тернер, Пол Экклстон, Фрэнк Лоуренс и другие профессионалы, с которыми я работал или сотрудничал.

Предисловие

Приветствую читателей книги о C# 11 и .NET 7! Если вы являетесь разработчиком программного обеспечения, то, вероятно, слышали о C# и .NET и, возможно, применяли эти инструменты для создания настольных, мобильных или веб-приложений. C# — это современный объектно-ориентированный язык программирования, разработанный Microsoft, а .NET — мощная и гибкая программная среда для создания приложений для Windows, Linux, macOS и других платформ.

Эта книга предназначена для разработчиков, желающих изучить новейшие функции и улучшения в C# 11 и .NET 7. Независимо от того, являетесь ли вы новичком или опытным программистом, эта книга даст вам знания и навыки, необходимые для использования последних разработок в C# и .NET.

Из этой книги вы узнаете о новых возможностях языка C# 11, таких как необрабатываемые строковые литералы, улучшенная обработка дат и типовые математические вычисления. Вы также познакомитесь с новыми API-интерфейсами и усовершенствованиями в .NET 7, такими как более понятный интерфейс командной строки, новые функции базовых библиотек и шаблоны проектов.

Более того, эта книга поможет вам в разработке практических приложений с использованием C# 11 и .NET 7. Вы научитесь создавать веб-приложения с помощью ASP.NET Core, мобильные приложения при помощи .NET MAUI, а также компилировать внутрибраузерный код с помощью Blazor. Эти технологии не являются новыми, однако в последнем обновлении .NET все они были значительно усовершенствованы, и в данной книге эти улучшения будут продемонстрированы.

Из этой книги вы также узнаете, как работать с некоторыми более продвинутыми инструментами .NET. Вы научитесь создавать и запускать модели искусственного интеллекта с помощью ML.NET. Вы также узнаете, как создавать распределенные приложения, применяя новейшие возможности контейнеризации в .NET.

Я надеюсь, что вы получите такое же удовольствие от чтения этой книги, как и я от ее написания. Желаю вам успешного программирования!

В *главе 1 "Знакомство со структурой приложений .NET 7"* читатель познакомится с .NET 7 и получит полный набор указаний, чтобы начать работу, даже если вы никогда раньше не встречались с .NET.

Сначала мы настроим среду разработки. Поскольку .NET-приложения можно создавать как в Windows, так и в Mac или Linux, вы узнаете, какие интегрированные среды разработки (IDE) и редакторы кода можно установить на выбранную вами операционную систему.

Затем мы создадим простое .NET-приложение на основе шаблона консольного приложения. Далее мы рассмотрим структуру проекта .NET. А после этого напишем код, который поможет вам познакомиться с основным синтаксисом C# и его встроенными типами данных.

В *главе 2 "Обзор возможностей C# 11"* продемонстрировано множество новых интересных возможностей, которые были добавлены в C# 11, чтобы облегчить жизнь разработчиков и сделать процесс написания программ более эффективным.

Сначала мы рассмотрим функцию автоматической инициализации структур, которая позволяет устанавливать для свойств объектов на основе структур значения по умолчанию, соответствующие их типам данных. Это поможет избежать случайной выдачи исключений. Далее мы рассмотрим обобщенные атрибуты. Данная возможность позволяет разработчикам при определении атрибутов использовать обобщенные типы языка C# и облегчает работу с аннотациями в коде.

После этого мы поговорим о сопоставлении последовательностей с образцом. Такая возможность дает разработчикам большую гибкость при сравнении наборов данных. Затем мы перейдем к рассмотрению новых возможностей языка, связанных со строками. К ним относятся новые необрабатываемые строковые литералы и многострочная строковая интерполяция. В завершение главы мы рассмотрим необходимые объектные члены и статические члены интерфейсов.

В *главе 3 "Что нового в .NET 7?"* основное внимание уделено новым возможностям, которые были добавлены в саму платформу .NET, состоящую из SDK, средств сборки и базовых библиотек. Для начала мы рассмотрим усовершенствования SDK и инструментов сборки. Среди новых возможностей в этих областях — улучшение интерфейса командной строки, оптимизация компилятора и т. д.

Затем мы поговорим о различных улучшениях базовых библиотек из пространств имен Microsoft и System. Среди новых возможностей, добавленных в эти библиотеки, — улучшение наблюдаемости, новые возможности JSON, улучшения RegEx, возможность использования архиватора TAR и ряд других улучшений. Наконец, мы рассмотрим выводимые из употребления функции.NET 7 и критические изменения, которые были внесены в платформу.

В названии *главы 4 "MAUI и кросс-платформенные нативные приложения"* MAUI расшифровывается как Multi-platform App UI (многоплатформенный пользовательский интерфейс программирования приложений). Это фреймворк, позволяющий разработчикам создавать нативные приложения, которые могут быть скомпилированы для работы под управлением Windows, Mac OS, iOS и Android. Замысел этого фреймворка заключается в том, чтобы на основе одной и той же кодовой базы

можно было собрать исполняемый файл для любой платформы. Причем это касается как настольных компьютеров, так и мобильных устройств.

Из этой главы вы узнаете, как с помощью MAUI создавать нативные приложения любого типа, поддерживаемого этим фреймворком. Вы узнаете, как организовать свою кодовую базу таким образом, чтобы впоследствии иметь возможность компилировать свой код в исполняемый файл любого типа. Некоторые платформы, поддерживаемые MAUI, имеют определенные ограничения, касающиеся допустимых и недопустимых действий. И в этой главе вы познакомитесь с этими ограничениями.

В *главе 5 "Доступ к базам данных с помощью Entity Framework 7"* мы знакомимся со средствами объектно-реляционного отображения. Серверным компонентам веб-приложений необходимо часто обращаться к некоторой базе данных. Обычно это делается с помощью инструмента объектно-реляционного отображения (ORM), который абстрагирует доступ к базе данных и позволяет манипулировать данными непосредственно в коде. ASP.NET Core поставляется со своим собственным ORM, который известен под названием Entity Framework Core. В этой главе мы рассмотрим последнюю версию этого ORM — Entity Framework 7.

Сначала мы познакомимся с основами реляционных баз данных, для работы с которыми было разработано решение Entity Framework 7. Затем вы узнаете, как применять сам ORM. Существуют два основных способа использования ORM в приложении ASP.NET Core. И в этой главе мы рассмотрим оба подхода — "сначала код" и "сначала база данных".

В *главе 6 "Типы веб-приложений в .NET"* читатель знакомится с ASP.NET Core — основным фреймворком .NET, предназначенным для создания веб-приложений. Мы также рассмотрим различные типы веб-приложений, которые поддерживает ASP.NET Core.

Мы начнем с основ ASP.NET Core, которые одинаковы для всех типов приложений ASP.NET Core. Далее мы рассмотрим Web API — особый тип приложения, предоставляющего интерфейс REST API для входящих HTTP-запросов, но не имеющего веб-страниц. Затем мы перейдем к приложениям типа "модель-представление-контроллер" (Model-View-Controller, MVC), которые позволяют динамически отображать веб-страницы в зависимости от действий контроллера и наличия данных из моделей. Наконец, мы рассмотрим Razor Pages — тип приложения ASP.NET Core, в котором у каждой веб-страницы есть связанный с ней объект на стороне сервера.

В *главе 7 "Blazor и WebAssembly в .NET"* рассмотрен Blazor — фреймворк, позволяющий разработчикам писать .NET-код, который может выполняться в браузерах. Это можно сделать двумя путями — с помощью Blazor WebAssembly либо Blazor Server.

Приложение Blazor WebAssembly компилируется в исполняемый файл, который может запускаться непосредственно в браузерах. Он также может быть размещен в стандартном приложении ASP.NET Core. Blazor Server, с другой стороны, выполняет весь код на сервере. В этом случае компонент в браузере будет взаимодействовать с кодом на сервере в реальном времени через SignalR. Каждая из этих моде-

лей хостинга имеет свои плюсы и минусы, хотя программный код может быть очень похожим. В данной главе представлен обзор всех этих моделей хостинга.

В *главе 8 "SignalR и двусторонняя связь"* представлена SignalR — встроенная библиотека ASP.NET Core, которая позволяет клиенту и серверу осуществлять двустороннее взаимодействие в реальном времени. В этой главе показано, как создавать серверные компоненты SignalR, а также как настраивать различные типы клиентов.

Мы рассмотрим два типа клиентов SignalR — JavaScript и .NET. Клиенты обоих типов можно использовать либо в браузере, либо как отдельные приложения. Например, JavaScript — это язык, изначально предназначенный для внутрибраузерных приложений. Однако благодаря таким технологиям, как Node.js, его можно использовать и в автономных приложениях. .NET в основном предназначен для автономных приложений. Но с помощью такой технологии, как Blazor, их можно выполнять и в браузере.

В *главе 9 "gRPC в ASP.NET Core"* описан gRPC — протокол-оболочка, который основан на HTTP/2 и обеспечивает эффективный обмен сообщениями. В этой главе показано, как использовать gRPC в ASP.NET Core. Мы рассмотрим все фундаментальные концепции gRPC. Вы познакомитесь с протоколом Protobuf, который применяется в gRPC для обмена сообщениями. И узнаете, как с помощью Protobuf настраивать серверные и клиентские компоненты gRPC.

Также вы ознакомитесь со всеми четырьмя типами вызовов, которые поддерживает gRPC: одиночным, потоковым серверным, потоковым клиентским и потоковым двунаправленным. Наконец, вы познакомитесь со всеми типами данных, которые поддерживает Protobuf.

В *главе 10 "Машинное обучение с помощью ML.NET"* мы познакомимся с ML.NET — библиотекой, позволяющей разработчикам с относительной легкостью создавать на платформе .NET приложения машинного обучения. Например, эта библиотека позволяет выбрать алгоритм машинного обучения и сгенерировать для него модель в коде C#. Затем на основе этой модели можно реализовать различные сценарии.

Из этой главы вы узнаете, как использовать ML.NET. Сначала мы рассмотрим базовые возможности этой технологии. Затем создадим несколько примеров моделей машинного обучения с помощью некоторых наиболее популярных алгоритмов. После этого вы познакомитесь с основами обучения и оценки вашей модели машинного обучения. Кроме того, будет показано, как использовать малокодовый построитель моделей для создания модели машинного обучения посредством графического пользовательского интерфейса.

В *главе 11 "Микросервисы и контейнеризация в .NET 7"* мы расскажем о микросервисах, контейнерах и управлении ими. Крупномасштабные приложения часто развертывают в виде взаимосвязанных микросервисов, которые можно масштабировать по мере необходимости. И, как правило, лучший способ развертывания микросервисов — контейнеры. Это позволяет обеспечить согласованное поведение каждого сервиса независимо от того, в какой среде он развернут. Поскольку он работает в отдельной изолированной среде, на него не влияют процессы, происходя-

щие в операционной системе хоста, если только он не был явно открыт для таких процессов.

В этой главе вы познакомитесь с образами контейнеров .NET 7 Docker и процессом интеграции функций Docker в проекты .NET. Затем мы изучим основы оркестрации контейнеров на примере Docker Swarm и Kubernetes.

Комплект исходного кода и цветные иллюстрации

Перейдите по следующей ссылке для загрузки *комплекта исходного кода* и *цветных иллюстраций* к данной книге: **https://rebrand.ly/4xyu6op**.

Пакет исходного кода для книги также размещен на GitHub по адресу **https://github.com/bpbpublications/Implementing-C-Sharp-11-and-.NET-7.0**.

В случае обновления кода он будет обновлен и в существующем репозитории GitHub.

Мы предлагаем подборки кода из нашего обширного каталога книг и видеоматериалов по адресу **https://github.com/bpbpublications**. Посмотрите обязательно!

Глава 1

Знакомство со структурой приложений .NET 7

Введение

В ноябре 2022 года компания Microsoft представила очередную версию своей кросс-платформенной среды разработки программного обеспечения под названием .NET 7. Эта среда поддерживает целый ряд языков программирования, однако наиболее популярным языком .NET является C#, а его новая версия, C# 11, представляет собой неотъемлемую часть .NET 7.

Основное преимущество использования .NET 7 по сравнению с другими платформами разработки ПО заключается в том, что она может работать под управлением любой из основных операционных систем персональных компьютеров, к которым относятся Windows, MacOS и Linux. В частности, в .NET 7, вы сможете создавать приложения, работающие как на ПК, так и на мобильных устройствах. В следующих главах книги вы узнаете, как это сделать.

Эта книга научит вас применять новейшие возможности как .NET 7, так и C# 11. Независимо от того, являетесь ли вы опытным разработчиком .NET или только недавно начали работать с C# и .NET, эта книга даст вам достаточно знаний по данным предметам, чтобы вы смогли создавать свои собственные .NET-приложения.

Если вы уже встречались с .NET и C#, то эта книга даст вам хорошее представление о новейших возможностях как самой платформы, так и языка. Если вы начинающий пользователь C#, то сначала вам следует ознакомиться с базовым синтаксисом C#, поскольку основное внимание далее уделено демонстрации новейших возможностей.

Однако мы все же кратко напомним все основные положения. Кроме того, в конце этой главы будут приведены тщательно отобранные ссылки на официальную документацию по языку. Поэтому, независимо от того, начинаете ли вы изучать .NET или уже являетесь опытным разработчиком ПО, специализирующимся на .NET, эта книга будет для вас полезной.

Структура

В этой главе мы рассмотрим следующие темы:

- настройка среды разработки;
- создание приложения .NET 7;
- обзор структуры проекта .NET 7;
- основы C# 11 и встроенные типы данных.

Цели

В этой главе мы сосредоточимся на настройке среды разработки и создании простого приложения с помощью шаблонов .NET 7. Затем, создав наш первый проект, мы вспомним некоторые основы языка C#.

Следующие главы будут посвящены новым возможностям .NET 7 и C# 11. Но прежде чем двигаться дальше, нам необходимо настроить фундаментальные зависимости. Итак, начнем.

Настройка среды разработки

Для начала работы с .NET 7 и C# 11 вам потребуется следующее:

- компьютер с операционной системой Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода.

Если у вас отсутствуют какие-либо из этих необходимых компонентов, давайте рассмотрим, что нужно сделать для их получения.

Компьютеры, подходящие для разработки ПО

Поскольку .NET — это кросс-платформенный фреймворк разработки ПО, он будет работать как под Windows, так и под Linux или MacOS. Поэтому подойдет компьютер, работающий под управлением одной из этих операционных систем. Кроме того, .NET совместим с различными архитектурами процессоров. Он будет работать с любыми процессорами Intel/AMD или ARM.

Что касается вычислительной мощности, дискового пространства и объема памяти, то для этого подойдет любой средний ноутбук или настольный компьютер потребительского класса. Для выполнения программного кода .NET не нужна сверхмощная машина. Однако я бы рекомендовал компьютер с оперативной памятью не менее 8 Гбайт.

.NET 7 SDK

Комплект средств разработки программного обеспечения (Software Development Kit, SDK) .NET содержит все, что необходимо .NET-разработчику, включая плат-

форму, компилятор и все вспомогательные инструменты. Последнюю версию .NET 7 SDK можно установить, выполнив следующие действия:

1. Для получения .NET 7 SDK зайдите на веб-страницу по адресу: **https://dotnet.microsoft.com/**.
2. На этой странице необходимо нажать кнопку **Download** (Загрузить) (рис. 1.1).

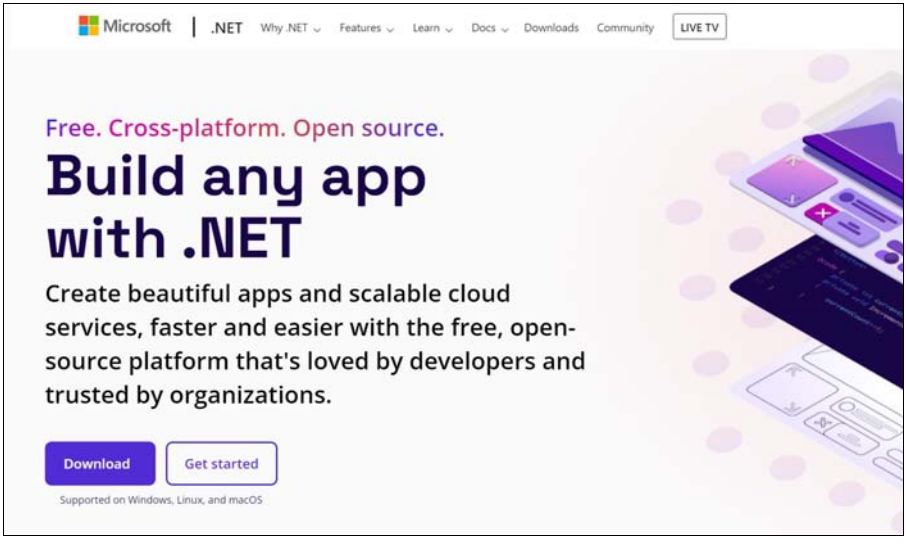


Рис. 1.1. Кнопка **Download** (Загрузить) на странице Microsoft .NET

3. После этого выполняется переход на страницу загрузки, где вам следует выбрать последнюю версию .NET 7 SDK, убедившись, что выбран именно вариант SDK, а не Runtime. Среда выполнения .NET 7 Runtime позволит выполнять приложения .NET 7 на вашем компьютере, но вы не сможете их создавать. Пакет .NET 7 SDK содержит не только среду выполнения, но и все необходимые средства разработки, включая интерфейс командной строки (Command Line Interface, CLI), которые мы подробно рассмотрим в *главе 3*.
4. Теперь подошло время установить подходящую среду IDE или редактор кода, если вы этого еще не сделали.

Установка редактора кода или среды IDE

Неважно, что вы выберете для разработки приложений — редактор кода или IDE. Возможен любой из этих вариантов. И независимо от вашего выбора вы сможете выполнять все упражнения, приведенные в этой книге. Однако было бы полезно знать разницу между ними, чтобы решить, какой инструмент будет более подходящим для вас.

Редактор кода позволяет писать текст программы и перемещаться по нему. В комплект поставки входят различные средства форматирования и выделения текста, поэтому ваш код будет легко читаться. Но этим возможности редактора кода

практически и ограничиваются. Как правило, для того чтобы собрать приложение из исходного кода, необходимы некоторые внешние инструменты или дополнительные плагины. Однако, поскольку возможности редакторов кода ограничены, они, как правило, намного компактнее IDE и быстрее загружаются.

Интегрированная среда разработки (Integrated Development Environment, IDE) может делать абсолютно все, что и редактор кода, и даже больше. Такие возможности, как создание новых проектов на основе различных шаблонов, запуск и отладка приложений, а также встраивание исходного кода в развертываемое приложение, доступны прямо "из коробки". И всеми этими возможностями можно управлять через графический интерфейс пользователя (Graphic User Interface, GUI). Но за все это приходится платить производительностью. Как правило, IDE медленнее устанавливается, занимает достаточно большую часть дискового пространства и загружается заметно дольше, чем редактор кода, особенно если вы запускаете ее на недостаточно мощном компьютере.

Так какой инструмент будет целесообразнее для разработки приложений с помощью .NET 7? Вы можете выбрать любой из них, в зависимости от ваших предпочтений. Если вы не против сборки и тестирования своего приложения с помощью интерфейса командной строки (CLI), который поставляется с .NET 7 SDK, вероятно, вам будет достаточно редактора кода. Однако если вы предпочитаете удобство работы с графическим пользовательским интерфейсом и не возражаете против увеличения времени загрузки и периодического снижения производительности, то вам, вероятно, следует выбрать IDE. Кроме того, я бы порекомендовал использовать IDE новичкам.

Теперь давайте рассмотрим, какие редакторы кода и IDE будут работать с .NET 7 и C# 11.

Установка подходящего редактора кода

Если вы предпочитаете простой редактор кода, то лучшим выбором будет Visual Studio Code. И не имеет значения, какая ОС установлена на вашем компьютере. Для всех поддерживаемых операционных систем существует доступная версия Visual Studio Code.

Visual Studio Code можно загрузить со следующей веб-страницы:
https://code.visualstudio.com/download.

При этом сначала вам необходимо выбрать вариант загрузки, соответствующий вашей ОС и архитектуре вашего процессора (рис. 1.2).

После загрузки вам просто нужно будет следовать указаниям по установке, соответствующим имеющейся ОС. После установки вам нужно будет загрузить плагин C# для редактора кода, чтобы выделение всех частей кода C# осуществлялось правильно. Это можно сделать сразу, либо вам будет предложено сделать это при первом вызове редактора для открытия любого файла с расширением cs. Если вы решите установить плагин сразу, то нужно будет открыть Visual Studio Code и щелкнуть на расположенной слева панели расширений, которая представлена символом,

состоящим из четырех квадратов. После этого вам следует ввести в поле поиска C# и установить первый указанный в результатах поиска плагин, как показано на рис. 1.3.

И это все, что вам нужно, чтобы начать создавать свои приложения. Однако, если вы предпочитаете IDE, то потребуются дополнительные шаги, описанные далее.

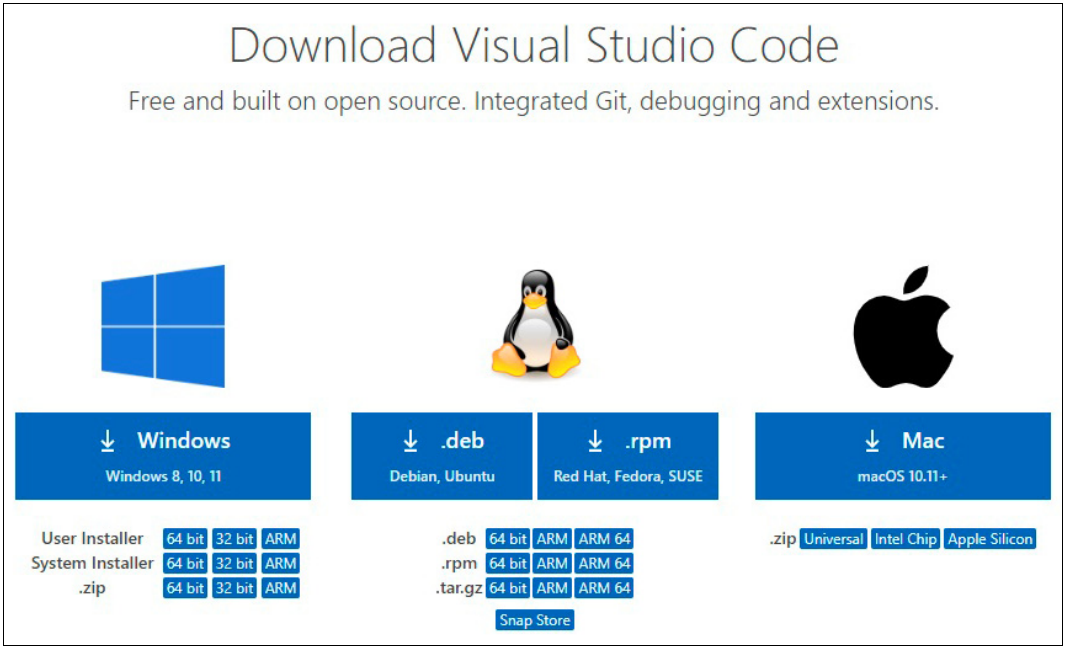


Рис. 1.2. Страница загрузки редактора кода Visual Studio Code

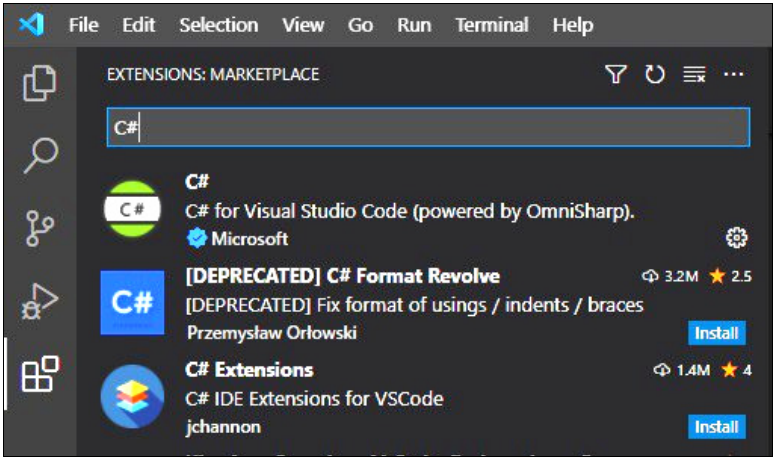


Рис. 1.3. Расширение C# в редакторе Visual Studio Code

Установка подходящей IDE

Установить IDE не так просто, как редактор кода. Visual Studio Code — единственный рекомендуемый редактор кода для разработки в .NET, независимо от того, какая у вас ОС. Но если говорить о IDE, то в разных операционных системах доступны разные варианты. Имеющиеся возможности можно резюмировать следующим образом:

Windows:

- Microsoft Visual Studio 2022;
- JetBrains Rider.

MacOS:

- Microsoft Visual Studio 2022 for Mac;
- JetBrains Rider.

Linux:

- JetBrains Rider.

Как видите, единственная общая IDE — это JetBrains Rider. Но у этой среды есть свои особенности, поэтому она может оказаться не лучшим вариантом на все случаи жизни. На самом деле у каждой IDE из этого списка есть свои плюсы и минусы. Мы рассмотрим каждый из вариантов, чтобы вы могли решить, какую IDE выбрать.

Microsoft Visual Studio 2022

Это официальная интегрированная среда разработки для .NET от Microsoft. Данная среда была разработана в основном для работы с .NET, однако она поддерживает целый ряд различных платформ, языков и технологий. Несмотря на свое название, она никак не связана с Visual Studio Code. Она и выглядит, и воспринимается по-другому. Общее между ними только то, что оба эти продукта выпускаются компанией Microsoft и оба могут использоваться для написания кода.

В комплект поставки входят все необходимые инструменты. Кроме того, вы можете получить этот продукт бесплатно, т. к. для него предусмотрена так называемая редакция сообщества — *community edition*. Также выпускаются премиальные редакции *professional* и *enterprise*, для работы с которыми необходимо приобрести лицензию. Они содержат больше инструментов, чем бесплатная версия для сообщества разработчиков. Тем не менее даже редакция *community* поставляется с достаточным количеством инструментов для разработки .NET-приложений. Для выполнения упражнений, приведенных в этой книге, вам, безусловно, не потребуется ничего, кроме издания *community*.

Самое большое преимущество Visual Studio 2022 по сравнению любой другой IDE — постоянное обновление синхронно с обновлениями .NET. Таким образом, при каждом обновлении .NET SDK (даже если это только предварительная версия) сразу же выпускается обновление для Visual Studio, обеспечивающее совместимость. В результате вы можете быть уверены, что ваша IDE всегда будет поддерживать новейшие возможности .NET.

Для загрузки Visual Studio 2022 можно посетить ее официальную страницу по следующей ссылке: **https://visualstudio.microsoft.com/downloads/**.

Перед вами откроется экран, показанный на рис. 1.4, где можно выбрать версию для загрузки. Выберите вариант `Community`, если вы не уверены в том, какая версия вам нужна. При необходимости вы всегда сможете перейти на другую редакцию позже.

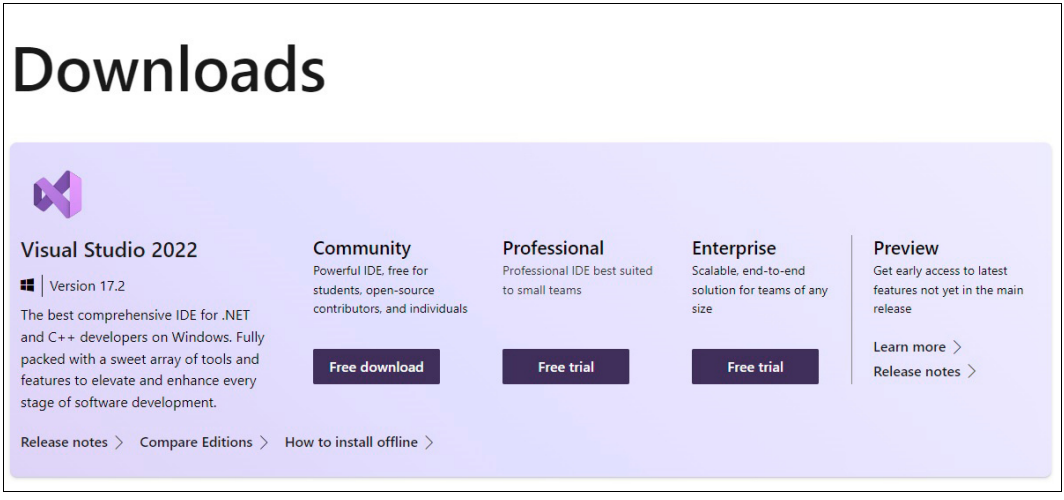


Рис. 1.4. Страница загрузки Visual Studio 2022

После начала загрузки вам останется только следовать указаниям по установке. Однако следует иметь в виду, что и загрузка, и установка могут занять некоторое время, поскольку Visual Studio 2022 является достаточно большой IDE.

Visual Studio 2022 предоставляет все необходимые инструменты, но главный ее недостаток заключается в том, что она доступна только для Windows. Существует IDE под названием Visual Studio for Mac, но, несмотря на название, она не является версией той же самой IDE для Mac. Это совершенно другая программа. И именно ее мы рассмотрим далее.

Microsoft Visual Studio 2022 for Mac

Если вы используете Mac, а не Windows, то Visual Studio 2022 for Mac может стать хорошим выбором IDE. Эта среда разработки имеет достаточный набор инструментов для создания .NET-приложений, но она проще, чем Windows-версия Visual Studio 2022. Кроме того, графический интерфейс этой IDE выглядит совершенно иначе, поэтому если вы ранее работали с Visual Studio под Windows, а потом перешли на Mac, то вам потребуется некоторое время, чтобы привыкнуть к нему.

Еще один недостаток Visual Studio for Mac состоит в том, что ее разработка несколько отстает от разработки Visual Studio для Windows и она не успевает за развитием .NET. Иногда даже приходится ждать месяцы, прежде чем вам будут доступны новые возможности .NET. А иногда единственный способ доступа к новым

возможностям .NET в этой IDE — установка предварительной версии, которая как часть ПО, еще не одобренная для официального выпуска, может содержать некоторые ошибки.

Загрузить Visual Studio for Mac можно с ее официальной страницы по следующей ссылке: **https://visualstudio.microsoft.com/vs/mac/**.

Поскольку существует только одна версия этой IDE, вам будет предложена единственная кнопка загрузки, как показано на рис. 1.5.



Рис. 1.5. Страница загрузки Visual Studio 2022 for Mac

Далее необходимо просто следовать указаниям по загрузке и установке, которые не требуют пояснений. Существует также третий вариант IDE, выпущенный компанией JetBrains под названием Rider. И он стоит вашего внимания независимо от применяемой ОС.

JetBrains Rider

Основное преимущество Rider перед другими IDE заключается в том, что по умолчанию она поставляется с множеством встроенных инструментов. Она автоматически находит потенциально проблемный код и выдает предложения по переработке структуры кода. Эта среда способна декомпилировать библиотеки сторонних разработчиков, что позволяет увидеть исходный код, на котором они были написаны. И этот список достоинств можно продолжить.

Если вы являетесь пользователем Windows, то за относительно невысокую цену вы получите гораздо более насыщенную функциями IDE, чем Visual Studio. Это будет еще заметнее, если вы пользуетесь Mac. А если вы предпочитаете Linux, то это будет ваш единственный вариант. IDE будет выглядеть идентично и обладать одинаковой функциональностью независимо от того, под какой ОС она запущена.

Эта среда имеет свои недостатки. Но я могу назвать только два. У нее нет бесплатной версии. После завершения первоначального 30-дневного пробного периода необходимо приобрести лицензию. Однако цена на нее, как правило, ниже, чем на редакцию *professional* или *enterprise* Visual Studio. Второй недостаток заключается

в том, что, поскольку эта IDE создана сторонней компанией, а не Microsoft, она иногда немного отстает от выпуска новых обновлений .NET SDK. Однако команда разработчиков Rider, как правило, работает быстро, поэтому такие задержки обычно не слишком велики. В большинстве случаев она обновляется быстрее, чем Visual Studio for Mac.

Rider можно получить на официальной странице загрузки по следующей ссылке: **https://www.jetbrains.com/rider/**. Веб-страница должна автоматически определить, под какой ОС вы работаете, поэтому вам будет показана ссылка на загрузку, соответствующая вашей ОС (рис. 1.6). Затем вам остается лишь следовать указаниям по установке.



Рис. 1.6. Страница загрузки IDE Rider

Будем считать, что к этому моменту вы уже выбрали и установили подходящую для вас IDE или редактор кода. Теперь мы готовы приступить к созданию наших первых приложений .NET 7 и изучению их структуры.

Создание приложения .NET 7

При создании .NET-приложения вы работаете с проектами и решениями. *Проект* — это набор файлов кода, которые впоследствии будут собраны в один исполняемый файл или одну справочную библиотеку. Эти файлы образуют приложение. Приложение может состоять из одного исполняемого файла или иметь основной исполняемый файл, а также ряд других файлов, обеспечивающих дополнительную функциональность. Эти другие файлы называют библиотеками, и их могут совместно использовать различные приложения. Библиотеки также представлены в исходном коде проектами.

Решение — это то, что объединяет несколько проектов. Оно представлено файлом с расширением sln, который размещается рядом с папками проекта в исходном коде. Наличие решения не обязательно, однако оно полезно, если вы используете IDE, т. к. это облегчит управление и организацию связанных проектов.

Как проекты, так и решения можно создавать как через интерфейс командной строки (CLI), так и через графический интерфейс IDE. И в этом разделе мы рассмотрим оба метода. Далее в этой книге мы будем использовать в основном команды CLI, поскольку они одинаковы для всех операционных систем. Кроме того, .NET CLI

поставляется вместе с .NET SDK, поэтому если вы установили SDK, то у вас уже есть и CLI.

Создание приложения с помощью CLI

К наиболее простому типу .NET-приложений относятся консольные приложения, которые не имеют графического интерфейса. Единственный способ взаимодействия с внешним миром — это текстовый интерфейс, например CMD, PowerShell, Terminal, Shell и т. д. И в силу такой простоты это идеальный тип приложения для нашей демонстрации.

Теперь перейдем к созданию нашего решения. Для этого нужно открыть любой терминал командной строки на ваш выбор, создать папку, в которую будет помещено решение, перейти в эту папку и выполнить следующую команду:

```
dotnet new sln
```

В результате в этой папке будет создан файл с именем, совпадающим с именем папки и расширением sln. Таким образом, если предположить, что папка называется BasicApp, файл решения будет носить имя BasicApp.sln.

Теперь создадим проект. Для демонстрационных целей назовем его `BasicConsoleApp` и создадим, выполнив следующую команду:

```
dotnet new console -o BasicConsoleApp
```

Аргумент `console` служит для указания на то, что мы используем шаблон Console Application (консольное приложение). Параметр `-o` означает *вывод*. Здесь мы помещаем название нашего проекта.

После выполнения этой команды на вашем компьютере внутри папки solution должна появиться папка с именем BasicConsoleApp. В ней вы найдете два файла: BasicConsoleApp.csproj и Program.cs. Файл с расширением csproj — это файл проекта, который предоставляет компилятору информацию о проекте. Он содержит различные XML-свойства, в том числе версию .NET. И если у вас установлена 7-я версия SDK, то она должна быть задействована автоматически. Ваш файл проекта должен выглядеть следующим образом:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

Файл Program.cs — это точка входа в ваше приложение. Он содержит код на языке C#, который будет выполняться первым при запуске приложения. По умолчанию он будет просто выводить на консоль сообщение "Hello World!", т. к. его содержимое будет выглядеть следующим образом:

```
// Дополнительную информацию см. на сайте https://aka.ms/new-console-template
Console.WriteLine("Hello, World!");
```

Теперь можно добавить созданный нами проект в решение. Для этого необходимо перейти в папку solution и выполнить следующую команду:

```
dotnet sln add BasicConsoleApp/BasicConsoleApp.csproj
```

Далее мы проделаем точно такую же процедуру, но с использованием графического интерфейса среды IDE.

Создание приложения с помощью графического интерфейса IDE

В нашем примере мы воспользуемся Visual Studio 2022 под Windows. Но процедура будет очень похожа независимо от того, какая у вас IDE. Графический интерфейс будет выглядеть по-разному. Названия некоторых элементов будут отличаться. Но принципы останутся теми же самыми.

Для начала необходимо открыть IDE и выбрать в меню пункт, позволяющий создать новый проект, как показано на рис. 1.7.

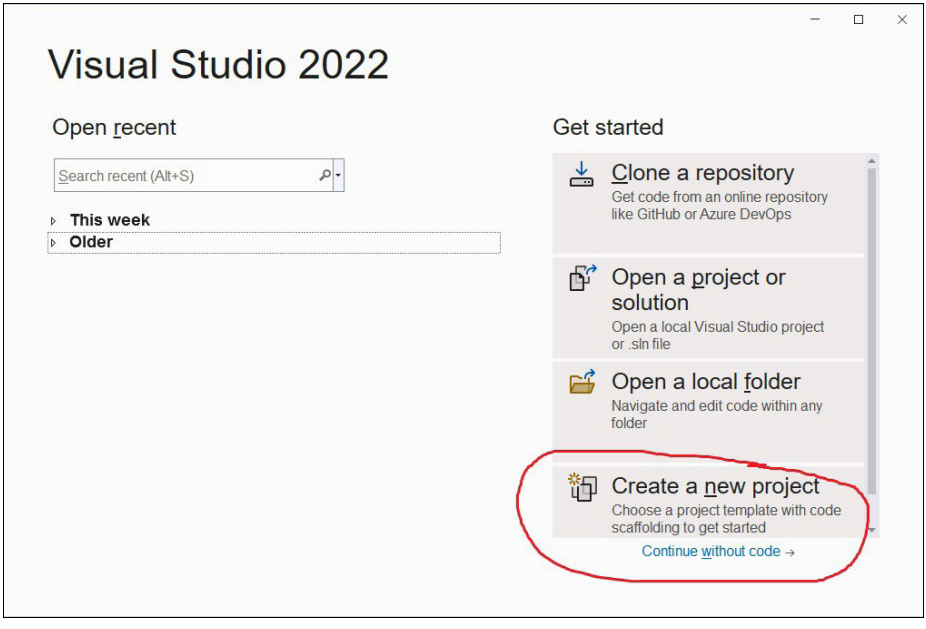


Рис. 1.7. Создание проекта в Visual Studio 2022

Затем следует выбрать опцию **Console App** (Консольное приложение) в качестве шаблона проекта, как показано на рис. 1.8. Но при этом обязательно выберите шаблон для C#, поскольку имеются шаблоны проекта и для других языков.

Затем последует ряд различных экранов, где вам будет предложено указать имена проекта и решения, а также задать версию фреймворка (необходимо выбрать

.NET 7). Детали того, как это делается в разных IDE, несколько отличаются. Например, на рис. 1.9 показано, как выглядит экран назначения имен в Visual Studio 2022.

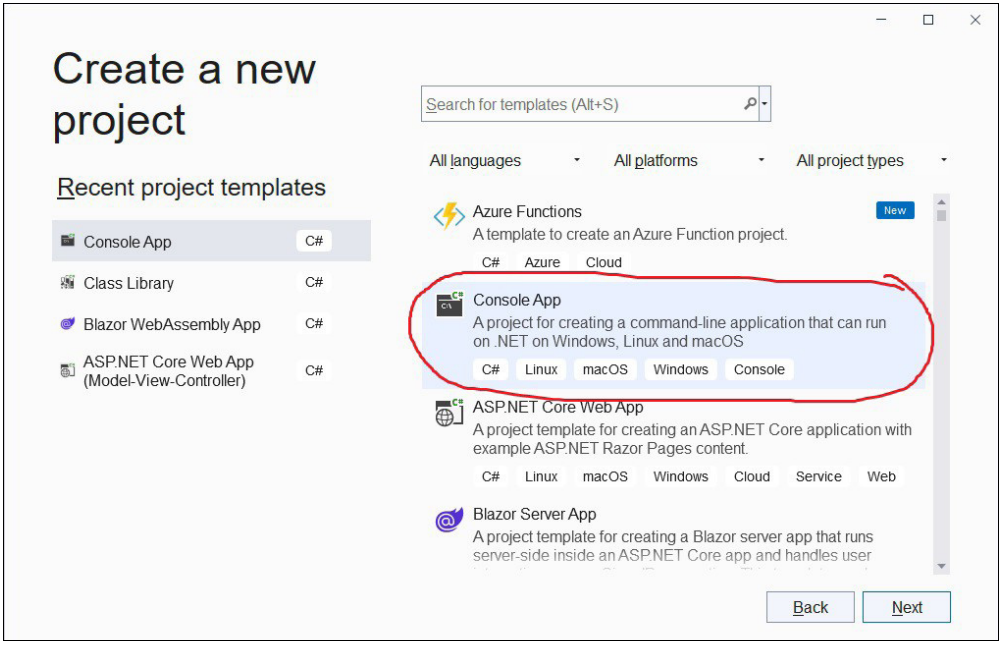


Рис. 1.8. Шаблон консольного приложения в Visual Studio 2022

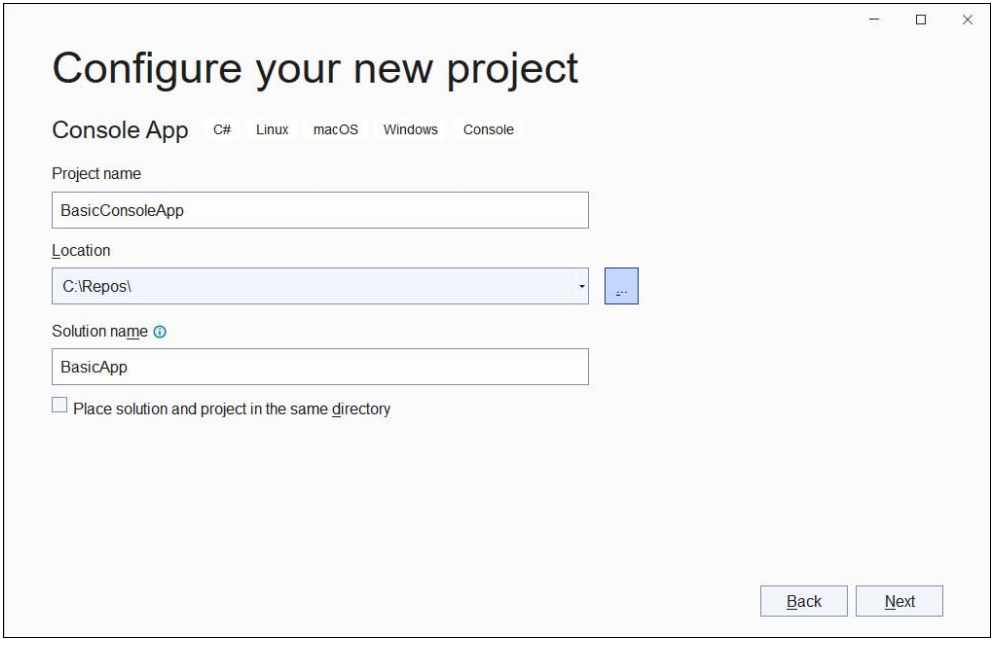


Рис. 1.9. Назначение имен решения и проекта

Затем, если оставить значения по умолчанию для всех оставшихся настроек и довести процедуру до конца, то и решение, и проект будут созданы, после чего IDE автоматически откроет решение. Оно должно выглядеть примерно так, как показано на рис. 1.10.

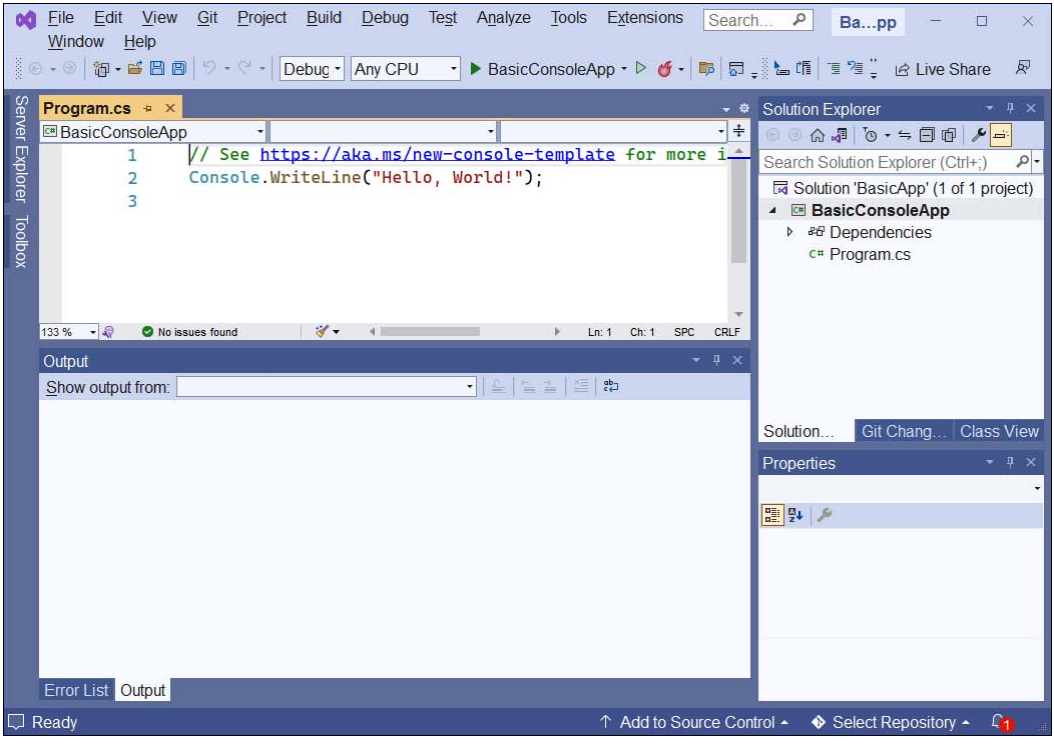


Рис. 1.10. Решение, открытое в Visual Studio 2022

Теперь, когда наше решение готово, мы можем приступить к добавлению в него некоторых полезных функций.

Обзор структуры проекта .NET 7

Чтобы продемонстрировать структуру проекта .NET 7, мы добавим в наше приложение некоторые полезные возможности. Полный пример можно найти по следующему URL-адресу: **https://github.com/fiodarsazanavets/a-complete-guide-to-implementing-csharp11-and-dotnet7/tree/main/Chapter-01**.

Добавление объекта *struct*

Прежде всего, создадим файл Customer.cs в папке проекта BasicConsoleApp.

Содержимое этого файла будет следующим:

```
namespace BasicConsoleApp;
public struct Customer
```

```
{
    public Customer(string name)
    {
        Name = name;
    }
    public string Name { get; set; }
    public short Age { get; set; }
    public int Id => randomId;
    private int randomId = (new Random()).Next();
}
```

Давайте воспользуемся этим примером, чтобы повторить базовый синтаксис C#. Мы начинаем наш файл с оператора объявления пространства имен. Пространства имен в C# служат для модульной организации кода. Если ваши объекты должны работать совместно друг с другом, вы назначаете для них одно и то же пространство имен, после чего можете ссылаться на это пространство имен, чтобы иметь возможность задействовать назначенные для него пользовательские типы данных.

В нашем примере тип данных `Customer`, который мы создали, — это структура. C# поддерживает и другие типы объектов, такие как класс и запись. Вскоре мы рассмотрим различия между ними. Но на данный момент мы решили использовать наш объект в виде структуры, чтобы продемонстрировать одну из новейших особенностей структур в C# 11.

Наш объект `Customer` имеет конструктор, который принимает параметр `name` строкового типа. Это позволяет присвоить значение свойству `Name` сразу после создания экземпляра объекта `Customer`. У него также есть два других свойства: `Age` и `Id`. Свойство `Age` представляет собой короткое целое число. По умолчанию его значение равно нулю. И это демонстрирует новую особенность, добавленную в C# 11. Если какие-либо свойства не заданы в конструкторе, им автоматически присваивается значение по умолчанию, которое для типа данных `short` равно нулю. Однако такое правило работает только для объектов типа `struct`. До выпуска C# 11 это приводило к ошибке компиляции, поскольку всем свойствам необходимо было присвоить значения до выполнения блока кода внутри конструктора.

Другое свойство, `Id`, доступно только для чтения и не может быть изменено после создания экземпляра объекта. Оно берет свое значение из поля `randomId`, которому присваивается случайное значение. И поскольку это поле приватное, область его видимости ограничена данным объектом. Все остальные члены объекта помечены словом `public`. Это ключевое слово в начале определения объектного члена называется модификатором доступа.

Добавление интерфейса и класса

Далее мы добавим в папку нашего проекта файл CustomerRepository.cs, и первое, что мы вставим в этот файл, будет следующее определение интерфейса:

```
namespace BasicConsoleApp;
public interface ICustomerRepository
```

```
{
    int Count { get; }
    void AddNewCustomer(Customer customer);
    Customer GetCustomer(int id);
    Customer GetCustomer(string name);
    IEnumerable<Customer> GetCustomers();
    IEnumerable<Customer> GetCustomers(string nameMatch);
}
```

Как и ранее, в самом начале мы добавили определение пространства имен. Поскольку пространство имен у нас такое же, как и в структуре `Customer`, нам не нужно импортировать в наш файл дополнительное пространство имен. Мы имеем неявный доступ к любому другому объекту, использующему то же пространство имен.

Интерфейсы не являются функциональными объектами, поэтому их нельзя задействовать напрямую. Они действуют как контракты для определений объектов. Интерфейс предоставляет сигнатуры для всех публичных членов, которые должен содержать объект, реализующий этот интерфейс. В противном случае код не будет компилироваться. Именно поэтому у членов интерфейса нет ни тела, ни модификатора доступа.

Однако интерфейс можно назначить в качестве типа возвращаемых данных для полей, свойств, методов и переменных. Если это сделать, то он может быть назначен абсолютно любому объекту, реализующему этот интерфейс. И здесь представлен пример этого. Для обеих версий нашего метода `GetCustomers` задан тип данных `IEnumerable`. `IEnumerable` — это встроенный интерфейс, предназначенный для реализации коллекций данных. И в нашем случае мы сможем возвращать коллекции абсолютно любого типа, если они реализуют этот интерфейс.

Теперь мы добавим в наш файл следующее определение класса:

```
internal class CustomerRepository : ICustomerRepository
{
    private readonly List<Customer> customers;
    public CustomerRepository()
    {
        customers = new List<Customer>();
    }
    public int Count => customers.Count;
    public void AddNewCustomer(Customer customer)
    {
        customers.Add(customer);
    }
    public Customer GetCustomer(int id)
    {
        return customers.SingleOrDefault(c => c.Id == id);
    }
    public Customer GetCustomer(string name)
    {
        return customers.SingleOrDefault(c => c.Name == name);
    }
```

```
    public IEnumerable<Customer> GetCustomers()
    {
        return customers;
    }
    public IEnumerable<Customer> GetCustomers(string nameMatch)
    {
        return customers.Where(c => c.Name.Contains(nameMatch));
    }
}
```

Этот класс реализует интерфейс, который мы определили ранее, т. к. после имени класса стоит двоеточие, а затем имя интерфейса. В языке C# двоеточие в данном контексте служит оператором реализации или наследования.

Вы также могли заметить, что класс имеет модификатор доступа `internal`, а не `public`. Между ними есть разница. Если модификатор доступа `public` делает объект видимым для всех остальных объектов, как внутри проекта, так и в любом проекте, который ссылается на ваш проект, то модификатор `internal` ограничивает видимость только теми объектами, которые существуют внутри этого же проекта. При реализации интерфейсов можно переходить от более низкого уровня ограничений к более высокому, но не наоборот. В нашем примере это проявляется в том, что публичный интерфейс реализуется внутренним классом.

Итак, у нас есть приватный список объектов `Customer`, доступный только для чтения. Когда мы применяем модификатор "только для чтения", экземпляр значения поля может быть создан только в момент объявления поля или в конструкторе объекта. Мы не можем присвоить этому полю новое значение впоследствии.

После этого мы лишь предоставляем дополнения для всех членов, которые мы ранее определили в интерфейсе. Обратите внимание, что у нас есть несколько методов с дублирующимися названиями. Это нормально, т. к. у них разные сигнатуры. Сигнатура метода определяется комбинацией имени, типа возвращаемых данных и входных параметров. Возможность определять методы с одинаковым именем и идентичным типом данных, но с разными параметрами, называется *полиморфизмом*.

Итак, мы реализуем следующие члены интерфейса:

- `Count` — возвращает текущее число клиентов в списке.
- `AddNewCustomer(Customer customer)` — принимает в качестве входного параметра объект `Customer` и добавляет его в список клиентов.
- `GetCustomer(int id)` — извлекает один элемент `Customer` на основе входного параметра `id`.
- `GetCustomer(string name)` — извлекает один объект `Customer` по имени.
- `GetCustomers()` — возвращает все записи из списка клиентов.
- `GetCustomers(string nameMatch)` — возвращает из списка клиентов только те элементы, имена которых содержат определенный шаблон.

Последние два метода демонстрируют, как можно вернуть конкретную реализацию, если возвращаемый тип был определен как интерфейс. Возвращаемый тип для обоих методов — `IEnumerable`. Но мы возвращаем `List`. Мы можем делать так, потому что тип данных `List` реализует `IEnumerable`.

Изменение точки входа в приложение

Далее мы заменим содержимое нашего файла Program.cs. Начнем с удаления из него существующего содержимого. Затем добавим следующий код:

```
using BasicConsoleApp;
// Создание репозитория
var customerRepository = new CustomerRepository();
// Создание объектов Customer
var customer1 = new Customer("John Smith");
var customer2 = new Customer("David Smith");
var customer3 = new Customer("Gary Rogers");
```

Так как класс `Program.cs` не содержит определения пространства имен, нам необходимо импортировать пространство имен объектов, которые мы только что создали. Для этого мы используем оператор, за которым следует имя пространства имен.

Затем мы создаем экземпляр объекта `CustomerRepository` и сохраняем его в переменной `customerRepository`. После этого мы создаем три экземпляра объекта `Customer` и сохраняем их в именованной переменной. Переменная — это временное хранилище данных в памяти. Переменные обычно определяют с помощью ключевого слова `var`, вместо которого можно явно указать тип данных. Тем не менее в большинстве ситуаций рекомендуется указывать `var`.

Затем мы добавляем следующий код, в котором немного изменяем данные в некоторых объектах `Customer`, после чего добавляем их все в `customerRepository`:

```
// Ввод дополнительных данных
customer1.Age = 30;
customer2.Age = 21;
// Добавление клиентов в репозиторий
customerRepository.AddNewCustomer(customer1);
customerRepository.AddNewCustomer(customer2);
customerRepository.AddNewCustomer(customer3);
```

Наконец, добавим в наш файл следующий код, в котором мы будем пробовать различные методы извлечения экземпляров объекта `Customer` из `CustomerRepository`:

```
// Извлечение данных из репозитория
Console.WriteLine("The following data has been obtained while iterating through all customers");
foreach (var customer in customerRepository.GetCustomers())
{
    Console.WriteLine($"""
    Customer id: {customer.Id},
```

```
    Customer Name: {customer.Name},
    Customer Age: {customer.Age}
    """);
}
// Извлечение отфильтрованных данных
Console.WriteLine("The following data has been obtained while iterating through customers
while filtering by 'Smith' in name:");
foreach (var customer in customerRepository.GetCustomers("Smith"))
{
    Console.WriteLine($"""
    Customer ID: {customer.Id},
    Customer Name: {customer.Name},
    Customer Age: {customer.Age}
    """);
}
// Извлечение одного клиента по имени
Console.WriteLine("The following data was returned for David Smith:");
var specificCustomer = customerRepository.GetCustomer("David Smith");
Console.WriteLine($"""
    Customer ID: {specificCustomer.Id},
    Customer Name: {specificCustomer.Name},
    Customer Age: {specificCustomer.Age}
    """);
```

Этот пример демонстрирует еще одну новую возможность языка, которая была добавлена в C# 11, — многострочные строковые литералы. Если обернуть строковое значение тремя двойными кавычками, то все, написанное внутри них, будет рассматриваться как литеральный символ, а не как специальный символ C#. Но эту возможность также можно объединить с функцией строковой интерполяции, если поставить в начало знак доллара. В результате появляется возможность вставить в строку данные из вашего кода. Для этого необходимо заключить код в фигурные скобки внутри строки.

Теперь мы можем запустить наше приложение, для чего нужно либо щелкнуть правой кнопкой мыши в окне IDE и выбрать пункт **Run** (Выполнить), либо выполнить следующую команду внутри папки проекта:

```
dotnet run
```

Ожидаемый результат показан на рис. 1.11.

Как мы видим, возраст Гэри Роджерса составляет 0 лет. Это связано с тем, что мы не задали возраст данного конкретного клиента в явном виде. Он был просто инициализирован значением по умолчанию для соответствующего типа данных, которое оказалось равным нулю. Следовательно, механизм автоматической инициализации свойств `struct` значениями по умолчанию сработал.

Далее мы кратко напомним основы языка C# и его встроенные типы данных. Если вы только начинаете свой путь программиста на C#, то для более детального изучения основ синтаксиса C# будут приведены ссылки на различные источники документации в Интернете.

```
The following data has been obtained while iterating through all customers:
Customer id: 452867481,
Customer Name: John Smith,
Customer Age: 30

Customer id: 31176956,
Customer Name: David Smith,
Customer Age: 21

Customer id: 2062322027,
Customer Name: Gary Rogers,
Customer Age: 0

The following data has been obtained while iterating through customers while filtering by 'Smith' in name:
Customer id: 452867481,
Customer Name: John Smith,
Customer Age: 30

Customer id: 31176956,
Customer Name: David Smith,
Customer Age: 21

The following data was returned for David Smith:
Customer id: 31176956,
Customer Name: David Smith,
Customer Age: 21
```

Рис. 1.11. Вывод нашего приложения

Основы C# 11 и встроенные типы данных

Многие возможности языка C# существуют с момента появления его первой версии, и многие из них до сих пор актуальны. Однако с течением времени C# развивался, и некоторые из старых функций устаревали, поскольку были изобретены более совершенные способы решения задач. Есть и такие части языка, которые изменили свою парадигму. Например, тип данных `string` раньше был значимым типом, т. е. при использовании его в качестве параметра передавалось только его значение. Если параметр изменяется внутри метода, в который он был передан, то исходная строковая переменная не меняется. Начиная с C# 10, строка стала ссылочным типом. Теперь, если строка передается в качестве параметра, то изменения, внесенные в этот параметр внутри метода, в который он был передан, приведут к тому, что эти изменения будут применены к исходной переменной.

Однако по-прежнему существуют различные способы реализации прежнего поведения для тех частей языка, поведение которых было изменено. Кроме того, в языке сохраняются прежние подходы, что обеспечивает его обратную совместимость со старыми версиями. Но в наших примерах мы будем ориентироваться исключительно на то, как все работает в C# 11. Начнем с перечисления всех встроенных типов данных.

Встроенные типы данных

Встроенные типы данных поддерживаются языком изначально. Они доступны в языке даже без дополнительных библиотек.

Для каждого из этих типов данных предусмотрено значение по умолчанию. Иными словами, если вы используете переменную, поле или свойство определенного типа и не задали значения явным образом, то будет принято значение по умолчанию.

Типы данных в C# также можно разделить на значимые и ссылочные. Когда вы передаете переменную значимого типа в метод, то туда передается только значение этой переменной, а не сама переменная (если только вы не решили передать переменную явно по ссылке с помощью ключевого слова `ref`). Если в метод передается переменная ссылочного типа, то речь идет о ссылке на всю передаваемую переменную, а не только на ее значение. Поэтому, если переданная переменная изменяется внутри метода, то изменяется и исходная переменная.

В табл. 1.1 приведены все встроенные типы данных, доступные в C#.

Таблица 1.1. *Встроенные типы данных в C#*

Название	Описание	Ссылочный или значимый тип	Значение по умолчанию
`bool`	Представление булевой величины, которая может принимать значение "истина" или "ложь"	Значимый тип	"Ложь"
`byte`	Представление одного байта, состоящего из 8 битов. При переводе в десятичную систему может принимать значения в диапазоне от 0 до 255	То же	`0`
`sbyte`	Представление одного байта, состоящего из 8 битов. Отличается от байта диапазоном принимаемых значений, который при переводе в десятичную систему составляет от –128 до 127	То же	`0`
`char`	Один символ	То же	`"\0"`
`decimal`	Числовой тип данных, поддерживающий десятичные дроби высокой точности	То же	`0.0M`
`float`	Числовой тип данных, поддерживающий десятичные дроби. Он меньше, чем `double`, но имеет гораздо меньшую точность	То же	`0.0F`
`double`	Числовой тип данных, аналогичный `float`, но имеющий вдвое бóльшую точность и вдвое больший размер	То же	`0.0D`
`int`	Числовое значение, не поддерживающее десятичную точку и имеющее размер 16 битов	То же	`0`
`uint`	Числовое значение, не поддерживающее десятичную точку и отрицательные значения. Имеет размер 16 битов	То же	`0`
`nint`	Указатель на область памяти, содержащую целочисленное значение	То же	`0`
`nuint`	Указатель на область памяти, содержащую беззнаковое целочисленное значение	То же	`0`
`long`	Числовое значение, не поддерживающее десятичную точку и имеющее размер 64 бита	То же	`0`

Таблица 1.1 *(окончание)*

Название	Описание	Ссылочный или значимый тип	Значение по умолчанию
`ulong`	Числовое значение, не поддерживающее десятичную точку и отрицательные значения. Имеет размер 64 бита	То же	`0`
`short`	Числовое значение, не поддерживающее десятичную точку и имеющее размер 32 бита	То же	`0`
`ushort`	Числовое значение, не поддерживающее десятичную точку и отрицательные значения. Имеет размер 32 бита	То же	`0`
`string`	Представление произвольного текста	Ссылочный тип, начиная с .NET 6. Раньше был значимый тип	`""`
`object`	Базовый тип для всех остальных типов данных в C#	Ссылочный тип	`null`
`dynamic`	Тип данных, принимающий любой другой тип данных и позволяющий динамически изменять его. Не рекомендуется для большинства ситуаций	То же	`null`

Более подробную информацию о встроенных типах данных C# можно найти в официальной документации Microsoft по следующей ссылке: **https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/built-in-types**.

Далее мы рассмотрим различные типы элементов управления из числа поддерживаемых C# 11. Они позволяют выполнять код условно, создавать ветвления в потоке выполнения и выполнять повторяющиеся действия.

Поток управления

Поток управления обеспечивает возможность принятия решений в процессе выполнения кода программы. Некоторые элементы потока управления позволяют многократно выполнять заданное действие, в то время как другие — выполнять действие только при определенном условии.

В табл. 1.2 перечислены основные элементы потока управления, доступные в C#.

Таблица 1.2. *Элементы потока управления C#*

Тип оператора	Описание
`if .. else if .. else`	Указывает, какое условие должно быть применено к блоку кода, чтобы он был выполнен
`switch .. case`	Другой тип условной логики, когда из нескольких вариантов выбирается один
`for`	Циклическое выполнение оператора, который будет выполняться, пока произвольно заданный счетчик не достигнет определенного значения

Таблица 1.2 *(окончание)*

Тип оператора	Описание
`foreach`	Для каждого элемента коллекции выполняется определенный фрагмент логики
`while`	Блок кода, выполнение которого продолжается, пока условие остается истинным
`do .. while`	То же самое, что и `while`, но блок кода выполняется не менее одного раза

Для более подробного ознакомления с потоком управления в C# можно посетить соответствующий раздел официальной документации по этому языку по следующей ссылке: **https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/tutorials/branches-and-loops-local**.

Далее мы рассмотрим различные способы создания собственных пользовательских типов в C#.

Пользовательские типы в C#

Помимо использования встроенных типов данных, в программировании очень важно уметь создавать собственные объекты. И C# поддерживает ряд структур (табл. 1.3), позволяющих разработчикам ПО делать это.

Таблица 1.3. *Структуры C#, позволяющие создавать пользовательские типы данных*

Тип структуры	Описание
`interface`	Предоставляет только сигнатуры публичных членов (методов, полей и свойств), которые должен определить реализующий объект
`class`	Структура ссылочного типа, позволяющая создавать пользовательские типы данных с заданными полями, свойствами и методами
`struct`	Структура значимого типа, позволяющая создавать пользовательские типы данных с заданными полями, свойствами и методами
`record`	Класс или структура, специально разработанная для представления объектов базы данных и хранения данных. Поэтому этот объект, как правило, состоит только из свойств

Для получения более подробной информации о системе типов C# можно обратиться к официальной документации по следующей ссылке:
https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/.

Далее мы рассмотрим модификаторы доступа, предусмотренные в C#.

Модификаторы доступа

Модификаторы доступа — это ключевые слова, которые управляют видимостью объектов и элементов объектов, таких как поля, свойства и методы. Например,

можно сделать некоторые элементы объекта полностью недоступными для других объектов. Также можно сделать определенный объект видимым для других объектов внутри той же библиотеки, но совершенно недоступным для любого кода вне библиотеки.

В табл. 1.4 приведен полный список модификаторов доступа, предусмотренных в C#.

Таблица 1.4. *Модификаторы доступа в C#*

Модификатор доступа	Описание
`public`	Видимый для всех объектов внутри исполняемого файла и для всех объектов любого другого исполняемого файла, который на него ссылается (публичный)
`private`	Видимый только для членов одного объекта (приватный)
`protected`	Видимый только для членов того же объекта и членов любого производного объекта (защищенный)
`internal`	Видимый для всех объектов внутри исполняемого файла (внутренний)
`protected internal`	Видимый для всех объектов внутри исполняемого файла или производного объекта, созданного в другом исполняемом файле (защищенный внутренний)
`private protected`	Видимый только для членов того же объекта и членов производного объекта, если производный объект принадлежит тому же проекту (приватный защищенный)

Более подробную информацию о модификаторах доступа в C# можно найти здесь: **https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers**.

Конечно, в C# есть гораздо больше, чем мы успели охватить в этой главе. Но эти краткие описания охватывают значительную часть основ языка.

На этом мы завершаем введение в C# 11 и .NET 7. Подведем итоги тому, что мы узнали.

Заключение

В C# 11 и .NET 7 появились некоторые новые возможности. Но старые функции по-прежнему будут работать.

Если вы знакомы с предыдущей версией платформы и языка, то, возможно, заметили, что в структуре проекта почти ничего не изменилось. После создания нового проекта он выглядит практически так же, как выглядел бы в .NET 6.

И на это есть веские причины. Microsoft продолжает свою традицию вносить как можно меньше критических изменений. Одно из преимуществ этого — простота переноса старых приложений на новую платформу. Как правило, достаточно лишь изменить версию фреймворка в файле проекта.

Но у такого подхода есть и недостаток. Поскольку по мере развития языка прежние функции не помечаются как устаревшие, сложность языка продолжает расти. Поэтому для освоения C# 11 надо изучить гораздо больше, чем требовалось для C# 1.

Однако есть возможность обойти эту проблему. Изучая язык, сосредоточьтесь только на его новейших особенностях. И игнорируйте все остальное. Если затем, просматривая чужой код, вы случайно наткнетесь на какую-то старую функцию, то всегда сможете легко ее найти, если знаете основы языка, которые мы кратко повторили в этой главе.

В следующей главе вы узнаете о новых функциях, добавленных в версию 11 языка C#.

Что следует помнить

- Для работы с C# 11 и .NET 7 необходимо загрузить .NET 7 SDK.
- Для работы с кодом подойдет либо редактор кода, либо IDE.
- Подходящий редактор кода — Visual Studio Code.
- В зависимости от используемой ОС подходящими IDE являются Visual Studio 2022, Visual Studio 2022 for Mac или Rider.
- Одна из новых возможностей C# — автоматическая инициализация свойств структуры `struct` значениями по умолчанию для соответствующего типа данных.
- Еще одна новая возможность C# — новый тип строкового литерала, в котором допустимы любые специальные символы.

Контрольные вопросы

1. **Какой минимальный набор инструментов необходим для начала работы с C# 11 и .NET 7?**

   а) .NET 7 SDK;

   б) Редактор кода;

   в) IDE;

   г) Все перечисленное.

2. **Какие типы данных поддерживают автоматическую инициализацию своих свойств значениями по умолчанию?**

   а) class;

   б) struct;

   в) record;

   г) Все перечисленное;

   д) Ничего из перечисленного.

3. Какое утверждение лучше всего описывает новые строковые литералы C# 11?

а) Их нельзя сочетать с интерполированными строками, поскольку фигурные скобки будут восприниматься как символьные литералы;

б) Их нельзя сочетать с интерполированными строками, поскольку этот тип строк не поддерживает знак доллара в начале определения значения;

в) Их можно сочетать с интерполированными строками, если вы поместите знак доллара в начале определения их значений;

г) Ни одно из этих утверждений не соответствует действительности.

Ответы

1. а.

2. б.

3. в.

Основные термины

- **IDE** — интегрированная среда разработки, предоставляющая все необходимые инструменты для работы с кодом.
- **Редактор кода** — программа, позволяющая писать код, но для его выполнения, как правило, требуются внешние средства.
- **SDK** — комплект средств разработки программного обеспечения, позволяющий использовать определенные языки, платформы, среды и наборы библиотек для создания кода.
- **CLI** — интерфейс командной строки, позволяющий выполнять команды с помощью текстового терминала.
- **Модификатор доступа** — ключевое слово, управляющее видимостью объектов или членов этих объектов.

Глава 2
Обзор возможностей C# 11

Введение

В этой главе будут рассмотрены все новейшие возможности, добавленные в C# 11. Некоторые из них уже были доступны в прежних версиях языка, но для их реализации нам приходилось устанавливать определенные внешние зависимости. В то же время другие возможности являются совершенно новыми. Одни из нововведений представляют собой улучшения функций, которые были добавлены в язык в последних версиях, тогда как другие совершенно самостоятельные.

Какими бы ни были новые возможности, в этой главе мы рассмотрим все. Для каждой из них далее будут приведены примеры реализации и показаны преимущества.

Структура

В этой главе мы рассмотрим следующие темы:

- автоматическая инициализация структур;
- обобщенные атрибуты;
- сопоставление последовательности с образцом;
- новые операции со строками.

Цели

К концу этой главы вы научитесь использовать каждую из возможностей C#, добавленную в язык с версией 11. Для этого здесь приведены примеры их реализации, которые вы можете опробовать в собственном коде. Это поможет вам понять важность и преимущества каждой из новых функций.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из перечисленных компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Автоматическая инициализация структур

Мы начнем с рассмотрения автоматической инициализации свойств в типах данных `struct`. Эта тема уже затрагивалась в *главе 1*. Но поскольку мы хотим перечислить все новые функции C# 11 в одном месте, в этой главе мы изучим эту возможность более подробно.

Прежде чем приступить к рассмотрению новых возможностей языка C#, нам нужно создать консольное приложение .NET. Я дам проекту этого приложения имя `NewFeatures`.

Если вы работаете с IDE, то можете создать новый проект приложения через графический интерфейс, выбрав шаблон консольного приложения. Но для этого также подойдет интерфейс командной строки (CLI), который доступен в любой системе, где установлен .NET SDK. Чтобы сделать это через CLI, вы можете выполнить следующую команду в папке по вашему выбору:

```
dotnet new console -o NewFeatures
```

В результате будет создана папка NewFeatures с проектом консольного приложения внутри него. Теперь мы добавим структуру, которая позволит нам продемонстрировать функцию автоматической инициализации. Для этого создайте папку с именем AutoDefaults внутри папки проекта NewFeatures. Затем создайте внутри этой папки файл StructExample.cs и наполните его следующим содержимым:

```
namespace NewFeatures.AutoDefaults;
public struct StructExample
{
    public int Id { get; set; } // Автоматически присвоено значение 0
    public string Name { get; set; } // Автоматически присвоено значение ""
    public bool Active { get; set; } // Автоматически присвоено значение "ложь"
}
```

Здесь мы добавили комментарии к свойствам этой структуры, чтобы указать, какое значение присваивается каждому из этих свойств при автоматической инициализации. Итак, в общих чертах, когда мы создаем новый экземпляр структуры

StructExample, мы можем сразу прочитать эти свойства. Попытка их прочтения не приведет к выдаче исключения NullReferenceException. И это несмотря на то, что мы не присвоили этим свойствам никаких значений в явном виде.

Чтобы проверить это поведение, мы можем заменить содержимое файла Program.cs следующим кодом:

```
using NewFeatures.AutoDefaults;
Console.WriteLine("Testing auto-defaults.");
var testStruct = new StructExample();
Console.WriteLine($"""
Struct data is as follows:
Id: {testStruct.Id},
Name: {testStruct.Name},
Active: {testStruct.Active}
""");
```

Такой код мог бы привести к выдаче исключения NullReferenceException в предыдущих версиях C#, как только мы попытались бы прочитать любое из свойств переменной testStruct. Но если мы запустим наше приложение, используя последнюю версию .NET, то оно выдаст следующий результат, подтверждающий, что все свойства были автоматически инициализированы путем присвоения им значений по умолчанию.

```
Testing auto-defaults.

Struct data is as follows:

Id: 0,
Name: ,
Active: False
```

Рис. 2.1. Свойства автоматически инициализированной структуры

Следующая особенность C# 11, которую мы рассмотрим, — это классы обобщенных атрибутов.

Обобщенные атрибуты

В C# классы атрибутов служат для расширения функциональности классов и членов классов и для добавления к ним метаданных. Например, помещение атрибута Authorize над методом в контроллере Web API гарантирует, что конечная точка, которую представляет этот метод, будет доступна только пользователям, прошедшим аутентификацию. А передача в этот атрибут дополнительных параметров приведет к применению к конечной точке еще более жестких требований авторизации.

Все классы атрибутов наследуют от класса System.Attribute. По принятым соглашениям имя класса атрибутов должно заканчиваться суффиксом Attribute, который можно опускать при указании атрибута в тексте программы. Например, упомянутый ранее атрибут Authorize в его источнике определен как AuthorizeAttribute.

Атрибут помещается перед классом или членом класса в квадратных скобках. А синтаксис присоединения атрибутов к классам и членам классов выглядит следующим образом:

```
[<имя класса атрибута без суффикса Attribute><необязательно: свойства атрибута в скобках>]
```

Как и любой другой класс, атрибуты могут иметь методы, поля и свойства. Но до версии C# 11 обобщенные типы для класса атрибута не поддерживались. Поэтому, если необходимо передать в класс атрибута произвольный тип, то единственный способ сделать это — передать его в конструктор атрибута и сохранить в какомлибо поле или свойстве. Но теперь в язык добавлены обобщенные атрибуты. И это значительно упростило ситуацию.

Но зачем вообще передавать определение произвольного типа в класс атрибута? Есть много сценариев, в которых такая возможность может оказаться полезной. Например, вы можете захотеть добавить в атрибут поведение, характерное для типа возвращаемых данных члена класса, над которым размещен атрибут. А для того, чтобы этот атрибут можно было применить в любом месте, этот тип должен быть внедряемым во время выполнения.

Пример обобщенного атрибута

Чтобы продемонстрировать работу обобщенных атрибутов, мы сравним их со старым способом передачи определений произвольных типов в классы атрибутов. Это поможет более наглядно продемонстрировать выгоду от использования обобщенного атрибута.

Для начала добавим в наш проект папку GenericAttributes и поместим в нее файл OldTypeAttribute.cs со следующим содержимым:

```
namespace NewFeatures.GenericAttributes;
public class OldTypeAttribute : Attribute
{
    public OldTypeAttribute(Type attributeType) => AttributeType = attributeType;
    public Type AttributeType { get; }
}
```

Как видно, в этом классе у нас есть свойство, доступное только для чтения, которое содержит тип. Оно задается при инициализации класса через конструктор класса. И чтобы узнать, какой тип имеет данный атрибут, нужно посмотреть на это свойство после инициализации атрибута.

Теперь добавим новый обобщенный атрибут. Для этого создадим в той же папке файл NewTypeAttribute.cs и наполним его следующим содержимым:

```
namespace NewFeatures.GenericAttributes;
public class NewTypeAttribute<T> : Attribute { }
```

Как видите, больше нет необходимости в дополнительном члене класса для хранения типа атрибута. Это также облегчает получение типа во время выполнения. Кроме того, улучшается синтаксис при применении атрибута.

Чтобы продемонстрировать, как применяются оба этих атрибута, добавим в ту же папку файл ParametrizedClass.cs. Он будет содержать следующий код:

```
namespace NewFeatures.GenericAttributes;
public class ParametrizedClass
{
    [OldType(typeof(int))]
    public int DoOldStyleJob() => default;
    [NewType<int>]
    public int DoNewStyleJob() => default;
}
```

Как видно, синтаксис обобщенного атрибута чище. Одним ключевым словом стало меньше. Но еще лучше то, что тип, который хранит атрибут, можно мгновенно получить посредством отражений, без обращения к его членам. Чтобы продемонстрировать это, добавим еще некоторый код в наш файл Program.cs. Во-первых, добавим в начало кода в файле оператор `using`:

```
using NewFeatures.GenericAttributes;
```

Затем добавим в конец файла следующий фрагмент:

```
Console.WriteLine("Testing Generic Attributes.");
var methods = typeof(ParametrizedClass).GetMethods();
foreach (var method in methods)
{
    var attribute = method?
        .GetCustomAttributes(false)
        .FirstOrDefault();
    if (attribute != null)
        Console.WriteLine($"""
        Method name: {method?.Name},
        Method attribute: {attribute.GetType()} """);
}
```

Результат выполнения этого кода показан на рис. 2.2.

Имея обобщенный атрибут, мы сразу видим, какой тип данных он представляет. Однако атрибут старого стиля не показывает эту информацию, когда мы использу-

```
Testing Generic Attributes.

Method name: DoOldStyleJob,
Method attribute: NewFeatures.GenericAttributes.OldTypeAttribute

Method name: DoNewStyleJob,
Method attribute: NewFeatures.GenericAttributes.NewTypeAttribute`1[System.Int32]

Method name: GetType,
Method attribute: System.Runtime.CompilerServices.NullableContextAttribute
```

Рис. 2.2. Классы обобщенных и необобщенных атрибутов

ем отражения для получения определения его экземпляра. Таким образом, обобщенный атрибут не только проще задать, но и легче прочитать.

Далее мы рассмотрим некоторые усовершенствования функции сопоставления последовательности с образцом, которая была добавлена в C# в его достаточно поздних версиях.

Сопоставление последовательности с образцом

Функция сопоставления последовательности с образцом в C# позволяет определить, соответствует ли определенное значение заданному шаблону. А в C# 11 эта функция была применена и к коллекциям.

Существуют различные образцы, с которыми можно сопоставлять коллекции. Можно проверять точное совпадение, когда последовательность, с которой сверяется коллекция, должна быть идентична последовательности внутри коллекции. Но также можно проверять и неточное совпадение, когда определенные места в последовательности могут совпадать с любым значением или должны попадать в заданный диапазон.

При сопоставлении последовательности с образцом полностью игнорируются типы коллекций. Уделяется внимание только последовательности в коллекциях. Неважно, работаете ли вы с массивами, списками или с чем-то еще. Если коллекция содержит определенную последовательность значений, то результат сопоставления будет положительным, если она соответствует последовательности, заданной в качестве шаблона.

Синтаксис сопоставления последовательности с образцом выглядит следующим образом:

```
<переменная коллекции> is [<последовательность>]
```

Чтобы проверить, содержит ли коллекция, с которой сопоставляется последовательность, точно такой же набор значений, следует поместить полный список точных значений, разделенных запятыми, в квадратные скобки в том порядке, в котором они должны появиться в коллекции. Чтобы проверить, содержит ли коллекция определенные значения в отдельных позициях, в то время как совпадения с другими значениями вам безразличны, можно просто указать символ отбрасывания (подчеркивание, _) в позициях значений, которые вам не важны. Если вы ищете определенную последовательность значений, и при этом вам не важно, в каком именно месте коллекции она находится, можно использовать две точки для представления диапазона любых значений. И, конечно же, все эти способы можно комбинировать для выполнения более сложного сопоставления последовательности с образцом.

Демонстрация сопоставления последовательности с образцом

Чтобы продемонстрировать, как работает сопоставление последовательности с образцом, создадим внутри нашего проекта папку PatternMatching. Затем добавим в нее файл Collection Operations.cs и наполним его следующим содержимым:

```
namespace NewFeatures.PatternMatching;
public class CollectionOperations
{
    private List<int> items = new List<int>
    {
        2, 3, 6, 7, 8
    };
    public bool MatchExactSequence => items is [2, 3, 6, 7, 8];
    public bool MatchWithDiscard => items is [2, _, 6, _, 8];
    public bool MatchWithRange => items is [2, ..];
}
```

В этом классе имеются следующие три поля:

1. `MatchExactSequence` — осуществляет поиск точного совпадения. Это означает, что коллекция, с которой сравнивается последовательность, должна иметь точно такое же содержимое.
2. Метод `MatchWithDiscard` — проверяет только наличие в коллекции определенного количества элементов и соответствие элементов с индексами 0, 2 и 4 определенным значениям. При этом неважно, какие значения имеют элементы с индексами 1 и 3.
3. Метод `MatchWithRange` — проверяет только равенство первого элемента коллекции 2. При этом неважно, сколько еще элементов содержит коллекция и что они собой представляют.

Но хорошая новость заключается в том, что эта новая функция сопоставления последовательности с образцом работает не только с коллекциями классических типов, но и с недавно добавленным типом `char span`.

Сопоставление последовательности с образцом типа *char span*

Тип данных `span` был добавлен в восьмую версию языка C#. Это тип коллекции, который предназначен для представления определенного участка другой коллекции. Например, если вы хотите изменять только определенный набор элементов внутри некоторого массива, то можете присвоить этот набор элементов объекту типа `span`. Таким образом, вы будете защищены от случайного изменения тех членов исходной коллекции, которые вы не хотели изменять.

В C# 11 можно выполнять сопоставление последовательности с образцом типа `span`. Чтобы продемонстрировать, как это работает, добавим файл CharSpanOperations.cs

в папку PatternMatching. В этом файле мы прежде всего создадим следующее определение класса:

```
namespace NewFeatures.PatternMatching;
public class CharSpanOperations
{
    private char[] charArray = new char[3]
    {
        'a',
        'b',
        'c'
    };
    private Span<char> GetSpan()
    {
        Span<char> span = charArray;
        return span;
    }
    private ReadOnlySpan<char> GetReadonlySpan()
    {
        ReadOnlySpan<char> span = charArray;
        return span;
    }
}
```

Затем добавим к нему следующие открытые свойства:

```
public bool MatchWholeSpan => GetSpan() is ['a', 'b', 'c'];
public bool MatchWholeReadOnlySpan => GetReadonlySpan() is ['a', 'b', 'c'];
public bool MatchWholeSpanWithDiscard => GetSpan() is ['a', _, 'c'];
public bool MatchWholeReadOnlySpanWithDiscard => GetReadonlySpan() is ['a', _, 'c'];
public bool MatchWholeSpanByRange => GetSpan() is ['a', ..];
public bool MatchWholeReadOnlySpanBtRange => GetReadonlySpan() is ['a', ..];
```

Теперь мы можем проверить, работают ли все наши примеры сопоставления последовательности с образцом. Для этого добавим в наш файл Program.cs оператор `using`:

```
using NewFeatures.PatternMatching;
```

Затем добавим в этот файл следующий код:

```
Console.WriteLine("Testing Pattern Matching.");
var collectionOperations = new CollectionOperations();
Console.WriteLine($"""
MatchExactSequence returns {collectionOperations.MatchExactSequence},
MatchWithDiscard returns {collectionOperations.MatchWithDiscard},
MatchWithRange returns {collectionOperations.MatchWithRange}
""");
var charSpanOperations = new CharSpanOperations();
Console.WriteLine($"""
MatchWholeSpan returns {charSpanOperations.MatchWholeSpan},
```

```
MatchWholeReadOnlySpan returns {charSpanOperations.MatchWholeReadOnlySpan},
MatchWholeSpanWithDiscard returns {charSpanOperations.MatchWholeSpanWithDiscard},
MatchWholeReadOnlySpanWithDiscard returns
                                    {charSpanOperations.MatchWholeReadOnlySpanWithDiscard},
MatchWholeSpanByRange returns {charSpanOperations.MatchWholeSpanByRange},
MatchWholeReadOnlySpanBtRange returns {charSpanOperations.MatchWholeReadOnlySpanBtRange}
""");
```

Если сопоставление последовательности с образцом работает так, как мы ожидаем, то каждое свойство с модификатором `public` во вновь созданных классах должно возвращать значение "истина" (`True`). И, как следует из консольного вывода, показанного на рис. 2.3, происходит именно это.

```
Testing Pattern Matching.

MatchExactSequence returns True,
MatchWithDiscard returns True,
MatchWithRange returns True

MatchWholeSpan returns True,
MatchWholeReadOnlySpan returns True,
MatchWholeSpanWithDiscard returns True,
MatchWholeReadOnlySpanWithDiscard returns True,
MatchWholeSpanByRange returns True,
MatchWholeReadOnlySpanBtRange returns True
```

Рис. 2.3. Демонстрация сопоставления последовательности с образцом

Как видите, в C# 11 существенно расширены возможности языка по операциям сопоставления последовательности с образцом. Но это также дало программистам новые возможности для работы со строками. Их мы и рассмотрим далее.

Новые операции со строками

В C# 11 появились новые возможности работы со строками, которые делают ваш код более компактным и удобным для чтения. Одно из новых улучшений — возможность использования нескольких строк в интерполированной строке. Это особенно полезно, когда требуется вставить в строку результаты какого-либо сложного выражения. Еще одна новая функция — необрабатываемый строковый литерал, о котором мы вкратце упоминали в *главе 1*.

Начнем с применения многострочной интерполяции строк. В качестве кода для вставки в строку мы возьмем многозвенное выражение языка интегрированных запросов (LINQ). И чтобы продемонстрировать преимущества этой функции, мы осуществим интерполяцию дважды. В первом случае мы воспользуемся однострочной интерполяцией строки в старом стиле. Затем реализуем многострочную интерполяцию строки в новом стиле.

Создадим папку StringOperations внутри папки нашего проекта и добавим в нее файл InterpolatedStrings.cs со следующим содержимым:

```
namespace NewFeatures.StringOperations;
public class InterpolatedStrings
{
    private List<(string, int)> names = new List<(string, int)>
    {
        ("Джон", 25),
        ("Майк", 34),
        ("Лоуренс", 42)
    };
    public string OldStyleInterpolation => $"The age of Laurence is:
         {names.Where(n => n.Item1 == "Laurence").Select(n => n.Item2).FirstOrDefault()}.";
    public string NewStyleInterpolation => $" The age of Laurence is: {
        names
            .Where(n => n.Item1 == "Laurence")
            .Select(n => n.Item2)
            .FirstOrDefault()
        }.";
}
```

Итак, у нас есть два строковых свойства: `OldStyleInterpolation` и `NewStyleInterpolation`. Оба доступны только для чтения и возвращают одинаковые значения. Единственное различие между ними заключается в форматировании кода.

Взглянув на этот код, можно сразу понять, чем так полезна многострочная интерполяция строк. Если посмотреть на свойство `OldStyleInterpolation`, то длинное LINQ-выражение размещено на одной строке, что затрудняет его чтение. Действительно, выражение настолько длинное, что не помещается по ширине страницы.

В свойстве `NewStyleInterpolation`, напротив, одно LINQ-выражение разбито на несколько строк. Это значительно облегчает чтение кода. И каждая строка выражения аккуратно укладывается в ширину страницы.

Далее мы рассмотрим необрабатываемый строковый литерал. Для этого создадим еще один файл в подпапке StringOperations и назовем его StringLiterals.cs. Содержимое этого файла будет следующим:

```
namespace NewFeatures.StringOperations;
public class StringLiterals
{
    public string RawStringLiteral => """
    This text may contain any symbols, including
    newlines, "quoted text",
        indentations, and so on.
    There is no need to escape any characters.
    """;
    public string NewStyleInterpolation => $"""
    This is a combination of a new string literal and interpolated string.
    This is a value from inserted code: {5 + 8}.
    """;
}
```

Таким образом, мы имеем новый необрабатываемый строковый литерал, применяемый в двух вариантах: самостоятельно и в сочетании со строковой интерполяцией. Синтаксис этого строкового литерала предусматривает окружение текстового значения с обеих сторон тремя двойными кавычками. Причем и начальная, и конечная последовательность из трех двойных кавычек должна располагаться на отдельной строке.

Все, что находится между ними, будет интерпретироваться как текст, даже если используемый там символ имеет в C# специальное значение. Вам не придется прибегать к экранированию специальных символов. При этом символ обратной косой черты (\), который обычно служит для экранирования специальных символов, также будет интерпретироваться в тексте как обычный символ. Также будет и с двойными кавычками до тех пор, пока мы не дойдем до новой строки, где имеется последовательность из трех двойных кавычек. Так мы узнаем, что дошли до конца текста.

Если перед открывающей последовательностью из трех двойных кавычек поставить знак доллара, то в строку с помощью фигурных скобок можно будет вставить код, как это делается при любом другом типе интерполяции строки. И это единственная ситуация, когда необходимо экранировать фигурные скобки, если вы хотите включить их как часть текста, а не использовать для интерполяции строки.

Теперь мы можем протестировать все добавленные нами строковые функции. Для этого мы поместим в файл Program.cs некоторый код. И первое, что мы сделаем, — добавим в начало файла следующий оператор:

```
using NewFeatures.StringOperations;
```

Затем мы протестируем нашу функцию многострочной интерполяции строк и посмотрим, дает ли она тот же результат, что и однострочная версия. Для этого добавим в файл следующий код:

```
Console.WriteLine("Testing String Operations.");
var interpolatedStrings = new InterpolatedStrings();
Console.WriteLine($"""
Old style interpolation:
{interpolatedStrings.OldStyleInterpolation}
New style interpolation:
{interpolatedStrings.NewStyleInterpolation}
""");
```

На рис. 2.4 показан полученный результат, который действительно подтверждает, что многострочная интерполяция строк работает так, как ожидалось.

Далее мы протестируем наши строковые литералы, чтобы убедиться в том, что они дают те результаты, которые мы ожидаем. Для этого добавим в наш файл Program.cs следующий код:

```
var stringLiterals = new StringLiterals();
Console.WriteLine($"""
Raw string literal:
{stringLiterals.RawStringLiteral}
```

```
String literal with interpolation:
{stringLiterals.NewStyleInterpolation}
""");
```

Результаты выполнения этого кода, показанные на рис. 2.5, подтверждают, что новые строковые литералы работают корректно как с интерполяцией, так и без нее.

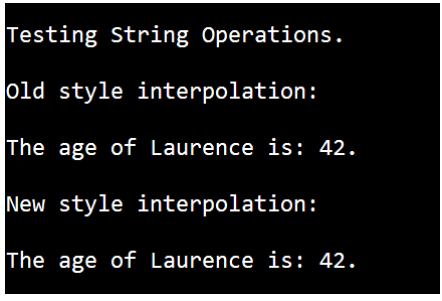


Рис. 2.4. Тестирование многострочной интерполяции строк

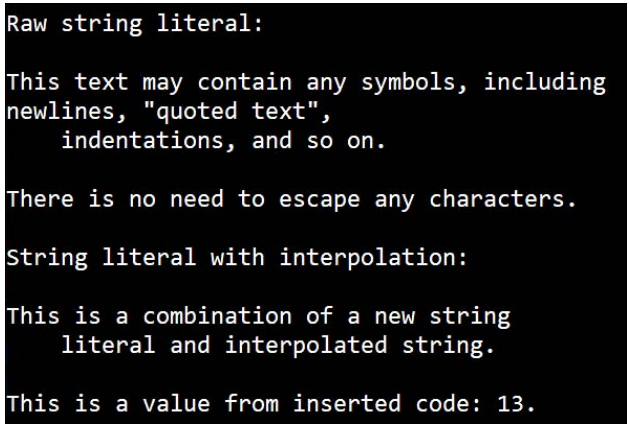


Рис. 2.5. Тестирование строковых литералов

Здесь мы видим очевидные преимущества строковых литералов нового стиля, когда используем их для вывода на консоль. Нам больше не нужны многократные вызовы метода `Console.WriteLine`. Теперь достаточно один раз вызвать метод и добавить несколько строк в сам текст.

На этом обзор новых возможностей C# 11 завершен. Подведем итоги тому, что мы узнали.

Заключение

В C# 11 добавлено множество полезных функций, которые облегчают чтение, написание и выполнение кода. Одна из таких возможностей — автоматическое присвоение значений по умолчанию не допускающим неопределенности полям

в структурах. Это позволяет писать компактный код и предотвращает случайные ошибки, если какое-либо из свойств объекта `struct` не было задано явно.

Еще одна полезная особенность — классы обобщенных атрибутов — позволяет легко определить, какой именно тип связан с атрибутом. Нам больше не нужно хранить тип в одном из членов класса атрибута. Это также делает наш код компактным и более удобным в использовании.

Кроме того, в C# 11 появилась новая возможность сопоставления последовательности с образцом. Эта мощная функция предоставляет множество способов применения различных типов коллекций в условной логике. Функция работает с коллекциями любого типа, включая `Span` и `ReadOnlySpan`.

В C# 11 также существенно улучшено использование строк. Во-первых, появилась возможность комбинировать несколько строк в интерполированных строках. Это позволяет отформатировать код таким образом, что его становится гораздо легче читать. Во-вторых, появился новый тип строкового литерала, который не требует от программистов вставки экранирующих символов. Причем этот новый строковый литерал может работать как с интерполяцией строк, так и без нее.

В следующей главе мы рассмотрим возможности, которые были добавлены в .NET 7, включая базовый SDK, интерфейс командной строки и основные библиотеки.

Что следует помнить

- Автоматическая инициализация свойств гарантирует, что не допускающим неопределенности свойствам типа данных `struct` будут присвоены некоторые значения.
- Классы обобщенных атрибутов позволяют добавлять в атрибут любой произвольный тип.
- С помощью функции сопоставления последовательности с образцом можно выполнять сопоставление коллекций.
- Многострочная строковая интерполяция дает программистам возможность использовать многострочные выражения внутри интерполируемых строк.
- Новый строковый литерал позволяет составлять строковые литералы, не требующие экранирования специальных символов.

Контрольные вопросы

1. Чем полезна автоматическая инициализация свойств?

а) Более компактный код;

б) Предотвращение появления случайных ошибок;

в) Не нужно использовать конструктор;

г) Все перечисленное.

2. **Чем полезны обобщенные атрибуты?**

   а) Легче определить тип, связанный с ними;

   б) Нет необходимости хранить тип в отдельном члене класса;

   в) Оба перечисленных;

   г) Ничего из перечисленного.

3. **Какие символы необходимо экранировать при использовании необрабатываемых строковых литералов без интерполяции?**

   а) Фигурные скобки;

   б) Символ новой строки;

   в) Двойные кавычки;

   г) Ничего из перечисленного.

4. **Какие символы необходимо экранировать при использовании необрабатываемых строковых литералов с интерполяцией?**

   а) Фигурные скобки;

   б) Символ новой строки;

   в) Двойные кавычки;

   г) Ничего из перечисленного.

Ответы

1. г.

2. в.

3. г.

4. а.

Основные термины

- **Автоматически инициализируемые свойства** — не допускающие неопределенных значений свойства, которым присваиваются значения по умолчанию, соответствующие их типу данных, если они не заданы явно.
- **Класс обобщенного атрибута** — класс атрибута, который использует обобщенный синтаксис и может быть связан с любым произвольным типом.
- **Необрабатываемый строковый литерал** — строковый литерал, окруженный последовательностями из трех двойных кавычек. Ни один из символов внутри такой строки не требует экранирования.

Глава 3
Что нового в .NET 7?

Введение

Мы рассмотрели основные изменения в языке C# 11. Однако платформа .NET 7 — это нечто большее, чем C#. Некоторые интересные новшества появились и в самой платформе.

Говоря упрощенно, .NET состоит из большого количества компонентов. К ним относятся средства сборки, генерации и анализа кода, компиляторы и т. д. Она также включает в себя интерфейс командной строки (CLI) и различные основные библиотеки, которые могут быть использованы любым .NET-совместимым языком программирования.

Как видите, платформа .NET — это достаточно обширная тема. И эта глава призвана охватить основные усовершенствования, которые были внесены во все ее части.

Структура

В этой главе мы рассмотрим новые возможности .NET 7, которые сведены в следующие темы:

- улучшения в SDK и инструментах сборки;
- обновления библиотек System и Microsoft;
- улучшения наблюдаемости;
- критические изменения в .NET 7.

Цели

Прочитав эту главу, вы познакомитесь с новыми функциями, добавленными в версию 7 платформы .NET. Вы также узнаете, какие из прежних функций были помечены как устаревшие, удалены или изменили свое поведение. Это поможет вам перенести свою кодовую базу на .NET 7 без нарушения работы существующего кода.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из перечисленных компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Улучшения в SDK и инструментах сборки

Мы начнем с рассмотрения наиболее фундаментальных компонентов .NET — SDK и поставляемых с ним средств сборки. Сюда относится и интерфейс командной строки (CLI).

В .NET 7 было внесено много улучшений, связанных исключительно с производительностью. Но есть и функциональные изменения, которые легко продемонстрировать. Именно этим мы сейчас и займемся, начав с улучшений CLI.

Улучшения инструментов командной строки

Одно из основных улучшений инструментов командной строки .NET — вывод команды `dotnet --help`. Он стал гораздо более подробным, чем раньше. Результат выполнения этой команды будет выглядеть следующим образом:

```
Использование: dotnet [runtime-options] [path-to-application] [arguments]
Запуск приложения .NET.
runtime-options:
  --additionalprobingpath <path>   Путь к политике проверки и проверяемым сборкам.
  --additional-deps <path>         Путь к дополнительному файлу deps.json.
  --depsfile                       Путь к файлу <приложение>.deps.json.
  --fx-version <version>           Версия установленной общей платформы, которую следует
                                   использовать для запуска приложения.
  --roll-forward <setting>         Накат до версии платформы (LatestPatch, Minor,
                                   LatestMinor,  Major, LatestMajor, Disable).
  --runtimeconfig                  Путь к файлу <приложение>.runtimeconfig.json.
path-to-application:
  Путь к DLL-файлу приложения, который необходимо выполнить.
Использование: dotnet [sdk-options] [command] [command-options] [arguments]
Выполнение команды пакета SDK для .NET.
sdk-options:
  -d|--diagnostics  Включение выходных данных диагностики.
  -h|--help         Показать справку командной строки.
```

```
  --info             Отображение сведений о .NET.
  --list-runtimes    Отображение установленных сред выполнения.
  --list-sdks        Отображение установленных пакетов SDK.
  --version          Отображение используемой версии пакета SDK для .NET.
Команды SDK:
  add                Добавление пакета или ссылки в проект .NET.
  build              Сборка проекта .NET.
  build-server       Взаимодействие с серверами, запущенными в ходе сборки.
  clean              Очистка выходных данных сборки проекта .NET.
  format             Применение настроек стилей к проекту или решению.
  help               Показать справку командной строки.
  list               Вывод списка ссылок на проекты в проекте .NET.
  msbuild            Выполнение команд Microsoft Build Engine (MSBuild).
  new                Создание нового файла или проекта .NET.
  nuget              Предоставление дополнительных команд NuGet.
  pack               Создание пакета NuGet.
  publish            Публикация проекта .NET для развертывания.
  remove             Удаление пакета или ссылки из проекта .NET.
  restore            Восстановление зависимостей, указанных в проекте .NET.
  run                Сборка и запуск проекта .NET.
  sdk                Управление установкой пакета SDK .NET.
  sln                Изменение файлов решения Visual Studio.
  store              Сохранение указанных сборок в хранилище пакетов среды выполнения.
  test               Запуск модульных тестов с помощью средства, указанного в проекте .NET.
  tool               Установка и настройка инструментов, расширяющих возможности .NET.
  vstest             Выполнение команд Microsoft Test Engine (VSTest).
  workload           Управление необязательными рабочими нагрузками.
Дополнительные команды из входящих в пакет инструментов:
  dev-certs          Создание сертификатов разработки и управление ими.
  fsi                Запуск F# Interactive / выполнение скриптов F#.
  user-jwts          Управлять JSON Web Token в разработке.
  user-secrets       Управление развертыванием секретов пользователей.
  watch              Запуск наблюдателя за файлами, который выполняет команду при изменении
                     файлов.
Для получения дополнительных сведений о команде выполните команду "dotnet [команда] --help".
```

Но это не единственное улучшение интерфейса командной строки .NET. Также реализовано дополнение ввода нажатием клавиши табуляции. Нажатие клавиши <Tab> после частичного ввода названия атрибута команды доводит ввод до конца. Таким образом, интерфейс командной строки стал совместимым с другими CLI.

NativeAOT и функция обрезки библиотек

Опережающая компиляция (AOT, Ahead of Time) — это механизм компиляции кода в двоичные файлы для конкретной аппаратной платформы. Между приложением и компьютером, на котором оно выполняется, больше не нужен промежуточный код. Приложение предварительно компилируется для работы на компьютере

с определенной операционной системой и архитектурой процессора. Благодаря отсутствию промежуточного кода, который необходимо интерпретировать в процессе выполнения, такие предварительно скомпилированные приложения работают значительно быстрее тех, что обращаются к платформе .NET во время выполнения. И NativeAOT — это функция .NET 7, обеспечивающая подобную компиляцию в .NET.

Поскольку приложение было предварительно скомпилировано для работы на конкретном типе компьютера, оно больше не зависит от компонентов .NET. Но есть и такие .NET-компоненты, которые не позволяют скомпилировать код в полностью нативное приложение. Чтобы ваше .NET-приложение работало с NativeAOT, его необходимо "обрезать"[1].

К счастью, в шаблоны проектов .NET 7 была добавлена функция, упрощающая процесс "обрезки" библиотек. Все, что нам нужно сделать, это вставить в код разметки файла проекта следующую запись:

```
<IsTrimmable>true</IsTrimmable>
```

Полное содержимое файла *.csproj может выглядеть следующим образом:

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net7.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <LangVersion>preview</LangVersion>
        <Nullable>enable</Nullable>
        <IsTrimmable>true</IsTrimmable>
    </PropertyGroup>
</Project>
```

Центральный диспетчер пакетов

Центральный диспетчер пакетов — это новый компонент, который был добавлен в систему управления пакетами NuGet. Он позволяет легко управлять общими зависимостями для нескольких проектов в сложных решениях.

Чтобы включить эту функцию, необходимо создать в решении файл Directory.Packages.props, который будет содержать XML-код с корневым элементом `Project`. Для включения центрального диспетчера пакетов внутри этого элемента необходимо добавить следующий код разметки:

```
<PropertyGroup>
<ManagePackageVersionsCentrally>true</ManagePackageVersionsCentrally>
</PropertyGroup>
```

[1] Избавить от несовместимых компонентов библиотек. — *Пер.*

Затем можно создать элемент `ItemGroup`, который будет содержать ссылки на каждый отдельный пакет, который вы хотите использовать в своих проектах. Пакет будет представлен элементом `PackageVersion`, который добавляется в следующем формате:

```
<PackageVersion Include="{ имя пакета NuGet }" Version="{ версия пакета NuGet }" />
```

Это будет единственная ссылка на конкретный пакет, которая понадобится во всем решении. Чтобы применить его в своем проекте, следует лишь добавить в любой произвольный элемент `ItemGroup` в файле *.csproj следующий элемент:

```
<PackageReference Include="{ имя пакета NuGet }" />
```

На этом мы завершаем обзор улучшений SDK и инструментов сборки. Далее мы перейдем к рассмотрению улучшений, которые были внесены в основные библиотеки .NET.

Обновления библиотек System и Microsoft

Теперь мы рассмотрим некоторые усовершенствования, добавленные в основные библиотеки, используемые в .NET, — в первую очередь это относится к пространствам имен `System` и `Microsoft`. Для демонстрации новых возможностей создадим проект консольного приложения. Назовем наш проект `CoreLibraryImprovements`. Создать его можно либо через графический интерфейс выбранной вами среды разработки, либо выполнив следующую команду внутри любой папки на вашем компьютере:

```
dotnet new console -o CoreLibraryImprovements
```

В результате будет создана папка CoreLibraryImprovements со структурой проекта. После этого мы начнем добавлять в проект классы для демонстрации новых возможностей базовых библиотек. Первая возможность, которую мы рассмотрим, — это улучшение связанных со временем типов данных в библиотеке `System`.

Поддержка микросекунд и наносекунд

До появления .NET 7 миллисекунда была наименьшей единицей времени, которая поддерживалась в .NET. Но теперь вы можете работать с микросекундами и наносекундами. Обе эти единицы измерения были добавлены в объекты `DateTime`, `DateTimeOffset`, `TimeOnly` и `TimeSpan`.

Для демонстрации применения этих единиц выполните следующие действия:

1. Добавим файл TimeDatatypeImprovements.cs в папку нашего проекта CoreLibrary Improvements. Первоначальное содержимое этого файла будет следующим:

```
namespace CoreLibraryImprovements;
public class TimeDatatypeImprovements
{
    public static void DemoNewTimeFeatures()
    {
    }
}
```

2. В методе `DemoNewTimeFeatures` разместим для каждого типа данных код, в который добавлены микросекунды и наносекунды. Сначала мы вставим в метод следующий код для демонстрации нового API `DateTime`:

```
var dateTime = new DateTime(2022, 3, 2, 15, 00, 30, 30, 30);
Console.WriteLine($"""
    DateTime object is {dateTime} with {dateTime.Microsecond}
                       microseconds and {dateTime.Nanosecond} nanoseconds.
    """);
```

3. В этом коде зададим значения микросекунд и наносекунд при создании нового экземпляра объекта `DateTime`.
4. Затем выведем эти значения на консоль.
5. Выполним аналогичные действия для объекта `DateTimeOffset`, добавив в метод следующий код:

```
var dateTimeOffset = new DateTimeOffset(2022, 3, 2, 15, 00, 30, 30, 30,
                     TimeSpan.FromMicroseconds(60 * 1000 * 1000));
Console.WriteLine($"""
    DateTimeOffset object is {dateTimeOffset } with {dateTimeOffset.Microsecond}
     microseconds and {dateTimeOffset.Nanosecond} nanoseconds.
""");
```

6. Далее вставим следующий код, демонстрирующий работу нового API на объекте типа `TimeOnly`:

```
var timeOnly = new TimeOnly(15, 00, 30, 30, 30);
Console.WriteLine($"""
    TimeOnly object is {timeOnly} with {timeOnly.Microsecond}
     microseconds and {timeOnly.Nanosecond} nanoseconds.
    """);
```

7. Как вы, вероятно, заметили, объект `TimeOnly` очень похож на `DateTime`, но в нем отсутствует часть с датой.

   Наконец, добавим следующий код для демонстрации нового API для данных типа `TimeOffset`:

```
var timeSpan = new TimeSpan(19, 3, 40, 20, 30, 30);
Console.WriteLine($"""
    TimeSpan object is {timeSpan} with {timeSpan.Microseconds}
    microseconds and {timeSpan.Nanoseconds} nanoseconds.
    Ticks per microsecond: {TimeSpan.TicksPerMicrosecond}.
    Nanoseconds per tick: {TimeSpan.NanosecondsPerTick}.
    """);
```

8. Как вы могли заметить, этот тип данных имеет некоторые дополнительные статические поля, а именно — `TicksPerMicrosecond` и `NanosecondsPerTick`.
9. Теперь, чтобы увидеть наш код в действии, мы можем заменить содержимое файла `Program.cs` в папке проекта следующим кодом:

```
using CoreLibraryImprovements;
Console.WriteLine("Demonstrating time-related data type improvements.");
TimeDatatypeImprovements.DemoNewTimeFeatures();
```

И если мы теперь запустим приложение, то результат должен выглядеть примерно так, как показано на рис. 3.1.

```
Demonstrating time-related data type improvements.
DateTime object is 02/03/2022 15:00:30 with 30 microseconds and 0 nanoseconds.

DateTimeOffset object is 02/03/2022 15:00:30 with 30 microseconds and 0 nanoseconds.

TimeOnly object is 15:00 with 30 microseconds and 0 nanoseconds.

TimeSpan object is 19.03:40:20.0300300 with 30 microseconds and 0 nanoseconds.
Ticks per microsecond: 10.
Nanoseconds per tick: 100.
```

Рис. 3.1. Демонстрация новых API, связанных со временем

На этом обзор новых функций .NET, связанных со временем, завершен. Далее мы рассмотрим новые возможности, касающиеся обработки JSON.

Новые возможности для работы с JSON

В .NET 7 добавлены следующие усовершенствования для работы с JSON:

- в класс `JsonWriterOptions` из библиотеки `System.Text.Json` добавлено свойство `MaxDepth`;
- заданный по умолчанию объект `JsonSerializerOptions` стал доступен снаружи, чтобы разработчики могли видеть его свойства;
- реализованы вызовы `HTTP PATCH` для JSON;
- добавлены функции полиморфизма объектов JSON.

Чтобы продемонстрировать эти возможности JSON, добавим в папку проекта файл NewJsonFeatures.cs и для начала наполним его следующим содержимым:

```
using System.Net.Http.Json;
using System.Text.Json;
using System.Text.Json.Serialization;
namespace CoreLibraryImprovements;
public static class NewJsonFeatures
{
}
```

Затем добавим в этот класс метод, который продемонстрирует, как использовать свойство `MaxDepth` класса `JsonWriterOptions`.

Свойство *MaxDepth* класса *JsonWriterOptions*

Свойство `MaxDepth` устанавливает максимальную глубину внутренних объектов внутри вывода JSON. Если это число превышено и структура объекта глубже, то выдается исключение. Метод, который мы добавим для демонстрации этого, будет выглядеть следующим образом:

```
public static void DemoJsonWriterOptions()
{
    var options = new JsonWriterOptions
    {
        Indented = true,
        MaxDepth = 5
    };
    using var fileStream = File.Create("output.json");
    using var writer = new Utf8JsonWriter(fileStream, options: options);
    using JsonDocument document = JsonDocument.Parse("""
    {
        "level1": {
            "level2": {
                "level3": {
                    "key": "value"
                }
            }
        }
    }
    """);
    var root = document.RootElement;
    if (root.ValueKind == JsonValueKind.Object)
    {
        writer.WriteStartObject();
    }
    else
    {
        return;
    }
    foreach (JsonProperty property in root.EnumerateObject())
    {
        property.WriteTo(writer);
    }
    writer.WriteEndObject();
    writer.Flush();
}
```

В этом методе мы задаем средство записи JSON, которое пишет JSON с отступами и способно работать только с пятью уровнями глубины. Затем с помощью этого средства записи мы записываем JSON-документ в файл.

В JSON-тексте в рассмотренном примере есть только три уровня глубины, поэтому он укладывается в наше максимальное значение 5. Если бы у нас было более пяти уровней, то этот код привел бы к выдаче исключения.

Конфигурация *JsonSerializerOptions* по умолчанию

Теперь мы добавим следующий метод, демонстрирующий доступность значений по умолчанию в объекте `JsonSerializerOptions`:

```
public static void ShowDefaultJsonSerializerOptions()
{
    var options = new JsonSerializerOptions
    {
        WriteIndented = true
    };
    Console.WriteLine("Showing default JsonSerializerOptions.");
    Console.WriteLine(JsonSerializer.Serialize(JsonSerializerOptions.
Default, options));
}
```

В этом методе мы сериализуем статическое свойство `Default` объекта `JsonSerializerOptions`, чтобы увидеть, какие настройки по умолчанию будут использоваться, если ни для одного из полей не задано явное значение.

Вызовы *HTTP PATCH* для JSON

Далее мы добавим следующий метод, чтобы продемонстрировать, как для JSON выполнить запрос `HTTP PATCH` из экземпляра класса `HttpClient`. Структура метода будет такой:

```
public static async Task DemoPatchAsJsonAsync()
{
    var client = new HttpClient();
    client.DefaultRequestHeaders.Add("Accept", "application/json");
    var jsonBody = new { Key = "value" };
    var response = await client.PatchAsJsonAsync("https://localhost", jsonBody);
}
```

Чтобы увидеть, как работает этот метод, нам потребуется разместить серверное приложение с соответствующей конечной точкой, принимающей запросы `PATCH` и полезную нагрузку объект `JSON`. В данном примере предполагается, что приложение будет размещено по адресу `https://localhost`. Однако если вы хотите посмотреть, как работает этот метод, то можете разместить соответствующее приложение по любому удобному для вас адресу.

Наконец, мы рассмотрим новую возможность полиморфизма JSON.

Полиморфизм JSON

Для демонстрации полиморфизма JSON нам потребуется добавить несколько классов, которые будут представлять объекты JSON. Мы можем добавить их в тот же файл сразу под определением класса NewJsonFeatures. И первая пара классов будет выглядеть следующим образом:

```
[JsonDerivedType(typeof(BasicBaseObject))]
[JsonDerivedType(typeof(BasicDerivedObject))]
public class BasicBaseObject
{
    public int BaseData { get; set; } = 1;
}
public class BasicDerivedObject : BasicBaseObject
{
    public int ExtraData { get; set; } = 2;
}
```

Здесь у нас два класса. Как указано атрибутами JsonDerivedType, оба этих типа предназначены для представления объекта JSON. И тот и другой могут наследовать и быть унаследованными. Именно поэтому, когда мы выводим класс BasicDerivedObject из BasicBaseObject, это будет известно сериализатору и десериализатору JSON.

Это самая базовая настройка полиморфизма JSON. Теперь добавим еще два класса, которые продемонстрируют более сложное использование:

```
[JsonDerivedType(typeof(BaseStringDiscriminator), typeDiscriminator: "baseObject")]
[JsonDerivedType(typeof(DerivedStringDiscriminator), typeDiscriminator: "derivedObject")]
public class BaseStringDiscriminator
{
    public int BaseData { get; set; } = 1;
}
public class DerivedStringDiscriminator : BaseStringDiscriminator
{
    public int ExtraData { get; set; } = 2;
}
```

Эти классы имеют параметры typeDiscriminator, заданные внутри атрибута JsonDerivedType. Данный параметр позволяет при десериализации привести JSON-строку к нужному типу. В C# при выведении одного класса из другого производный класс неявно сопоставляет свой тип с базовым типом. Поэтому, если ваш метод ожидает получения параметра типа BaseStringDiscriminator, в него можно передать переменную DerivedStringDiscriminator.

Параметр typeDescriminator позволяет задействовать эту особенность языка при десериализации JSON-строки в класс. Значение, указанное в этом параметре, соответствует значению поля $type внутри JSON-строки. При указании типа BaseStringDiscriminator в качестве цели десериализации JSON-строка будет десериализована в этот тип, если вы укажете baseObject в качестве значения поля $type. Но она будет десериализована в тип DerivedStringDiscriminator, если вы указали значение derivedObject.

Можно также использовать дискриминатор числового типа:

```
[JsonDerivedType(typeof(BaseIntDiscriminator), 0)]
[JsonDerivedType(typeof(DerivedIntDiscriminator), 1)]
public class BaseIntDiscriminator
{
    public int BaseData { get; set; } = 1;
}
public class DerivedIntDiscriminator : BaseIntDiscriminator
{
    public int ExtraData { get; set; } = 2;
}
```

В этом случае действует тот же принцип, как и для дискриминатора типов строк. Но здесь это будет целое число в поле `$type`.

Теперь посмотрим эти возможности в действии, добавив в наш класс `NewJsonFeatures` следующий метод:

```
public static void DemoJsonPolymorphism()
{
    Console.WriteLine("Demonstrating basic JSON polymorphism:");
    Console.WriteLine(JsonSerializer.Serialize(new BasicDerivedObject()));
    Console.WriteLine("Demonstrating JSON polymorphism with string type discriminator:");
    var jsonStringDiscrimnator = JsonSerializer.Deserialize<BaseStringDiscriminator>("""
    {
        "$type": "derivedObject",
        "ExtraData":2,
        "BaseData":1
    }
    """);
    Console.WriteLine($"JSON is of derived type: {jsonStringDiscrimnator
                                                  is DerivedStringDiscriminator}.");
    Console.WriteLine("Demonstrating JSON polymorphism with integer type discriminator:");
    var jsonIntDiscrimnator = JsonSerializer.Deserialize<BaseIntDiscriminator>("""
    {
        "$type": 1,
        "ExtraData":2,
        "BaseData":1
    }
    """);
    Console.WriteLine($"JSON is of derived type: {jsonIntDiscrimnator
                                                  is DerivedIntDiscriminator}.");
}
```

Итак, мы сериализуем объект с базовым полиморфизмом JSON в JSON-строку. Затем мы тестируем работу дискриминатора типов при десериализации JSON-строки в класс. При этом тестируем как числовой, так и текстовый дискриминатор типов.

Теперь добавим некоторую логику для выполнения написанного нами кода.

Тестирование возможностей JSON

Добавим в наш файл Program.cs следующий код:

```
Console.WriteLine("Demonstrating JSON improvements.");
NewJsonFeatures.DemoJsonWriterOptions();
NewJsonFeatures.ShowDefaultJsonSerializerOptions();
NewJsonFeatures.DemoJsonPolymorphism();
```

Результат показан на рис. 3.2.

```
Demonstrating JSON improvements.
Showing default JsonSerializerOptions.
{
  "Converters": [],
  "PolymorphicTypeConfigurations": [],
  "AllowTrailingCommas": false,
  "DefaultBufferSize": 16384,
  "Encoder": null,
  "DictionaryKeyPolicy": null,
  "IgnoreNullValues": false,
  "DefaultIgnoreCondition": 0,
  "NumberHandling": 0,
  "IgnoreReadOnlyProperties": false,
  "IgnoreReadOnlyFields": false,
  "IncludeFields": false,
  "MaxDepth": 0,
  "PropertyNamingPolicy": null,
  "PropertyNameCaseInsensitive": false,
  "ReadCommentHandling": 0,
  "UnknownTypeHandling": 0,
  "WriteIndented": false,
  "ReferenceHandler": null
}
Demonstrating basic JSON polymorphism:
{"ExtraData":2,"BaseData":1}
Demonstrating JSON polymorphism with string type discriminator:
JSON is of derived type: True.
Demonstrating JSON polymorphism with integer type discriminator:
JSON is of derived type: True.
```

Рис. 3.2. Результат работы наших методов, связанных с JSON

На этом демонстрация новых функций .NET 7 для работы с JSON завершена. Далее мы рассмотрим новые возможности класса `Stream`.

Новые возможности класса *Stream*

Основное улучшение в классе `Stream` базовой библиотеки .NET `System` заключается в добавлении двух методов: `ReadExactly` и `ReadAtLeast`. Эти методы обеспечивают большую гибкость при чтении данных из потока в буфер массива байтов.

Метод `ReadExactly` считывает точное число байтов, содержащихся в буфере. Если данных недостаточно, чтобы заполнить буфер, выдается исключение.

В методе `ReadAtLeast` есть определенный параметр, задающий минимальное число байтов, которые требуется прочитать. Буфер не обязательно должен быть заполнен. Исключение выдается, если для чтения доступно меньше байтов, чем указано в параметре.

Чтобы продемонстрировать оба этих метода в действии, добавим в наш проект CoreLibraryImprovements файл NewStreamFeatures.cs и наполним его следующим содержимым:

```
namespace CoreLibraryImprovements;
public static class NewStreamFeatures
{
    public static void DemoReadExactly()
    {
       using var fileStream = File.Open("output.json", FileMode.Open);
        var buffer = new byte[10];
        fileStream.ReadExactly(buffer);
        Console.WriteLine($"""
        Bytes read with ReadExactly:
        {BitConverter.ToString(buffer)}
        """);
    }
    public static void DemoReadAtLeast()
    {
       using var fileStream = File.Open("output.json", FileMode.Open);
        var buffer = new byte[10];
        fileStream.ReadAtLeast(buffer, 10);
        Console.WriteLine($"""
        Bytes read with ReadAtLeast:
        {BitConverter.ToString(buffer)}
        """);
    }
}
```

Итак, у нас есть два метода: `DemoReadExactly` и `DemoReadAtLeast`. Оба метода считывают данные из файла, который был создан, когда мы тестировали некоторые новые функции, связанные с JSON. Метод `DemoReadExactly` демонстрирует, как использовать метод `ReadExactly` класса `Stream`, который доступен любому классу, наследующему от `Stream`, такому как `FileStream` из нашего примера. Метод `DemoReadAtLeast` демонстрирует действие метода `ReadAtLeast`. Здесь мы устанавливаем длину буфера и минимальное число байтов равными 10, что сделает поведение обоих методов идентичным.

Теперь добавим в файл Program.cs следующий код для проверки этих новых методов:

```
Console.WriteLine("Demonstrating Stream improvements.");
NewStreamFeatures.DemoReadExactly();
NewStreamFeatures.DemoReadAtLeast();
```

Если мы теперь запустим нашу программу, то должны ожидать получения результата, аналогичного показанному на рис. 3.3.

На этом обзор новых возможностей работы с потоками в .NET 7 завершен. В следующем разделе мы рассмотрим улучшения механизма RegEx.

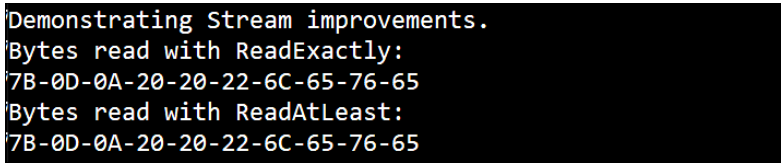


Рис. 3.3. Данные, считанные методами `ReadExactly` и `ReadAtleast`

Улучшения RegEx

RegEx (регулярные выражения) — технология, которая позволяет программистам проверять, соответствует ли некоторый произвольный текст определенному шаблону. Это очень мощный инструмент, поскольку вы можете построить любой шаблон, используя относительно небольшое число символов. Но из-за его способности работать с практически безграничным количеством шаблонов, правил и их комбинаций применение регулярных выражений требует относительно больших вычислительных затрат.

И в .NET 7 появилась функция, которая существенно повышает производительность регулярных выражений. Если вы хотите, чтобы ваш код работал только с определенным регулярным выражением, то можете предварительно скомпилировать его. Благодаря этому сравнение с шаблоном во время выполнения будет очень быстрым. И именно это мы и рассмотрим далее.

Добавим в проект `CoreLibraryImprovements` файл NewRegexFeatures.cs со следующим содержимым:

```
using System.Text.RegularExpressions;
namespace CoreLibraryImprovements;
public partial class NewRegexFeatures
{
    [RegexGenerator(@"^[a-z]+$", RegexOptions.IgnoreCase)]
    public static partial Regex LettersRegex();
    public static void DemoPrecompiledRegex(string input)
    {
        Console.WriteLine($"'{input}' matches '^[a-z]+$' RegEx:
           {LettersRegex().IsMatch(input)}.");
        Console.WriteLine($"Число совпадений: {LettersRegex().Count(input)}.");
        var matchEnumerator = LettersRegex().EnumerateMatches(input);
        while (matchEnumerator.MoveNext())
            Console.WriteLine($"Match of {matchEnumerator.Current.Length}
                                    found at index {matchEnumerator.Current.Index}.");
    }
}
```

Здесь мы сначала определяем метод, возвращающий регулярное выражение. Приведенный выше атрибут `RegexGenerator` указывает на то, что данный объект сопровождается предварительно скомпилированным шаблоном. Чтобы это работало, необходимо наличие статических и частичных модификаторов доступа. В нашем случае этот шаблон определяет, состоит ли конкретный текст полностью из букв. Другие символы не допускаются.

Затем у нас есть метод `DemoPrecompiledRegex`, который использует предварительно скомпилированный шаблон RegEx. Он принимает входной текст и сравнивает его с шаблоном. В результате этого сравнения формируется вывод в консоли.

Этот метод также демонстрирует еще одно усовершенствование регулярных выражений, появившееся в .NET 7, — метод `EnumerateMatches`. Этот метод позволяет перечислить все части входного текста, соответствующие шаблону.

Еще одна полезная особенность нового предварительно скомпилированного регулярного выражения, о которой стоит упомянуть, заключается в том, что анализатор Roslyn автоматически выделяет в коде места, где существующую логику регулярного выражения можно заменить на новую. Это позволит вам переработать структуру и оптимизировать старый код всего несколькими щелчками мыши, если вы работаете с Roslyn-совместимой IDE, например Visual Studio.

Теперь протестируем этот метод, добавив в файл Program.cs нашего проекта следующие строки кода:

```
Console.WriteLine("Demonstrating RegEx improvements.");
var lettersOnlyText = "letters";
var mixedText = "fwef340";
NewRegexFeatures.DemoPrecompiledRegex(lettersOnlyText);
NewRegexFeatures.DemoPrecompiledRegex(mixedText);
```

Здесь мы проводим сопоставление двух строк. Одна из них содержит только буквы, что и предполагает наш шаблон RegEx, а другая содержит также и цифры. На рис. 3.4 показан результат, который ожидается при выполнении этого кода.

На этом завершим демонстрацию усовершенствований, касающихся регулярных выражений, и рассмотрим новые возможности, связанные с криптографией.

```
Demonstrating RegEx improvements.
'letters' matches '^[a-z]+$' RegEx: True.
The number of matches: 1.
Match of 7 found at intex 0.
'fwef340' matches '^[a-z]+$' RegEx: False.
The number of matches: 0.
```

Рис. 3.4. Демонстрация регулярных выражений с опережающей компиляцией

Улучшения в криптографии

В .NET всегда существовало множество способов работы с сертификатами шифрования SSL/TLS. Ранее работа с атрибутами имени сертификата не всегда была удобна. Основной способ заключался в передаче строки, в которой были закодиро-

ваны различные атрибуты. Но это неудобно, если в каком-либо из имен присутствуют специальные символы. Кроме того, поскольку вы имели дело с необработанными строками, было легко допустить опечатку.

В .NET 7 эта проблема была исправлена путем добавления класса `X500DistinguishedNameBuilder`, который содержит методы, позволяющие добавлять атрибуты именования, применяемые в сертификатах шифрования. В результате каждое имя стало достаточно коротким, и вероятность случайной ошибки в написании сведена к минимуму. А поскольку имя рассматривается как отдельная единица, то наличие специальных символов внутри него не нарушит целостности всей структуры атрибута.

Для демонстрации класса `X500DistinguishedNameBuilder` добавим в наш проект файл CryptographyEnhancements.cs и наполним его следующим содержимым:

```
using System.Security.Cryptography.X509Certificates;
namespace CoreLibraryImprovements;
public static class CryptographyEnhancements
{
    public static void DemoCertificateNameBuilder()
    {
        var builder = new X500DistinguishedNameBuilder();
        builder.AddCommonName("CertificateSubject");
        builder.AddOrganizationalUnitName("TestUnit");
        builder.AddOrganizationName("Scientific Programmer Ltd.");
        Console.WriteLine($"The certificate name is: {
                                builder.Build().Decode(X500DistinguishedNameFlags.None)}");
    }
}
```

Здесь мы добавляем в конструктор несколько произвольных имен, выстраивая структуру имен, а затем расшифровываем ее, чтобы увидеть, из чего она состоит. Теперь добавим в файл Program.cs следующий код, который позволит нам запустить этот метод:

```
Console.WriteLine("Demonstrating cryptography improvements.");
CryptographyEnhancements.DemoCertificateNameBuilder();
```

Результат выполнения кода показан на рис. 3.5.

Теперь перейдем к рассмотрению еще одной чрезвычайно полезной возможности .NET 7 — TAR API.

```
Demonstrating cryptography improvements.
The certificate name is: O=Scientific Programmer Ltd., OU=TestUnit, CN=CertificateSubject
```

Рис. 3.5. Декодированный результат использования класса `X500DistinguishedNameBuilder`

Новый TAR API

TAR — один из популярных алгоритмов архивирования файлов. Однако, несмотря на эту популярность, в .NET не было встроенных функций для работы с этим алгоритмом. Но теперь это упущение исправлено.

Чтобы продемонстрировать, из чего состоит эта новая функциональность, добавим в наш проект файл TarApi.cs, наполнив его следующим содержимым:

```
using System.Formats.Tar;
using System.IO.Compression;
namespace CoreLibraryImprovements;
public static class TarApi
{
}
```

Теперь добавим в этот класс следующий метод, который демонстрирует, как с помощью TAR API выбрать все файлы, находящиеся в указанной папке, и сохранить их в TAR-архиве в указанном месте:

```
public static void CreateTarFile(
    string sourceDirectoryName,
    string destinationFileName)
{
    TarFile.CreateFromDirectory(
        sourceDirectoryName: sourceDirectoryName,
        destinationFileName: destinationFileName,
        includeBaseDirectory: true);
}
```

Далее мы добавим следующий метод, чтобы продемонстрировать, как извлечь содержимое TAR-файла в указанный каталог:

```
public static void ExtractTarFile(
    string sourceFileName,
    string destinationDirectoryName)
{
    TarFile.ExtractToDirectory(
        sourceFileName: sourceFileName,
        destinationDirectoryName: destinationDirectoryName,
        overwriteFiles: false);
}
```

Рассмотренный API также может работать с потоками, в том числе с потоками в памяти, что позволяет обрабатывать сложную логику без сохранения промежуточных файлов на диске. Например, следующий пример демонстрирует, как можно извлечь TAR-архив в поток в памяти, а затем записать из этого потока другой TAR-архив:

```
public static void CreateTarFileFromStream(
    string sourceDirectoryName,
    string destinationDirectoryName)
```

```
{
    using var stream = new MemoryStream();
    TarFile.CreateFromDirectory(
        sourceDirectoryName: sourceDirectoryName,
        destination: stream,
        includeBaseDirectory: true);

    TarFile.ExtractToDirectory(
        source: stream,
        destinationDirectoryName: destinationDirectoryName,
        overwriteFiles: false);
}
```

Этот API также позволяет передать все или только часть содержимого одного TAR-файла в другой. Как это можно сделать, демонстрирует следующий метод:

```
public static void TransferFilesToDifferentArchive(
    string sourceFileName,
    string destinationFileName)
{
    using var stream = File.OpenRead(sourceFileName);
    using var reader = new TarReader(stream, leaveOpen: false);

    TarEntry? entry;
    while ((entry = reader.GetNextEntry()) != null)
    {
        destinationFileName = Path.Join(destinationFileName, entry.Name);
        entry.ExtractToFile(destinationFileName, overwrite: true);
    }
}
```

Кроме того, вместе с TAR можно использовать алгоритм сжатия GZIP:

```
public static void ExtractFromGzipArchive(
    string sourceFileName,
    string destinationDirectoryName)
{
    using var compressedStream = File.OpenRead(sourceFileName);
    using var decompressor = new GZipStream(compressedStream, CompressionMode.Decompress);
    TarFile.ExtractToDirectory(
        source: decompressor,
        destinationDirectoryName: destinationDirectoryName,
        overwriteFiles: false);
}
```

На этом обзор нового TAR API, добавленного в седьмую версию платформы .NET, завершен.

В базовых библиотеках .NET 7 также есть ряд улучшений, которые заслуживают отдельного рассмотрения. Речь идет об улучшениях, связанных с наблюдаемостью. О них мы и поговорим далее.

Улучшения наблюдаемости

В .NET 7 добавлено несколько новых способов, которые могут помочь в мониторинге приложений. В платформу внесен целый ряд улучшений, поскольку разработчики хотят, чтобы она соответствовала развивающимся стандартам облачно-ориентированных технологий `OpenTelemetry`.

Новые функции наблюдаемости в .NET 7 можно условно разделить на следующие категории:

- новые способы мониторинга действий;
- показатель `UpDownCounter`.

Чтобы продемонстрировать эти улучшения, создадим новый проект консольного приложения и назовем его `ObservabilityImprovements`. Создать его можно либо через графический интерфейс среды разработки, либо выполнив следующую команду:

```
dotnet new console -o ObservabilityImprovements
```

Теперь начнем добавлять в него содержимое. Сначала рассмотрим новые способы мониторинга действий.

Новые способы мониторинга действий

`System.Diagnostics.Activity` — это класс, представляющий запущенный процесс в контексте ведения журнала. Его основное назначение — получение информации о состоянии запущенных процессов, а также других вспомогательных сведений и их запись в журнал.

Чтобы посмотреть, какие возможности мониторинга действий доступны в .NET 7, добавим в проект `ObservabilityImprovements` файл ActivityMonitoring.cs со следующим содержимым:

```
using System.Diagnostics;
namespace ObservabilityImprovements;
public static class ActivityMonitoring
{
}
```

Теперь начнем добавлять в него различные методы, чтобы продемонстрировать новые возможности мониторинга действий. Начнем с добавления логики, которая отслеживает остановленное действие.

Мониторинг остановленных действий

Для мониторинга остановленных действий в класс `Activity` было добавлено свойство `IsStopped`. Чтобы продемонстрировать его использование, мы можем добавить в наш только что созданный класс следующий метод:

```
public static void MonitorStoppedActivity()
{
    var activity = new Activity("test");
```

```
    activity.Start();
    activity.Stop();
    Console.WriteLine($"Is activity stopped? {activity.IsStopped}");
}
```

Здесь мы сначала запускаем действие, затем останавливаем его и проверяем состояние.

Далее мы рассмотрим возможность вызова события при изменении текущего действия.

Событие изменения текущего действия

Чтобы продемонстрировать, как изменение текущего действия может вызвать событие, добавим следующий метод:

```
public static void DemoCurrentChangedEvent()
{
    Activity.CurrentChanged += ChangeEvent;
    var activity = new Activity("test");
    activity.Start();
    activity = new Activity("test2");
    activity.Start();
    void ChangeEvent(object? sender, ActivityChangedEventArgs e)
    {
        Console.WriteLine($"Operation changed from {
            (e.Previous?.OperationName ?? "[No Activity]")} to {
                e.Current?.OperationName}.");
    }
}
```

Здесь мы связываем обработчик событий со статическим событием `CurrentChanged` в классе `Activity`. Затем запускаем новое действие, которое создаст триггер для этого события. Событие генерируется, даже если текущему действию предшествовало отсутствие действия. Затем мы создаем еще одно действие, которое запускает еще одно изменение.

Теперь рассмотрим новую возможность перечисления свойств действия.

Перечисление свойств действия

В .NET 7 добавлена возможность перечисления тегов, ссылок и событий действия, которая реализована путем добавления методов `EnumerateTagObjects`, `EnumerateLinks` и `EnumerateEvents`. Чтобы продемонстрировать, как это работает, добавим следующий метод:

```
public static void DemoActivityEnumerators()
{
    var activity = new Activity("test");
    activity.SetTag("tag1", "value1");
```

```
    activity.SetTag("tag2", "value2");
    activity.SetTag("tag3", "value2");
    Console.WriteLine("Activity has the following tags:");
    foreach (ref readonly KeyValuePair<string, object?> tag
                                                    in activity.EnumerateTagObjects())
    {
        Console.WriteLine($"Имя тега: {tag.Key}, tag value: {tag.Value}");
    }
    activity.AddEvent(new ActivityEvent("event1"));
    activity.AddEvent(new ActivityEvent("event2"));
    Console.WriteLine("Activity has the following events:");
    foreach (var ev in activity.EnumerateEvents())
    {
        Console.WriteLine($"Event name: {ev.Name}");
    }
}
```

Кроме того, сами эти свойства имеют теги, которые можно перечислить:

```
public static void DemoInnerTagEnumerators()
{
    var tagCollection = new List<KeyValuePair<string, object?>>()
    {
        new KeyValuePair<string, object?>("tag1", "value1"),
        new KeyValuePair<string, object?>("tag2", "value2"),
    };
    var activityLink = new ActivityLink(default, new ActivityTagsCollection(tagCollection));
    Console.WriteLine("ActivityLink has the following tags:");
    foreach (ref readonly KeyValuePair<string, object?> tag
        in activityLink.EnumerateTagObjects())
    {
        Console.WriteLine($"Tag name: {tag.Key}, tag value: {tag.Value}");
    }
    var e = new ActivityEvent("TestEvent", tags: new ActivityTagsCollection(tagCollection));
    Console.WriteLine("ActivityEvent has the following tags:");
    foreach (ref readonly KeyValuePair<string, object?> tag
        in e.EnumerateTagObjects())
    {
        Console.WriteLine($"Tag name: {tag.Key}, tag value: {tag.Value}");
    }
}
```

Далее мы рассмотрим новый тип показателя, добавленный в платформу .NET 7, — `UpDownCounter`.

Показатель *UpDownCounter*

`UpDownCounter` — это класс, который был добавлен в пространство имен `System.Diagnostics.Metrics`. Он предназначен для использования вместе с классом `Meter` из того же пространства имен.

Этот тип показателя в других библиотеках телеметрии часто называют индикатором. Для обычного показателя-счетчика предусмотрено только увеличение. Однако датчик (или реверсивный счетчик) предназначен для счета в прямом или обратном направлении. В следующем примере показано, как можно задействовать этот показатель:

```
public static Meter MeterObject = new Meter("HTTP.Connections", "1.0.0");
public static UpDownCounter<int> ActiveConnections =
                                 MeterObject.CreateUpDownCounter<int>("Active-Connections");
```

Затем, чтобы изменить значение показателя, мы можем вызвать метод `Add` с положительным или отрицательным целочисленным параметром:

```
ActiveConnections.Add(10);
ActiveConnections.Add(-2);
```

Обычный показатель-счетчик предназначен для мониторинга величин, которые могут только увеличиваться. Например, он пригоден для мониторинга количества запросов, которые получает система. В свою очередь, реверсивный счетчик предназначен для отслеживания величин, которые должны меняться в реальном времени. Например, с его помощью можно реализовать оперативный мониторинг числа пользователей, присутствующих в системе. Когда пользователь входит в систему, значение счетчика увеличивается. А когда кто-то выходит из системы, значение счетчика уменьшается.

На этом завершим обзор изменений, внесенных в .NET 7, и рассмотрим критические изменения, из-за которых ваш старый код может оказаться не полностью совместимым с .NET 7.

Критические изменения в .NET 7

Далее рассмотрим основные изменения, из-за которых ваш старый код после перехода на .NET 7 может вести себя по-другому либо вообще не работать. И именно поэтому важно их знать.

Допустимость неопределенных значений в пространстве имен Microsoft.Extensions

По сравнению с .NET 6 изменена допустимость неопределенных значений для данных различных типов. Например, до .NET 6 объекты, как и строки, допускали возможность значения `NULL`. Этот подход изменился. Теперь для того, чтобы они допускали значение `NULL`, их необходимо явно пометить как таковые.

С одной стороны, вы все еще можете настроить свои приложения на реализацию прежнего поведения. Но, с другой стороны, существует множество библиотек, которые просто не поспевают за этими изменениями. В их число входят различные библиотеки, использующие пространство имен Microsoft.Extensions.

Но теперь это исправлено, и библиотеки полностью соответствуют правилам допустимости неопределенных значений как в новой, так и в старой версии .NET. Однако это может вызвать потенциальные проблемы, если вы использовали библиотеки в .NET 6 с адаптацией к новым правилам допустимости неопределенных значений. Если вы применяете какие-либо библиотеки с этим пространством имен и перенесли свое приложение на .NET 7, убедитесь, что соответствующие фрагменты кода по-прежнему работают так, как нужно.

Конечные точки: устаревшие и не допускающие неопределенных значений

В различных библиотеках .NET есть и другие конечные точки, поведение которых в отношении допустимости неопределенных значений было изменено. Кроме того, некоторые конечные точки были признаны устаревшими либо удалены. При наличии конечных точек первого типа выдается предупреждение. Конечные точки второго типа будут препятствовать компиляции вашего кода.

Самый простой способ определить, затронут ли ваш код, — перенести решение на .NET 7 и попробовать собрать его. Если появились новые предупреждения или ошибки, связанные с устареванием, значит, в код необходимо внести некоторые изменения, чтобы он соответствовал новым API.

Ограничение *PatternContext*

Обобщенный класс `PatternContext<T>` позволяет использовать в качестве `T` любой тип. Однако в .NET 7 для этой цели допускается только `struct`. Если до этого у вас был любой другой тип, то ваш код больше не будет компилироваться.

Многоуровневый поиск в Windows отключен

Ранее при выполнении команды `dotnet` осуществлялся поиск среды в нескольких возможных местах установки. Но теперь этот поиск ведется только в одном месте, которое указано переменной среды `DOTNET_ROOT`. Для большинства разработчиков такой проблемы, скорее всего, не будет, но знать об этом все же стоит.

Сериализация пользовательских типов в *MSBuild*

В .NET 7 `MSBuild` больше не поддерживает сериализацию типов, производных от `BuildEventArgs` и `ITaskItem`, через сериализатор `BinaryFormatter`. Поэтому код, в котором ранее использовались эти типы, необходимо изменить.

Заключение

В платформу .NET 7 было добавлено множество полезных функций. Они охватывают как базовый SDK, так и основные библиотеки, которые существуют в пространствах имен под `System` или `Microsoft`. Некоторые из этих улучшений носят нефункциональный характер, например повышают производительность компилятора. Но есть и множество функциональных улучшений, наиболее важные из которых были описаны в этой главе.

Среди наиболее заметных улучшений SDK, рассмотренных в этой главе, — новые возможности интерфейса командной строки, функции NativeAOT и центральный диспетчер пакетов NuGet. Улучшения базовых библиотек — новые API, связанные со временем, новые функции для работы с потоками и JSON, предварительная компиляция для RegEx, улучшения в криптографии и новый API для архиватора TAR.

Среди новых функций базовых библиотек были особо выделены улучшения наблюдаемости, добавленные для приведения .NET в соответствие со стандартами `OpenTelemetry`. К ним относятся новые способы мониторинга действий и новые типы показателей.

Однако в .NET были внесены и некоторые критические изменения. Большинство из них достаточно легко выявить — достаточно перенести свой код на .NET 7 и попробовать собрать приложение. При этом для устаревших функций будут выдаваться предупреждения, а полностью измененные функции не позволят собрать ваш код.

В следующей главе мы расскажем о MAUI — фреймворке для создания приложений в машинном коде, которые можно перенести как на мобильные устройства, так и на настольные компьютеры.

Что следует помнить

- Интерфейс командной строки в .NET 7 стал более удобным и предоставляет больше информации о поддерживаемых командах.
- Приложения .NET 7 можно скомпилировать в нативные исполняемые файлы с автоматической обрезкой библиотек.
- Центральный диспетчер пакетов NuGet помогает разработчикам управлять зависимостями внутри сложных решений.
- К объектам, связанным со временем, добавлены микросекунды и наносекунды.
- В .NET 7 появились новые функции для работы с JSON, включающие поддержку полиморфизма JSON, доступ к функциям сериализации по умолчанию и т. д.
- Новые функции для работы с потоками позволяют указать точное число байтов для чтения в поток.
- Предварительная компиляция регулярных выражений позволяет гораздо быстрее выполнять сопоставление с текстовым шаблоном.

- В .NET 7 можно легко добавлять атрибуты имен криптографических сертификатов.
- Добавлен новый API для работы с файлами TAR.
- Улучшения наблюдаемости, добавленные в .NET 7, включают в себя мониторинг действий и новые типы показателей.
- Также в .NET 7 были внесены некоторые критические изменения, о которых следует знать разработчикам.

Контрольные вопросы

1. Что такое NativeAOT?

а) Фреймворк создания приложений для мобильных устройств;

б) Фреймворк создания приложений для настольных компьютеров;

в) Технология, позволяющая скомпилировать код в приложение, выполняемое непосредственно на конкретном типе аппаратных средств;

г) Ничего из перечисленного.

2. Что такое полиморфизм JSON?

а) Возможность наследования от класса JSON и десериализации JSON-строки в соответствующий тип;

б) Возможность замены типов данных в полях JSON;

в) Возможность добавлять или удалять поля JSON по своему усмотрению;

г) Возможность хранить как отдельный JSON-объект, так и коллекцию таких объектов в одном поле.

3. Для чего можно использовать .NET TAR API?

а) Распаковка и упаковка TAR-файлов;

б) Перенос содержимого между архивами TAR;

в) Перенос содержимого между архивами TAR и памятью;

г) Все вышеперечисленное.

4. **Какие из указанных далее свойств класса `System.Diagnostics.Activity` можно перечислять?**

а) Теги;

б) Ссылки;

в) События;

г) Все перечисленное.

Ответы

1. в. **3.** г.

2. а. **4.** г.

Основные термины

- **NativeAOT** — технология, позволяющая разработчикам компилировать свое приложение в формат, близкий к аппаратному, для работы на компьютере конкретного типа.
- **Обрезка библиотек** — процесс удаления определенных .NET-компонентов из библиотек для обеспечения их совместимости с NativeAOT.
- **Центральный диспетчер пакетов NuGet** — функция централизованного управления зависимостями NuGet для всего решения.
- **JSON** — JavaScript Object Notation — формат для передачи данных.
- **Сериализация JSON** — процесс преобразования объекта из кода в JSON-строку.
- **Десериализация JSON** — процесс преобразования JSON-строки в определенный класс.
- **Полиморфизм JSON** — возможность используемых для JSON классов наследовать друг от друга и получать JSON-строку для десериализации в соответствующий производный класс, когда в качестве цели десериализации указан базовый класс.
- **RegEx** (REGular EXpression) — технология сопоставления строк с шаблоном.
- **TAR** — формат архивирования файлов.
- **GZIP** — алгоритм сжатия данных, используемый совместно с TAR.
- **OpenTelemetry** — фреймворк для применения функций наблюдаемости в облачно-ориентированных приложениях.

Глава 4

MAUI и кросс-платформенные нативные приложения

Введение

Многоплатформенный пользовательский интерфейс (Multi-platform UI, MAUI) — это фреймворк для разработки нативных приложений для мобильных устройств и настольных компьютеров на единой кодовой базе. Первоначально интерфейс MAUI был представлен в .NET 6, но достиг своей зрелости лишь через несколько месяцев после выхода первых выпусков .NET 6. Поэтому на момент выхода .NET 7 эту технологию все еще целесообразно считать новой.

До появления MAUI существовало несколько средств разработки нативных приложений в .NET. Приложения для настольных компьютеров с ОС Windows можно было создавать с помощью WinForms и Windows Presentation Framework. А для разработки кросс-платформенных мобильных приложений использовали Xamarin.Forms. Этот инструмент имел ограниченные возможности по созданию Windows-приложений с помощью Unified Windows Platform. При этом в .NET не было встроенного инструмента разработки приложений для настольных компьютеров с операционными системами, отличными от Windows. И не было единой платформы, позволяющей одинаково хорошо разрабатывать приложения и для настольных компьютеров, и для мобильных устройств.

Но появление MAUI изменило правила игры. Это полностью унифицированная платформа для разработки нативных приложений любых типов на основе единой кодовой базы. Из этой главы вы узнаете, как использовать этот инструмент.

Структура

В этой главе мы рассмотрим новые возможности .NET 7, которые сведены в следующие темы:

- введение в MAUI;
- использование MAUI для создания настольных приложений;
- использование MAUI для создания мобильных приложений;
- ограничения разработки для Mac OS и iOS.

Цели

Прочитав эту главу до конца, вы сможете создать свое собственное приложение с помощью MAUI. Вы познакомитесь с основными паттернами проектирования, используемыми в MAUI. Вы также узнаете, как скомпилировать свою кодовую базу в приложение для любой операционной системы, поддерживаемой MAUI.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из перечисленных компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Общие сведения о MAUI

Для тех, кто ранее использовал для создания мобильных приложений Xamarin.Forms, интерфейс MAUI покажется достаточно простым в освоении. Этому есть веские причины, поскольку MAUI является развитием Xamarin.Forms.

Но это не просто новая версия Xamarin.Forms, в которой появились дополнительные возможности развертывания настольных приложений. MAUI отличается от своего предшественника по многим принципиальным моментам. Например, в нем гораздо более простая структура проектов, в которой все поддерживаемые платформы развертывания, как настольных, так и мобильных приложений, сведены в один проект. При этом в своем приложении вы по-прежнему можете использовать и дополнительные проекты. Специально для MAUI предусмотрен даже новый тип шаблона проекта для библиотек классов. Но нам больше не нужно создавать отдельный проект для каждой целевой операционной системы.

Также между Xamarin.Forms и MAUI существуют некоторые различия в синтаксисе разметки для построения пользовательских интерфейсов. Несмотря на то что обе технологии поддерживают XAML в качестве основного языка разметки, в них применяются разные библиотеки элементов. Поэтому некоторые элементы в компоновках приложений будут иметь разные имена и атрибуты.

Но, пожалуй, наиболее существенное отличие Xamarin.Forms от MAUI заключается в том, что первый инструмент — это отдельный фреймворк, тогда как второй полностью интегрирован в .NET, начиная с шестой версии. Поэтому MAUI имеет доступ ко всем новейшим языковым возможностям и ко всем остальным функциям

фреймворка .NET. Например, в приложениях MAUI возможны команды интерфейса командной строки .NET. И именно это мы и сделаем далее.

Включение среды разработки MAUI

Прежде чем приступить к написанию MAUI-приложений, необходимо включить рабочую нагрузку MAUI на вашем рабочем компьютере. Это можно сделать с помощью .NET CLI, выполнив следующую команду:

```
dotnet workload install maui
```

Затем можно установить инструмент, который проверит наличие в среде недостающих зависимостей MAUI:

```
dotnet tool install -g Redth.Net.Maui.Check
```

Для последующего применения этого инструмента можно выполнить следующую команду:

```
maui-check
```

В качестве альтернативы можно настроить рабочую нагрузку MAUI с помощью средств IDE. Например, если вы работаете с Visual Studio под Windows, можно открыть Visual Studio Installer и нажать кнопку **Modify** (Изменить) рядом с последней версией IDE, как показано на рис. 4.1.

Затем необходимо включить рабочие нагрузки с названием .NET Multi-platform App UI development (Разработка пользовательского интерфейса приложений на .NET для нескольких платформ), как показано на рис. 4.2.

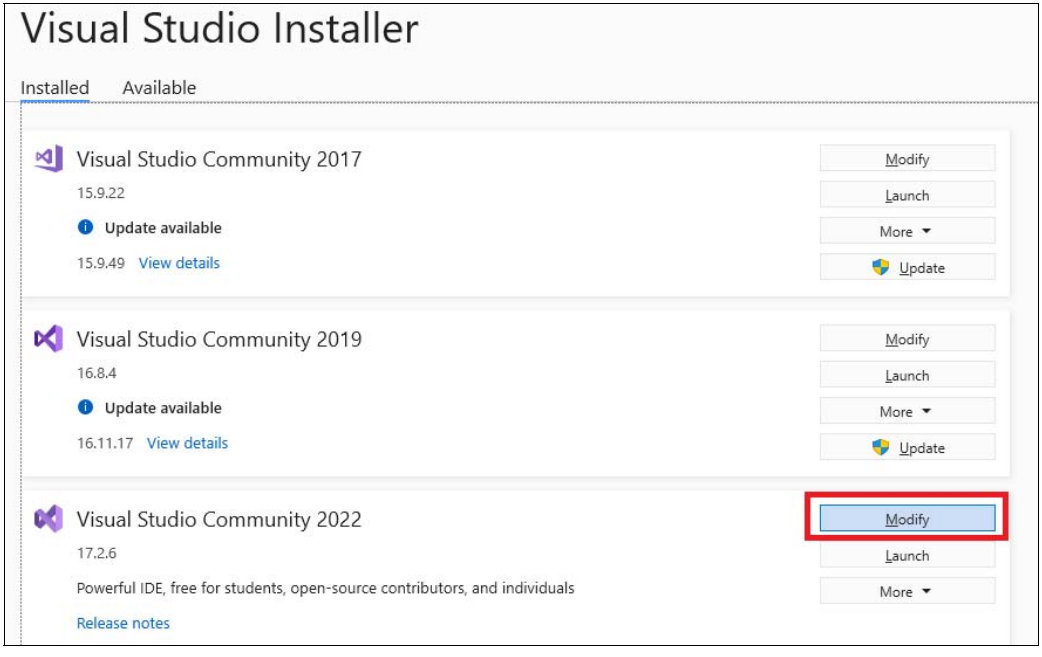


Рис. 4.1. Изменение рабочих нагрузок с помощью Visual Studio Installer

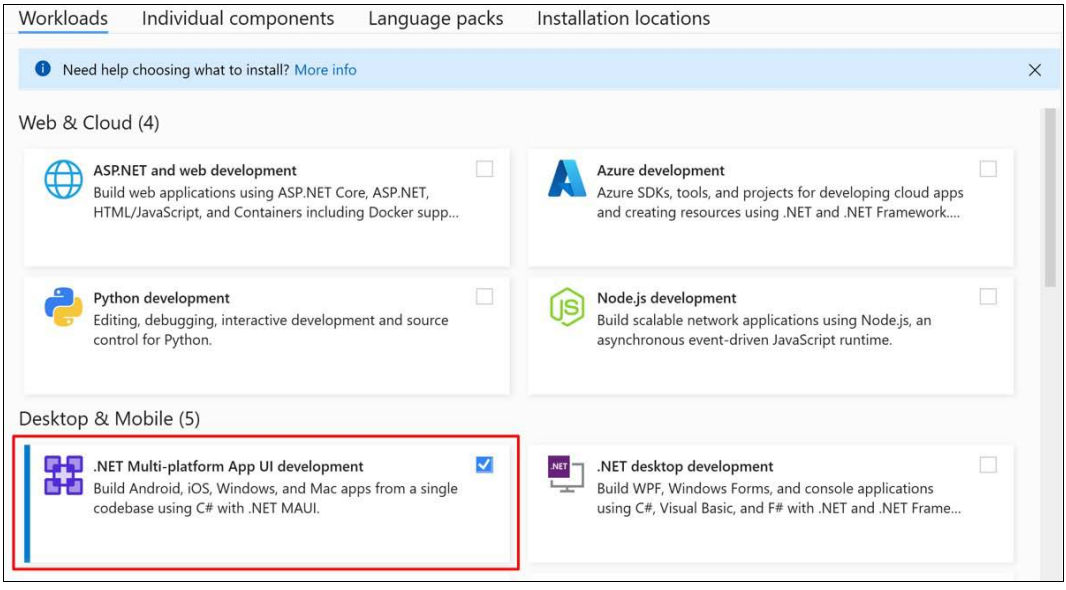


Рис. 4.2. Включение рабочей нагрузки MAU в Visual Studio Installer

Теперь мы можем приступить к созданию MAUI-приложений, чтобы посмотреть, как они работают.

Создание простых MAUI-приложений

Для создания MAUI-приложения с именем `SampleMauiApp` выполним следующую команду:

```
dotnet new maui -o SampleMauiApp
```

Теперь мы можем взглянуть на базовую структуру проекта MAUI-приложения, которая представлена на рис. 4.3.

По умолчанию в корневой папке находится несколько файлов с кодом. MauiProgram.cs — это файл, содержащий точку входа в приложение, которая является общей для всех платформ. App.xaml — это базовый файл, используемый при-

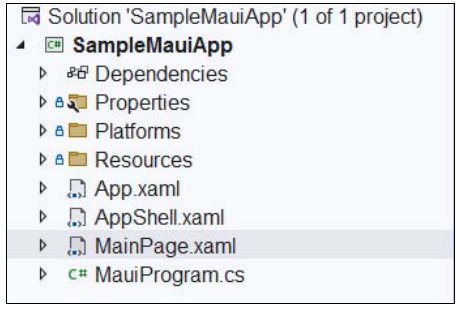


Рис. 4.3. Корневая папка проекта MAUI

ложением. Он содержит ссылки на все общие ресурсы, например на конкретные шрифты и стили. Для него предусмотрен файл кода программной части с именем App.xaml.cs. Этот файл указывает приложению, какую компоновку оболочки следует загрузить.

Файл AppShell.xaml содержит основную компоновку приложения. Ее назначение — предоставление общей структуры для всех страниц вашего приложения. Сама по себе она не предназначена для представления пользовательского интерфейса приложения. Для этой цели в нашем примере служит файл MainPage.xaml. Как и для всех других XAML-файлов, для него предусмотрен файл кода программной части на C# с расширением cs.

Проект MAUI также содержит папку Platforms, в которой хранится соответствующий конкретной платформе код точки входа и любой другой нативный код, применимый к платформам. Пример структуры этой папки показан на рис. 4.4.

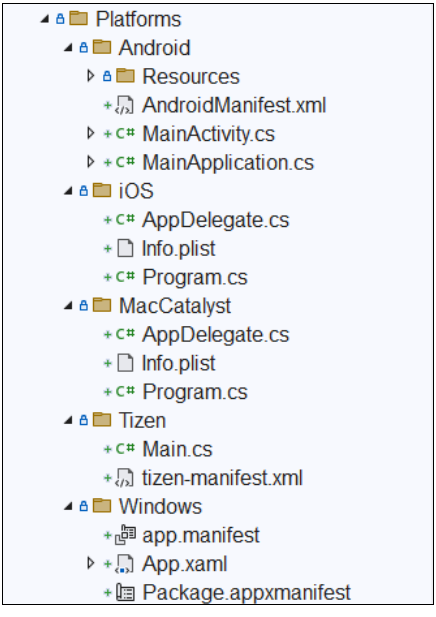


Рис. 4.4. Структура папки Platforms

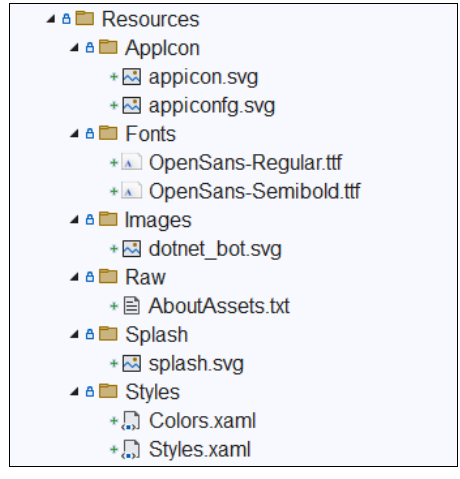


Рис. 4.5. Структура папки Resources

И наконец, еще одна заслуживающая внимания папка проекта MAUI — Resources — содержит все изображения, шрифты, текст для конкретного языкового стандарта и любые другие ресурсы, которые могут понадобиться вашему приложению. Структура этой папки полказана на рис. 4.5.

Теперь внесем некоторые изменения в файл MainPage.xaml, чтобы продемонстрировать, как он работает. Если мы откроем его, то увидим различные вложенные элементы. Корневой элемент называется `ContentPage`. Это элемент-контейнер, который представляет всю страницу. Непосредственно внутри находится элемент `ScrollView`. Это элемент-контейнер со встроенной функцией прокрутки, если все содержимое страницы не помещается на одном экране. Затем следуют элементы `Image`

и Label, которые представляют собой изображение и текстовую метку соответственно.

Также имеется элемент Button, который представляет собой кнопку, на которую можно нажать. Но кроме нее у нас нет никаких элементов, с которыми пользователь мог бы взаимодействовать. Поэтому мы добавим такой элемент. Вставим элемент Entry прямо над элементом Button. Этот элемент представляет собой поле с редактируемым текстом. Теперь содержимое файла MainPage.xaml выглядит следующим образом:

```
<?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="SampleMauiApp.MainPage">

    <ScrollView>
        <VerticalStackLayout
            Spacing="25"
            Padding="30,0"
            VerticalOptions="Center">
            <Image
                Source="dotnet_bot.png"
                SemanticProperties.Description="Cute dot net bot waving hi to you!"
                HeightRequest="200"
                HorizontalOptions="Center" />

            <Label
                Text="Hello, World!"
                SemanticProperties.HeadingLevel="Level1"
                FontSize="32"
                HorizontalOptions="Center" />

            <Label
                Text="Welcome to .NET Multi-platform App UI"
                SemanticProperties.HeadingLevel="Level2"
                SemanticProperties.Description="Welcome to dot net Multi platform App U I"
                FontSize="18"
                HorizontalOptions="Center" />

            <Entry
                x:Name="IncrementInput"
                Placeholder="1"
                HorizontalOptions="Center" />

            <Button
                x:Name="CounterBtn"
                Text="Click me"
                SemanticProperties.Hint="Counts the number of times you click"
```

```
            Clicked="OnCounterClicked"
            HorizontalOptions="Center" />

        </VerticalStackLayout>
    </ScrollView>
</ContentPage>
```

Это поле добавляет возможность изменения шага, на который увеличивается значение счетчика при нажатии кнопки. Чтобы все заработало, заменим содержимое файла MainPage.xaml.cs следующим:

```
namespace SampleMauiApp;
public partial class MainPage : ContentPage
{
    int count = 0;
    public MainPage()
    {
        InitializeComponent();
    }
    private void OnCounterClicked(object sender, EventArgs e)
    {
        var increment = int.Parse(IncrementInput.Text);
        count += increment;
        if (count == 1)
            CounterBtn.Text = $"Clicked {count} time";
        else
            CounterBtn.Text = $"Clicked {count} times";
        SemanticScreenReader.Announce(CounterBtn.Text);
    }
}
```

Как мы видим, в результате свойство `IncrementInput` стало доступно в нашем коде программной части. То же самое имя мы указали в атрибуте `x.Name` добавленного элемента `Entry`. Мы интерпретируем содержимое этого элемента как целое число (`int`). Затем это число служит в качестве приращения.

Мы лишь вкратце коснулись синтаксиса MAUI XAML. Теперь давайте рассмотрим элементы, доступные в XAML, более подробно.

Справочные данные по MAUI XAML

Элементы XAML, используемые в MAUI, можно условно разделить на страницы, компоновки и представления, последние из которых также принято называть элементами управления. Мы рассмотрим все категории, начиная со страниц, которые перечислены в табл. 4.1.

Далее мы поговорим о компоновках. В контексте MAUI компоновка — это элемент управления, который располагается внутри элемента управления страницы. Его назначение — выступать в качестве контейнера для представлений. В табл. 4.2 представлены компоновки, которые поддерживает MAUI.

Таблица 4.1. *Страницы MAUI XAML*

Тип страницы	Описание
`ContentPage`	Самый простой тип, предусматривающий отображение одной страницы
`FlyoutPage`	Управление двумя связанными страницами: общая страница, представляющая элементы, и страница, содержащая подробную информацию об отдельном элементе
`NavigationPage`	Страница, представляющая иерархию навигации
`TabbedPage`	Страница, содержащая ряд страниц, представленных вкладками

Таблица 4.2. *Компоновки MAUI XAML*

Тип компоновки	Описание
`AbsoluteLayout`	Помещает каждый вложенный элемент в определенное место
`BindableLayout`	Может автоматически генерировать свое содержимое, привязывая его к коллекции элементов
`FlexLayout`	Позволяет размещать вложенные элементы рядом друг с другом или обертывать их различными вариантами стилей
`Grid`	Компоновка табличного типа, в которой вложенные элементы управления располагаются в строках и столбцах
`HorizontalStackLayout`	Размещает вложенные элементы по горизонтали
`StackLayout`	Размещает вложенные элементы по горизонтали или вертикали
`VerticalStackLayout`	Размещает вложенные элементы по вертикали

И наконец, рассмотрим отдельные представления, которые поддерживает MAUI (табл. 4.3).

Таблица 4.3. *Некоторые представления MAUI XAML*

Тип представления	Описание
`ActivityIndicator`	Использует анимацию, показывающую, что приложение занято
`BlazorWebView`	Размещает приложение Blazor внутри MAUI
`Border`	Рисует границы либо фон
`BoxView`	Рисует квадраты либо прямоугольники
`Button`	Кнопка, которую можно нажать
`CarouselView`	Отображает прокручиваемую коллекцию элементов
`CheckBox`	Флажок, который можно установить
`CollectionView`	Отображает прокручиваемую коллекцию элементов, основанную на параметрах компоновки
`ContentView`	Позволяет создавать пользовательские многократно используемые элементы управления

Таблица 4.3 *(окончание)*

Тип представления	Описание
DatePicker	Позволяет выбрать конкретную дату
Editor	Принимает многострочный текстовый ввод
Ellipse	Рисует эллипс либо окружность
Entry	Принимает однострочный текстовый ввод

В .NET MAUI существует множество представлений, поэтому для их описания может потребоваться несколько таблиц. В табл. 4.4 приведены дополнительные представления.

Таблица 4.4. *Дополнительные представления MAUI XAML*

Тип представления	Описание
Frame	Обрамляет элементы компоновки настраиваемой рамкой
GraphicsView	Холст для рисования двумерной графики
Image	Отображает изображение
ImageButton	Изображение, при щелчке на котором может выполняться какое-либо действие
IndicatorView	Работает совместно с представлением CarouselView и отображает число элементов на нем
Label	Отображает текст, доступный только для чтения
Line	Отображает линию
ListView	Отображает прокручиваемый список выбираемых элементов
Path	Отображает сложные криволинейные формы
Picker	Отображает список элементов, из которого можно выбрать один элемент
Polygon	Отображает произвольный двумерный многоугольник
Polyline	Отображает ломаную линию с несколькими вершинами
ProgressBar	Отображает ход выполнения некоторого длительного действия
RadioButton	Позволяет пользователю делать выбор между несколькими значениями
Rectangle	Отображает прямоугольник либо квадрат
RefreshView	Включает функцию "потянуть для обновления" для прокручиваемого содержимого
RoundRectangle	Отображает прямоугольник или квадрат со скругленными углами
ScrollView	Элемент-контейнер, позволяющий прокручивать вложенные в него элементы
SearchBar	Реализует функцию поиска
Slider	Позволяет пользователю выбрать дробное числовое значение из заданного диапазона

Таблица 4.4 *(окончание)*

Тип представления	Описание
`Stepper`	Позволяет пользователю выбрать дробное числовое значение из числа заданных величин
`SwipeView`	Реализует жест прокрутки
`Switch`	Двухпозиционный переключатель (вкл./выкл.)
`TableView`	Представляет таблицу
`TimePicker`	Позволяет выбрать конкретное время
`WebView`	Позволяет отобразить любой произвольный HTML-код, который может быть нормально отображен в браузере

Однако XAML — не единственный способ построения представлений в MAUI. Поскольку каждый из этих элементов представлен в коде классом, пользовательские интерфейсы можно писать на чистом C#. Это можно сделать и в Xamarin.Forms. Однако, поскольку MAUI полностью интегрирован в текущую версию .NET, появился и другой способ написания пользовательского интерфейса, недоступный в Xamarin.Forms, — Blazor.

Работа с Blazor на платформе .NET MAUI

Blazor — это набор технологий, позволяющих разработчикам компилировать .NET-код в формат WebAssembly, который представляет собой двоичный код, способный работать в браузерах. Преимущества Blazor многочисленны, начиная с возможности писать клиентский код на C# и заканчивая повышенной производительностью при выполнении скомпилированного низкоуровневого кода. Поскольку и MAUI, и Blazor полностью интегрированы в .NET, эти две технологии можно объединить. Сейчас вы узнаете, как это сделать.

Зависимости Blazor можно добавить в MAUI-проект задним числом, однако для MAUI-приложений с пользовательским интерфейсом Blazor существует шаблон проекта. Этот шаблон обеспечивает доступ к Blazor. На компьютере, используемом для разработки, должны быть включены базовые рабочие нагрузки веб-разработки MAUI и ASP.NET. Добавление последнего компонента обусловлено тем, что Blazor — это в первую очередь технология веб-разработки.

Теперь создадим проект MAUI с пользовательским интерфейсом Blazor, выполнив следующую команду:

```
dotnet new maui-blazor -o BlazorMauiApp
```

Теперь мы можем посмотреть на структуру проекта (рис. 4.6).

Здесь присутствуют все стандартные компоненты MAUI, включая App.xaml, MauiProgram.cs и MainPage.xaml. Но на этот раз файл MainPage.xaml содержит в своем коде разметки компонент `BlazorView`. Также имеются файлы с расширением RAZOR, относящиеся к технологии Blazor. Здесь же находится папка wwwroot

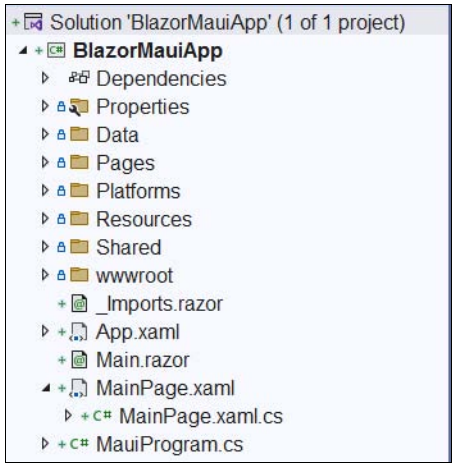


Рис. 4.6. Структура проекта Blazor MAUI

с файлами JavaScript, HTML и CSS, которые обычно встречаются в веб-приложениях.

Таким образом, компоненты MAUI выступают лишь в качестве основы для нашего приложения. Так как `BlazorView` является доминирующим компонентом пользовательского интерфейса, создание всего пользовательского интерфейса делегируется Blazor.

Более подробно мы рассмотрим технологию Blazor в *главе 7*. А сейчас внесем некоторые изменения в существующий код и посмотрим, как это отразится на создаваемом приложении. Для этого найдем файл Counter.razor в папке Pages. Эта страница похожа на стандартный пользовательский интерфейс Blazor, поскольку содержит счетчик и кнопку для увеличения его значения. И, как и раньше, мы добавим возможность изменять значение шага, привязав его к переменной. Таким образом, содержимое файла Counter.razor будет выглядеть следующим образом:

```
@page "/counter"
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<input type="number" @bind="increment" />
<button class="btn btn-primary" @onclick="IncrementCount">Click me</ button>
@code {
    private int? increment = 1;
    private int currentCount = 0;
private void IncrementCount()
    {
        currentCount += increment ?? 0;
    }
}
```

Позже мы запустим этот код как на настольной, так и на мобильной платформе. Но сейчас рассмотрим еще одну важную особенность MAUI — поддержку различных архитектурных паттернов.

Архитектурные паттерны MAUI

В MAUI можно просто использовать стандартный паттерн программирования, поставляемый в составе шаблонов проектов. Но как только ваше приложение станет более сложным, вам, вероятно, придется перейти на устоявшиеся архитектурные паттерны. В противном случае ваш код станет трудно поддерживать.

Существуют три основных архитектурных паттерна, поддерживаемых MAUI: Model-View-ViewModel (MVVM), ReactiveUI (RxUI) и Model-View-Update (MVU). MVVM — это паттерн, который доступен в исходной конфигурации MAUI, поэтому мы рассмотрим пример его реализации. Два других паттерна подключаются к MAUI с помощью сторонних библиотек. Но мы все же дадим обзор каждого из них.

Model-View-ViewModel

Паттерн проектирования MVVM состоит из трех основных компонентов: Model (модель), View (представление) и ViewModel (объект модель-представление).

- Model — это компонент, представляющий некоторые внутренние данные, например запись в базе данных.
- View — это отдельная страница пользовательского интерфейса.
- ViewModel — это объект, содержащий данные, которые непосредственно привязаны к элементам управления в конкретном представлении.

Чтобы показать пример реализации MVVM, создадим новый проект MAUI, выполнив следующую команду:

```
dotnet new maui -o MvvmMauiApp
```

Затем поместим в папку проекта файл MainPageViewModel.cs и наполним его следующим содержимым:

```
namespace MvvmMauiApp
{
    public class MainPageViewModel
    {
        public int IncrementBy { get; set; } = 1;
    }
}
```

По сути, мы создаем приложение, логика которого такая же, как и у предыдущего приложения. Но на этот раз значение шага будет находиться внутри модели представления, которая привязывается непосредственно к элементу управления в пользовательском интерфейсе. Для того чтобы это работало, нам необходимо добавить объект `ViewModel` в конструктор класса `MainPage`. Затем мы сопоставим этот объект `ViewModel` свойству `BindingContext`:

```
namespace MvvmMauiApp;
public partial class MainPage : ContentPage
```

```
{
int count = 0;
public MainPage(MainPageViewModel viewModel)
{
    BindingContext = viewModel;
    InitializeComponent();
}
private void OnCounterClicked(object sender, EventArgs e)
{
count += ((MainPageViewModel)BindingContext).IncrementBy;
if (count == 1)
    CounterBtn.Text = $"Clicked {count} time";
else
    CounterBtn.Text = $"Clicked {count} times";
SemanticScreenReader.Announce(CounterBtn.Text);
}
}
```

Теперь нам нужно изменить разметку внутри файла MainPage.xaml. Сначала мы заменим открывающий элемент `ContentPage` на следующий:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MvvmMauiApp"
    x:Class="MvvmMauiApp.MainPage">
    <ContentPage.BindingContext>
        <local:MainPageViewModel />
    </ContentPage.BindingContext>
```

Здесь мы указали тип, который будет использоваться в качестве `BindingContext`. Теперь добавим следующий элемент непосредственно перед элементом разметки `Button`:

```
<Entry
    x:Name="IncrementInput"
        Text="{Binding IncrementBy}"
        HorizontalOptions="Center" />
```

Текст в этом поле теперь привязан непосредственно к свойству `IncrementBy` класса `MainPageViewModel`. Но поскольку теперь этот класс внедряется в элемент `MainPage` через его конструктор, нам необходимо зарегистрировать оба этих класса в контейнере внедрения зависимостей. Для этого откроем файл MauiProgram.cs и добавим следующие строки непосредственно перед возвращением `builder.Build()`:

```
builder.Services.AddTransient<MainPage>();
builder.Services.AddTransient<MainPageViewModel>();
```

Теперь наше приложение должно быть идентично по своей функциональности первому приложению, которое мы создали. Но на этот раз взаимодействие между

пользовательским интерфейсом и бэкендом осуществляется через привязку объекта `ViewModel`.

Теперь мы кратко рассмотрим архитектурные паттерны, которые можно применять в приложениях MAUI за счет доступа к сторонним платформам.

Паттерны, поддерживаемые MAUI через сторонние платформы

Несмотря на то что эти паттерны недоступны в исходной конфигурации MAUI, они могут значительно облегчить жизнь разработчикам, использующим MAUI. Поэтому их стоит рассмотреть.

ReactiveUI

Паттерн ReactiveUI сочетает в себе MVVM и реактивное программирование. Если вы, как разработчик, уже встречались с MVVM, то увидите много знакомых концепций, таких как ViewModel и привязка данных. Однако этот паттерн также добавляет множество приемов для быстрого и эффективного обновления пользовательского интерфейса, что делает его гораздо более отзывчивым, чем при использовании других подходов.

В MAUI паттерн ReactiveUI становится доступным при установке пакета NuGet ReactiveUI.Maui. Документация по API доступна по следующему URL-адресу: **https://www.reactiveui.net/api/reactiveui.maui/**.

Model-View-Update

Паттерн MVU стал популярен благодаря платформам и библиотекам для веб-разработки, таким как React. Как и ReactiveUI, он отлично подходит для создания быстро реагирующих пользовательских интерфейсов. Однако его структура совершенно иная.

Как следует из названия, паттерн состоит из трех основных компонентов: View (представление), Model (модель) и Update (обновление).

View представляет собой пользовательский интерфейс и все его данные, т. е. аналогичен View и ViewModel из MVVM. Model отражает текущее состояние приложения. A Update — это функция, создающая новую копию состояния приложения, т. е. Model.

Самый простой способ подключить MVU в MAUI — это использовать библиотеку Comet, которая представлена пакетом NuGet Clancy.Comet. Репозиторий кода пакета можно найти по следующей ссылке: **https://github.com/dotnet/Comet**.

Теперь мы рассмотрим процесс запуска и публикации приложений MAUI на конкретных платформах. Для начала разберем процесс создания настольных приложений под Windows.

Использование MAUI для создания настольных приложений

MAUI доступен на двух настольных операционных системах — Mac и Windows. Приложения MAUI можно создавать и для Linux. Но эта возможность реализована сообществом и официально компанией Microsoft не поддерживается. Теперь мы обсудим процесс запуска и публикации приложения MAUI на ПК с ОС Windows.

Подготовка среды разработки настольных приложений

Прежде чем приступить к созданию и запуску приложения MAUI под Windows, необходимо включить режим разработчика, который можно найти в настройках. Если вы попытаетесь запустить приложение из среды разработки, то вам будет предложено включить режим разработчика. На рис. 4.7 показано, как выглядит это предложение в Visual Studio.

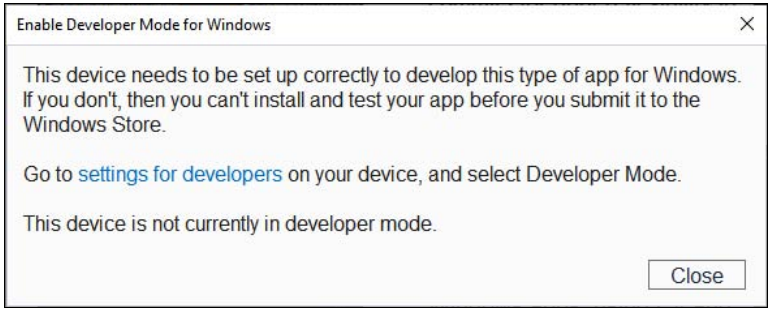


Рис. 4.7. Запрос на включение режима разработчика в Windows

Теперь ваша среда готова к запуску приложений MAUI. Первое, что мы рассмотрим, — запуск приложения MAUI в режиме отладки.

Запуск настольного приложения в режиме отладки

Как и в случае с приложениями любого другого типа, разработчику необходимо иметь возможность отлаживать кодовую базу MAUI и определять в ней точки останова. Процесс работы с приложениями MAUI идентичен работе с приложениями любого другого типа.

Если вы работаете с приложением MAUI под Windows, то профиль Windows должен быть включен по умолчанию. Таким образом, запустить приложение можно будет, либо выполнив соответствующую команду из .NET CLI, либо нажав кнопку **Run** (Выполнить) в IDE.

Если мы запустим приложение SampleMauiApp или MvvmMauiApp, которое мы создали ранее, пользовательский интерфейс будет выглядеть как на рис. 4.8.

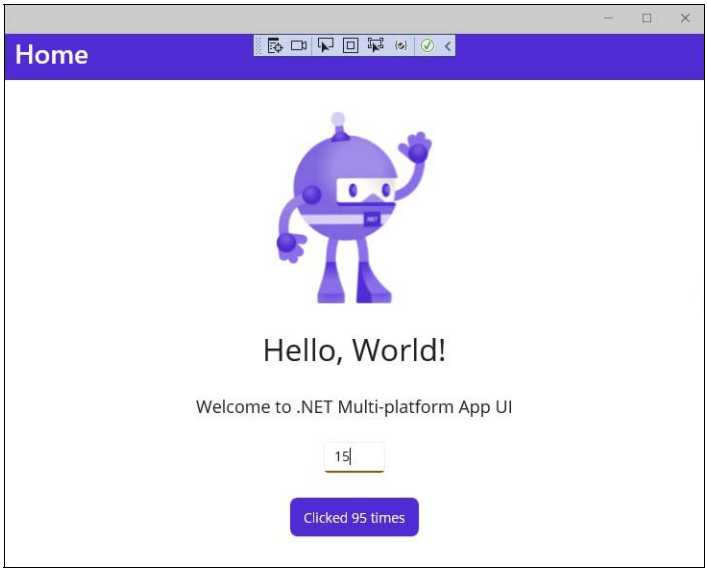


Рис. 4.8. Пользовательский интерфейс MAUI

Если мы запустим приложение BlazorMauiApp и перейдем к компоненту `Counter` (Счетчик), следует ожидать появления пользовательского интерфейса, показанного на рис. 4.9.

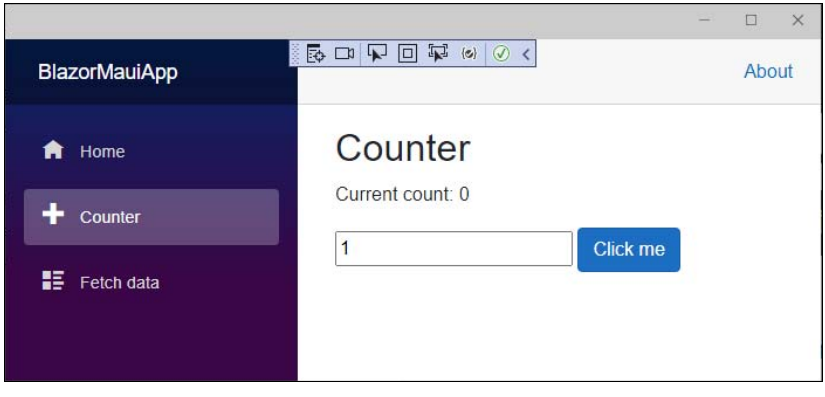


Рис. 4.9. Пользовательский интерфейс приложения Blazor MAUI

На этом завершим обзор процесса запуска MAUI-приложения на настольном ПК и рассмотрим, как опубликовать это приложение, чтобы его можно было развернуть на других устройствах.

Публикация настольного приложения

Если вы хотите разместить любое Windows-приложение в официальном магазине приложений или иным образом распространять его в профессиональной среде, это приложение должно быть подписано сертификатом. Иначе операционная система

выдаст предупреждение о том, что приложение выпущено неизвестным издателем. Однако это не обязательно, если вы хотите создать пакет развертывания исключительно для целей оценки и тестирования.

Как и в случае с .NET-приложениями любого другого типа, наиболее простой способ публикации приложения MAUI — через IDE. Это можно сделать и через CLI, но процедура будет более сложной. Чтобы опубликовать приложение из Visual Studio, достаточно выбрать основной проект приложения на панели **Solution Explorer** (Обозреватель решений), щелкнуть на нем правой кнопкой мыши и выбрать пункт **Publish** (Опубликовать). Затем вам будет предложено выбрать способ распространения, как показано на рис. 4.10.

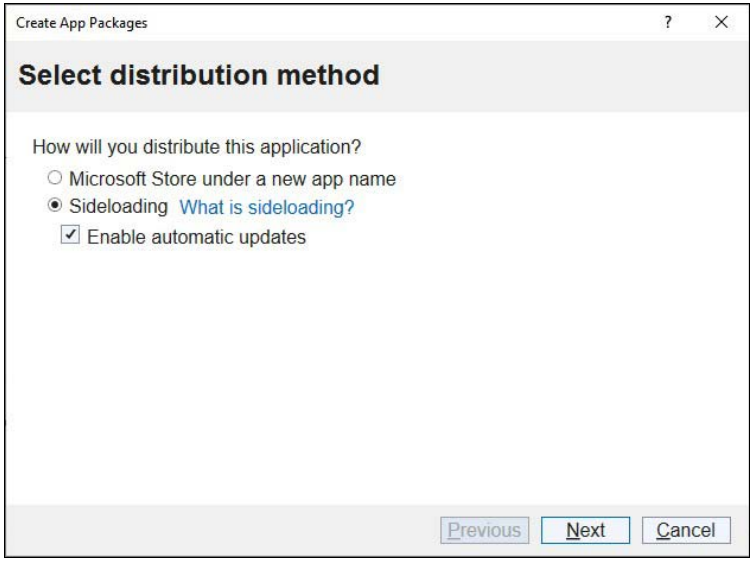


Рис. 4.10. Выбор способа распространения при публикации MAUI-приложения в Windows

Затем вам будет предложено выбрать метод подписания пакета, который необходим, чтобы сделать пакет готовым к использованию в рабочей среде. После этого вы сможете указать архитектуру процессора для вашего пакета. Далее можно просто выполнить все указанные шаги для завершения процесса формирования пакета.

Мы обсудили процесс запуска и развертывания приложений MAUI на настольном компьютере с ОС Windows. Далее рассмотрим, как создавать мобильные MAUI-приложения для Android.

Использование MAUI для создания мобильных приложений

Создавать MAUI-приложения для Android можно на любом компьютере с любой операционной системой, если он поддерживает либо возможность подключения к ней реального Android-устройства и запуска его в режиме разработчика, либо ис-

пользование эмуляторов. Итак, давайте научимся настраивать нашу среду разработки, чтобы начать создавать в ней мобильные приложения.

Подготовка среды разработки мобильных приложений

Самый простой способ протестировать мобильное приложение перед его публикацией — использовать эмулятор, т. е. виртуальную машину, в точности имитирующую реальное мобильное устройство. Проще всего создать такую систему через IDE. Например, в Visual Studio можно выбрать вкладку **Tools** (Средства), затем выбрать пункт **Android** и нажать кнопку **Android SDK Manager** (Диспетчер SDK Android), как показано на рис. 4.11.

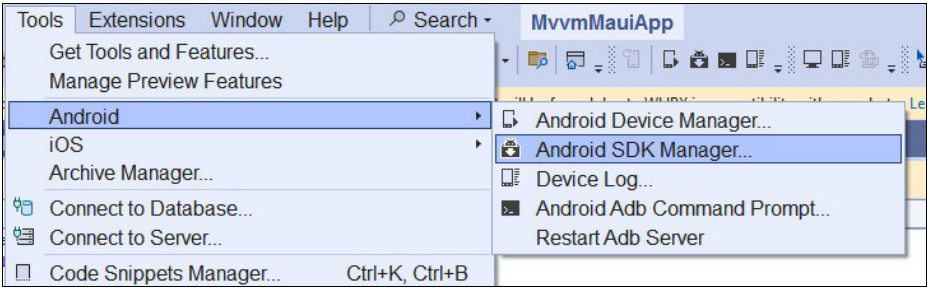


Рис. 4.11. Пункт меню **Диспетчер SDK Android** в Visual Studio

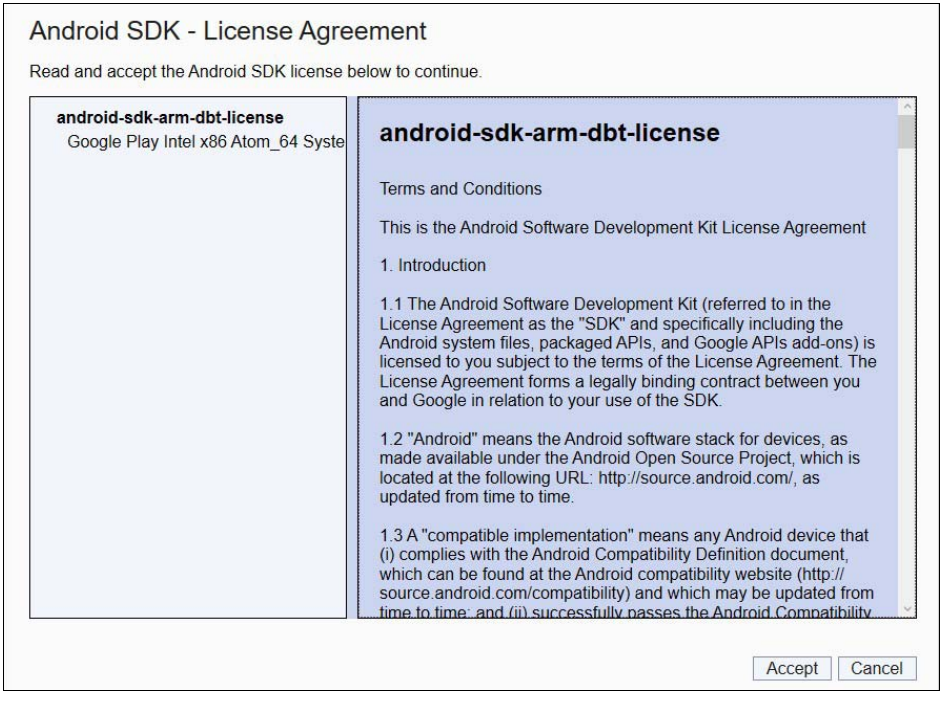


Рис. 4.12. Лицензионное соглашение для Android SDK

После этого можно выбрать любой эмулятор из списка и установить его. При выборе установки будет предложено принять лицензионное соглашение, как показано на рис. 4.12.

Для установки эмуляторов потребуется некоторое время. После этого вы сможете запускать на них наши приложения.

Запуск мобильного приложения на эмуляторе

Запустить приложение на эмуляторе можно с помощью меню **Run** (Запуск) в IDE, как показано на рис. 4.13.

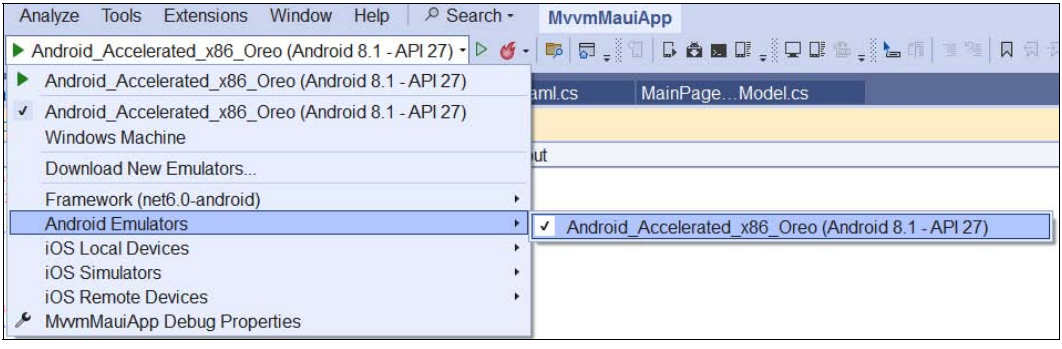


Рис. 4.13. Выбор эмулятора Android в меню **Run** (Запуск) Visual Studio

Остальной процесс не отличается от запуска в Visual Studio приложения любого другого типа. После нажатия кнопки **Run** (Запуск) при выбранном эмуляторе Android выполняется запуск эмулятора, на котором будет запущено ваше приложение. Во время его работы вы сможете задавать точки останова в коде.

На этом мы завершаем тему запуска MAUI-приложений для Android на ПК и далее рассмотрим процесс публикации приложения, который позволит распространять его через магазины приложений.

Публикация мобильного приложения

Процесс публикации MAUI-приложений для Android мало чем отличается от их публикации для Windows. Если в качестве цели сборки выбрана ОС Android, а в качестве конфигурации сборки — Release, то достаточно щелкнуть правой кнопкой мыши на проекте в **Solution Explorer** (Обозреватель решений) и нажать кнопку **Publish** (Опубликовать). После этого откроется окно публикации для Android, как показано на рис. 4.14.

В результате в папке bin/Release/net7.0-android будет создан APK-файл, с помощью которого данное приложение можно установить на Android-устройство.

В качестве альтернативы можно выполнить следующую команду из папки проекта через CLI:

```
dotnet build -f net7.0-android -c:Release
```

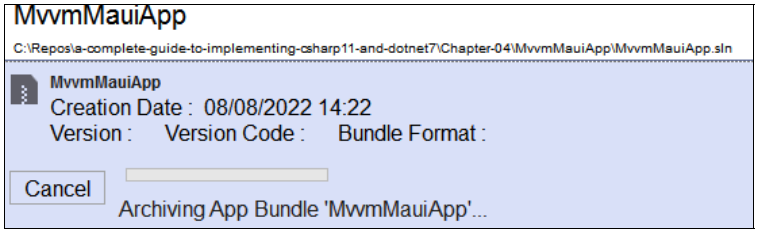


Рис. 4.14. Публикация MAUI-приложения для Android

На этом завершим описание процесса публикации MAUI-приложений для Windows и Android и далее рассмотрим процесс запуска и публикации MAUI-приложений для Mac и iOS.

Ограничения разработки для Mac OS и iOS

Когда мы описывали процесс создания MAUI-приложений для Windows, вы, вероятно, заметили, что все его этапы подразумевают, что мы делаем это на Windows-устройстве. Не существует такого понятия, как эмулятор Windows. Если попытаться запустить Windows-приложение на машине, работающей под управлением другой операционной системы, оно просто не будет работать.

Аналогичное ограничение существует и для приложений для Mac и iOS. Для их запуска требуется устройство Apple. Но даже при наличии эмуляторов iOS их можно запускать только на другом устройстве Apple. Это связано с тем, что для размещения своих приложений в любом из магазинов приложений Apple требуется учетная запись Apple. Но это не единственная причина. Для эмуляции iOS на вашем рабочем компьютере потребуются специальные инструменты. Apple отказывается публиковать какие-либо из этих инструментов для любой операционной системы, кроме собственной.

Дополнительные инструменты, необходимые для публикации приложений для iOS

Обычно для создания стандартных нативных приложений для iOS требуется IDE под названием XCode. Эта среда разработки доступна только на Mac. К сожалению, эмуляция iOS-устройств невозможна без инструментов, которые предоставляет данная среда разработки. Кроме того, нельзя создать нативный пакет для развертывания своего приложения в iOS. Таким образом, даже если вы собираетесь разрабатывать приложения для iOS с помощью MAUI, вам все равно придется установить XCode, который можно найти по следующей ссылке: **https://developer.apple.com/xcode/**.

После установки XCode необходимо подключить к нему существующую учетную запись разработчика или создать новую. Для этого необходимо открыть окно **Preferences** (Настройки) через меню **XCode**. Затем следует выбрать вкладку **Accounts** (Учетные записи). Для добавления новой учетной записи в появившемся

диалоговом окне следует нажать кнопку со знаком "плюс" (+). После этого можно привязать свой Apple ID и следовать дальнейшим указаниям.

Затем в файл проекта необходимо добавить некоторые метаданные и подписать пакет. Инструмент публикации должен уведомить вас, если каких-либо требуемых компонентов не хватает. Аналогичным образом можно подключить Visual Studio for Mac к Mac-хосту для сборки, чтобы иметь возможность использовать IDE для работы с кодом MAUI. Это понадобится вам, если вы все же хотите задействовать все возможности IDE, поскольку XCode не был предназначен для работы с .NET и C#.

Затем, чтобы опубликовать приложение для iOS, необходимо выполнить следующую команду из папки проекта:

```
dotnet publish -f net7.0-ios -c Release
```

После этого пакет приложения будет помещен в папку bin вашего проекта.

Менее строгие требования к приложениям для Mac OS

Если вы хотите опубликовать MAUI-приложение для Mac OS, то ограничения будут несколько мягче. Вам не понадобится XCode или какие-либо другие дополнительные инструменты. Вам даже не потребуется ничего помимо стандартной .NET IDE или редактора кода. Однако если вы собираетесь опубликовать свое приложение в официальных репозиториях приложений, то все равно будет нужна учетная запись разработчика и подпись пакета.

Mac OS позволит загружать и устанавливать пакеты без подписи, но при этом будет выдаваться предупреждение о безопасности. На этом мы завершаем обзор .NET MAUI.

Заключение

Из этой главы вы узнали, что .NET MAUI — это развитие Xamarin.Forms, и что этот инструмент позволяет разработчикам писать как настольные, так и мобильные приложения. В отличие от своего предшественника, MAUI полностью интегрирован с .NET 7 и имеет доступ ко всем новейшим возможностям языка C# 11. Кроме того, структура проектов в нем гораздо удобнее, чем в Xamarin.Forms. Он также поддерживает полный спектр мобильных и настольных операционных систем.

Поскольку .NET MAUI полностью интегрирован с .NET 7, его можно объединить с любыми библиотеками и платформами .NET 7. Например, для создания пользовательского интерфейса с его помощью можно использовать Blazor WebAssembly.

.NET MAUI поддерживает три основных архитектурных паттерна — MVVM, RxUI и MVU. Последний паттерн позволяет разработчикам, не имевшим ранее опыта работы с MAUI, быстро освоить его, поскольку этот паттерн весьма легок для понимания.

Особое удобство MAUI заключается в том, что вы можете написать свое приложение на любом компьютере и скомпилировать его для любой другой поддерживае-

мой операционной системы. Однако здесь есть одна оговорка. Если вы планируете разрабатывать приложения для Mac OS или iOS, то вам понадобится Mac. Это связано с тем, что Apple не позволяет разработчикам публиковать программное обеспечение, которое было разработано не на Mac, и не предоставляет инструментарий для других операционных систем.

В следующей главе вы познакомитесь с последней версией инструмента Entity Framework Core, который применяется для доступа к структурным данным SQL путем их абстрагирования в коде.

Что следует помнить

- .NET MAUI может использоваться для создания мобильных и настольных приложений.
- MAUI — это развитие Xamarin.Forms.
- MAUI полностью интегрирован с .NET 7 и имеет полный доступ к новейшим возможностям языка.
- Основной язык разметки для пользовательских интерфейсов MAUI — XAML.
- Создать пользовательский интерфейс с помощью MAUI можно на основе других технологий, в том числе Blazor WebAssembly.
- MAUI может использовать один из следующих архитектурных паттернов: MVVM, RxUI и MVU.
- Приложения, созданные для Mac OS или iOS, могут быть опубликованы только с устройства Mac из-за ограничений, накладываемых компанией Apple.

Контрольные вопросы

1. Под какими операционными системами можно запускать приложения MAUI?

а) Android;

б) iOS;

в) Windows;

г) Все вышеперечисленное.

2. В чем ключевое различие между MAUI и Xamarin.Forms?

а) Xamarin.Forms не имеет доступа к новейшим возможностям языка C#, в то время как MAUI имеет такой доступ;

б) Xamarin.Forms не может запускать Blazor WebAssembly, а MAUI — может;

в) Xamarin.Forms предназначался в первую очередь для мобильных приложений, в то время как MAUI предназначен как для мобильных, так и для настольных приложений;

г) Все вышеперечисленное.

3. **Какой архитектурный паттерн не поддерживается в MAUI?**

   а) ReactiveUI;

   б) MVVM;

   в) MVC;

   г) MVU.

4. **Какой основной язык используется в MAUI для определения компоновок пользовательского интерфейса?**

   а) XML;

   б) XAML;

   в) JSON;

   г) HTML.

Ответы

1. г.
2. г.
3. в.
4. б.

Основные термины

- **MAUI** — многоплатформенный пользовательский интерфейс; фреймворк для разработки нативных мобильных и настольных приложений на платформе .NET.
- **Xamarin.Forms** — предшественник MAUI, который был специально разработан для создания кросс-платформенных мобильных приложений.
- **MVVM** — **Model-View-ViewModel**; архитектурный паттерн, состоящий из компонента пользовательского интерфейса (View), абстракции данных бэкенда (model) и данных, наполняющих пользовательский интерфейс (ViewModel).
- **ReativeUI** — фреймворк, добавляющий компоненты реактивного программирования поверх MVVM.
- **MVU** — архитектура Model-View-Update, известная также как архитектура Elm, представляет собой архитектурный паттерн, состоящий из пользовательского интерфейса (View), абстракции данных (Model) и компонента, обновляющего модель при выполнении некоторого действия в пользовательском интерфейсе (Update).

Глава 5

Доступ к базам данных с помощью Entity Framework 7

Введение

Entity Framework Core (EF Core) — это инструмент объектно-реляционного отображения (Object-Relational Mapper, ORM), который используется в платформе .NET. Цель ORM — обеспечить взаимодействие кода приложений с реляционными базами данных путем абстрагирования объектов БД в виде объектов кода (классов, записей и т. д.). Существуют и другие ORM для .NET, но они создаются и поддерживаются сторонними разработчиками. EF Core разрабатывает основная группа разработчиков Microsoft наряду с остальными компонентами .NET.

В этой главе мы рассмотрим основные возможности EF Core. К концу главы вы научитесь использовать этот инструмент в коде приложения для работы с данными в БД. Мы также рассмотрим сценарии, предусматривающие написание кода EF Core на основе существующей БД, и расскажем о процессе создания новой БД с объектами EF Core.

Кроме того, для полного понимания ORM необходимо знать основы реляционных БД. Поэтому в этой главе будет дан краткий обзор этой концепции.

Наконец, т. к. в рамках данной книги рассматриваются новейшие возможности C# 11 и .NET 7, мы опишем, какие новые функции были добавлены в седьмую версию EF Core, правильное название которой EF7, а не EF Core 7. Таким образом, вы не только изучите основы EF Core, но и научитесь использовать его передовые возможности.

Структура

В этой главе мы рассмотрим новейшие возможности Entity Framework 7, которые собраны в следующие темы:

- знакомство с основами реляционных БД;
- введение в Entity Framework 7;
- подход "сначала код" в EF7;

- подход "сначала база данных" в EF7;
- новейшие функции EF7.

Цели

К концу этой главы вы узнаете, как интегрировать базовое приложение .NET с БД с помощью EF7. Вы также научитесь применять новейшие возможности EF7.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из перечисленных компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Знакомство с основами реляционных БД

Поскольку EF7 — это инструмент объектно-реляционного отображения (Object-Relational Mapping, ORM), предназначенный в первую очередь для взаимодействия с реляционными БД, то прежде чем приступать к изучению его работы, необходимо понять, как функционируют реляционные БД. К счастью, для того чтобы начать использовать EF7, не нужно быть экспертом в области баз данных. Но понимание основ, безусловно, необходимо.

Обзор систем управления реляционными БД и SQL

Реляционные БД строятся на основе реляционных систем управления базами данных (РСУБД). Существует несколько различных технологий РСУБД, выпускаемых разными вендорами. Несмотря на принципиальные архитектурные различия, у них есть и общие черты. Любая РСУБД, независимо от вендора, имеет следующие компоненты:

- базы данных, состоящие из таблиц;
- каждая таблица имеет жесткую схему, т. е. состоит из фиксированного числа столбцов, и каждый из этих столбцов принимает данные только определенного типа (целочисленные, текстовые, двоичные/булевы и т. д.).
- таблицы ссылаются друг на друга с помощью так называемых внешних ключей, что обеспечивает взаимосвязь между ними;

- для получения данных, манипулирования ими и модификации структуры самой БД используется язык SQL.

Язык структурированных запросов (SQL) представляет собой универсальный язык управления базами данных. У разных вендоров РСУБД варианты (или диалекты) SQL различны, поэтому оператор, написанный для РСУБД определенного типа, не обязательно будет совместим с РСУБД другого типа. Основополагающие команды либо являются одинаковыми независимо от вендора, либо очень похожи друг на друга. Например, рассмотрим следующее SQL-выражение:

```
SELECT * FROM USERS
```

Любой, кто знаком с SQL, сможет определить, что делает это выражение. Ключевое слово `SELECT` указывает на то, что мы выбираем некоторые данные. Символ `*` означает, что мы выбираем все поля. В качестве альтернативы можно указать конкретные поля, которые требуется вернуть. Ключевое слово `FROM` служит для указания объекта, из которого мы хотим извлечь данные, обычно это таблица. `USERS` — это имя конкретного объекта.

Выражение SQL для получения данных обычно называют запросом. SQL-выражение для изменения, вставки или обновления данных называют командой.

Наиболее распространенные типы РСУБД:

- **SQL Server** — производится компанией Microsoft как патентованный продукт, предназначенный в основном для коммерческих проектов;
- **Oracle Database** — основная РСУБД, выпускаемая компанией Oracle как патентованный продукт, в основном для коммерческих проектов;
- **MySQL** — упрощенная бесплатная РСУБД с открытым исходным кодом, поддерживаемая компанией Oracle и часто используемая для некоммерческих проектов;
- **PostgreSQL** — бесплатная открытая РСУБД с расширенными возможностями, в основном для коммерческих проектов.

Теперь рассмотрим типичную структуру таблицы РСУБД.

Таблицы, отношения и нормализация

Поскольку данные в реляционных БД хранятся в таблицах, мы рассмотрим пример такой таблицы. Затем мы внесем в него некоторые изменения, чтобы выделить основные особенности РСУБД.

Представим, что у нас есть система, управляющая данными компании, владеющей несколькими заводами. В базе данных этой системы хранится информация о сотрудниках заводов, их зарплате, сменах, должностях и т. д. Первоначально мы можем сохранить эту информацию в одной таблице, которая будет иметь следующие столбцы, как показано в табл. 5.1.

Ничто не мешает хранить данные именно таким образом, однако с подобным подходом связано множество проблем.

Таблица 5.1. *Структура исходной плоской таблицы с информацией о сотрудниках*

Имя столбца	Тип данных
First Name (Имя)	Текст
Last Name (Фамилия)	Текст
Job Title (Должность)	Текст
Annual Salary (Годовой оклад)	Десятичное число
Date of Birth (Дата рождения)	Дата
Date of Joining the Company (Дата приема на работу)	Дата
Employment Location (Место работы)	Текст
Factory Name (Название завода)	Текст
Shift Start (Начало смены)	Время
Конец смены (Конец смены)	Время

- Во-первых, это просто плоская таблица, содержащая все данные о сотрудниках. Такую таблицу мы легко можем создать в Excel. Зачем тогда вообще создавать РСУБД?
- Во-вторых, хранение всей информации в одной таблице усложняет внесение любых изменений. Например, вполне вероятно, что число наименований должностей с соответствующими окладами ограничено. Кроме того, существует конечное число заводов, на которых работают сотрудники. Но что произойдет, если для одной из должностей будет увеличен связанный с ней оклад? А если закроется какой-либо завод и его сотрудников потребуется перевести на соседнее предприятие? Если данные о должности, окладе и месте работы хранятся в той же таблице, что и информация о сотруднике, то необходимо найти и обновить все записи, содержащие устаревшую информацию. Если у вас крупная организация с тысячами сотрудников, то этот процесс может занять много времени.
- Кроме того, в одной плоской таблице сложно хранить информацию о сменах сотрудников. В нашей таблице хранится информация о времени начала и окончания смены. Все выглядит нормально, если сотрудник работает в одну и ту же смену и если завод открыт только в определенные дни. Но что делать, если это не так? Что если завод работает круглосуточно и каждый сотрудник может совмещать работу в несколько смен? Или что делать, если один и тот же сотрудник работает на разных заводах? Такую информацию было бы очень сложно хранить в плоской таблице. Пришлось бы выделить для нее поле ввода произвольного текста, что усложнило бы запросы, либо ввести несколько дублирующих друг друга столбцов для нескольких пар значений времени начала и конца смены и дней работы в каждую смену.

Это иллюстрирует, пожалуй, самый очевидный недостаток хранения большого объема данных в плоской таблице. Но реляционная база данных может решить все

эти проблемы. Все, что нам нужно сделать, это превратить эту плоскую таблицу в несколько таблиц, связанных друг с другом. И именно это мы и сделаем далее.

Введение первичных ключей

Перед тем как разделить нашу таблицу, мы добавим в нее *первичный ключ* — столбец, который содержит уникальное значение для каждой строки. Это может быть также сочетание нескольких столбцов, но комбинация значений в каждой строке все равно должна быть уникальной.

Наличие первичного ключа имеет целый ряд преимуществ. Но основные из них следующие:

- первичный ключ является одновременно уникальным идентификатором записи и ее индексом, поэтому ускоряется поиск конкретной записи;
- первичный ключ одной таблицы может быть использован в качестве так называемого *внешнего ключа* другой таблицы, что позволяет установить связь между двумя таблицами.

Итак, что же мы можем использовать в качестве первичного ключа в нашей таблице? Теоретически это может быть сочетание имени и фамилии. Но тогда нет гарантии, что в каждом ряду мы получим уникальную комбинацию, т. к. существуют разные люди, носящие одно и то же имя. К имени и фамилии можно добавить дату рождения. Но и это не гарантирует уникальность. Кроме того, сочетание в первичном ключе данных разных типов усложнит управление.

Чтобы решить эту проблему, мы просто сделаем то, что обычно делается в такой ситуации — добавим еще один столбец, в котором будет храниться только первичный ключ. Это будет целочисленный столбец с автоматически генерируемыми последовательными значениями, поэтому каждая запись гарантированно будет иметь уникальное значение. Столбец такого типа обычно называют `идентификатором`. Ему можно дать имя `id` или `<имя таблицы>_id`, например `employee_id`. Далее мы применим методику, известную как нормализация, чтобы разбить одну таблицу на несколько.

Нормализация и отношения внешних ключей

Нормализация — это процесс удаления из таблицы избыточных и повторяющихся данных и переноса их в отдельные таблицы, на которые исходная таблица сможет ссылаться. Таким образом, вместо повторения каждое значение будет храниться в своей таблице. Если другой таблице потребуется обратиться к данному значению, то она обратится к индексу записи таблицы, в которой хранится нужное значение. Таким образом, если впоследствии потребуется изменить значение (например, название завода или должность), то это нужно будет сделать только в одном месте.

Теперь применим нормализацию к нашим данным. Поскольку у нас есть конечное число должностей, у каждой из которых свой годовой оклад, мы перенесем информацию о должностях в отдельную таблицу, структура которой приведена в табл. 5.2.

Далее создадим таблицу, содержащую информацию о заводах (табл. 5.3).

Таблица 5.2. *Структура таблицы Jobs (Должности)*

Имя столбца	Тип данных	Примечания
Job ID (ИД должности)	Целое число (идентификатор)	Первичный ключ
Job Title (Должность)	Текст	–
Compensation (Оклад)	Десятичное число	–

Таблица 5.3. *Структура таблицы Factories (Заводы)*

Имя столбца	Тип данных	Примечания
Factory ID (ИД завода)	Целое число (идентификатор)	Первичный ключ
Factory Name (Название завода)	Текст	–
Factory Location (Местонахождение завода)	Текст	–

Таблицей смен мы займемся позже. А пока вернемся к нашей таблице, содержащей оставшиеся данные о сотрудниках, которая теперь будет иметь структуру, показанную в табл. 5.4.

Таблица 5.4. *Структура таблицы Employees (Сотрудники)*

Имя столбца	Тип данных	Примечания
Employee ID (ИД сотрудника)	Целое число (идентификатор)	Первичный ключ
First Name (Имя)	Текст	–
Last Name (Фамилия)	Текст	–
Date of Birth (Дата рождения)	Дата	–
Date of Joining the Company (Дата приема на работу)	Дата	–
Job ID (ИД должности)	Целое число	Внешний ключ (соответствует первичному ключу таблицы Jobs (Должности))

Здесь показан первый пример связи между двумя таблицами посредством внешнего ключа. Столбец `Job ID` в таблице `Employees` ссылается на столбец `Job ID` в таблице `Jobs`. Здесь имеет место отношение "один ко многим" между должностью и сотрудниками, поскольку может быть несколько сотрудников с одинаковым названием должности. В предыдущем примере (да и в любом другом случае связи с использованием внешнего ключа) значение, которое вы помещаете в столбец `Job ID` таблицы `Employees`, должно совпадать со значением в столбце `Job ID` таблицы `Jobs`.

Наконец, добавим таблицу `Shifts`, структура которой приведена в табл. 5.5.

Теперь мы можем легко хранить достаточно подробную информацию о сменах, которые назначены сотрудникам. Также появилась возможность привязать сотруд-

Таблица 5.5. *Структура таблицы Shifts (Смены)*

Имя столбца	Тип данных	Примечания
Shift Id (ИД смены)	Целое число (идентификатор)	Первичный ключ
Weekday (День недели)	Целое число	Допустимое значение: от 1 до 7
Start Time (Время начала)	Время	–
End Time (Время окончания)	Время	–
Employee ID (ИД сотрудника)	Целое число	Внешний ключ (первичный ключ таблицы Employees (Сотрудники))
Factory ID (ИД завода)	Целое число	Внешний ключ (первичный ключ таблицы Factories (Заводы))

ника к нескольким заводам через таблицу `Shifts`. Таким образом, между сотрудниками и заводами теперь существует отношение "многие-ко-многим".

На этом обзор реляционных БД завершен. Тема эта, безусловно, гораздо шире, тем более что у каждой РСУБД есть свои нюансы. Но мы рассмотрели достаточно основных положений, чтобы понять, как работает EF7. Поэтому давайте настроим нашу среду для использования этого инструмента.

Введение в Entity Framework 7

Ранее мы кратко упомянули о том, что для поиска и манипулирования данными в реляционных БД применяется язык под названием SQL. Но если вы работаете с ORM, например с Entity Framework 7 (EF7), то в большинстве случаев вам не придется иметь дело с SQL напрямую. Таблицы и отношения будут представлены в коде объектами, которые в C# обычно являются либо классами, либо записями. ORM сделает за вас все необходимые сопоставления.

Для начала работы с EF7 понадобятся две вещи: инструменты командной строки EF и пакет NuGet, представляющий конкретный провайдер РСУБД.

EF7 поддерживает следующие провайдеры:

- Microsoft SQL Server — представлен пакетом NuGet Microsoft.EntityFrameworkCore.SqlServer;
- SQLight — представлен пакетом NuGet Microsoft.EntityFrameworkCore.Sqlite;
- Azure Cosmos DB — представлен пакетом NuGet Microsoft.EntityFrameworkCore.Cosmos;
- PostgreSQL — представлен пакетом NuGet Npgsql.EntityFrameworkCore.PostgreSQL;
- MySQL — представлен пакетом NuGet Pomelo.EntityFrameworkCore.MySql;
- In-memory database (База данных в памяти) — представлен пакетом NuGet Microsoft.EntityFrameworkCore.InMemory.

Не все эти провайдеры представляют собой реальные реляционные БД. Например, Azure Cosmos DB — это хранилище документов, представляющее другой тип БД. Точно так же БД In-memory Database не является настоящей базой данных. Но тем не менее реляционная БД — это основной тип баз данных, для работы с которым предназначен EF7.

Для установки инструментов командной строки EF можно выполнить в вашем терминале следующую команду:

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

Если эти инструменты уже установлены, то для их обновления до последней версии можно выполнить следующую команду:

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

EF7 можно использовать на любой ОС, способной работать с .NET. Однако Visual Studio 2022 для Windows предлагает наиболее богатую пользовательскую среду. Например, Visual Studio 2022 поставляется со встроенной версией SQL Server под названием `LocalDB`, поэтому нет необходимости устанавливать экземпляр SQL Server вручную. Его просто нужно включить с помощью программы Visual Studio Installer. Для этого достаточно открыть программу установки, нажать кнопку **Modify** (Изменить) рядом с экземпляром Visual Studio 2022 и выбрать установку рабочей нагрузки **Data storage and processing** (Хранение и обработка данных), как показано на рис. 5.1.

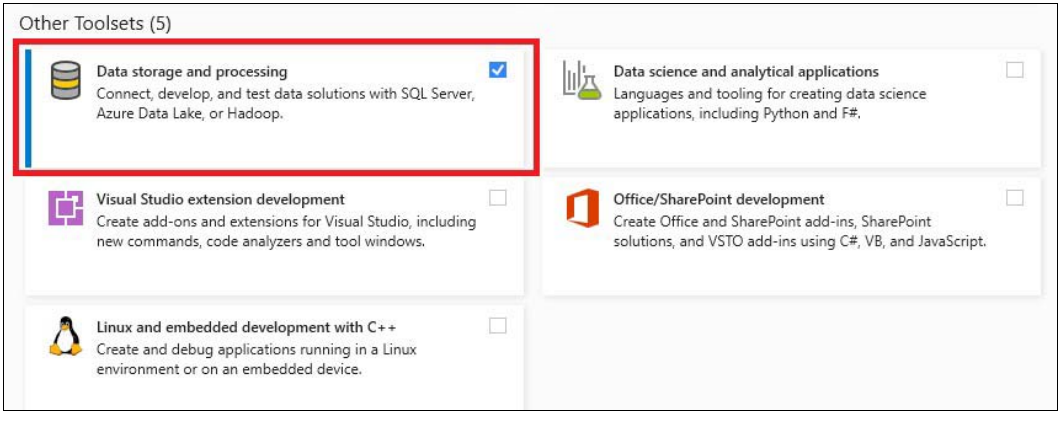


Рис. 5.1. Установка рабочей нагрузки для хранения и обработки данных

После установки мы сможем получить доступ к SQL Server, выбрав опцию **SQL Server Object Explorer** (Обозреватель объектов SQL Server) на вкладке **View** (Вид), как показано на рис. 5.2.

Это все, что нужно для начала работы с EF7. Теперь рассмотрим, как использовать EF7 в коде. Мы начнем с применения одной из самых мощных его возможностей — создания новой БД исключительно из кода. Такой подход известен под названием "сначала код".

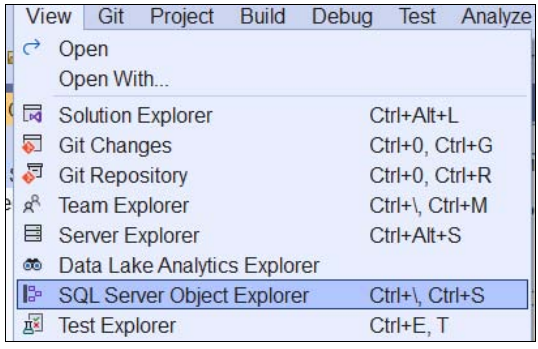


Рис. 5.2. Доступ к обозревателю объектов SQL Server

Подход "сначала код" в EF7

Подход "сначала код" не только дает возможность создать полноценную базу данных из кода, но и позволяет заполнить созданную базу исходными данными. Теперь мы рассмотрим шаги, необходимые для применения Entity Framework в вашем приложении, автоматического создания базы данных и заполнения ее некоторыми данными. Для начала рассмотрим основы EF7.

Добавление кода Entity Framework

Entity Framework широко используется в приложениях ASP.NET Core, построенных на основе паттерна Model-View-Controller (MVC). Более подробно мы рассмотрим приложения этого типа в *главе 6*. А сейчас создадим веб-приложение MVC с доступом к БД, выполнив следующую команду:

```
dotnet new mvc -o MvcDataApp
```

Далее мы перейдем в папку проекта MvcDataApp и выполним следующую команду для установки пакета NuGet, содержащего средства диагностики EF7:

```
dotnet add package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
```

Альтернативный путь — установка этого пакета через графический интерфейс среды разработки.

Далее мы установим следующий пакет NuGet, содержащий провайдер Entity Framework для SQL Server:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Теперь мы готовы к добавлению объектов, которые будут представлять таблицы нашей БД. Мы воспользуемся тем же решением для управления заводами, которое описывали ранее, когда рассказывали о нормализации БД.

Добавление объектов-сущностей

Первый объект, который мы добавим, будет представлять собой таблицу `Jobs`. Чтобы добавить этот объект, создадим в папке проекта Models файл Job.cs и наполним его следующим содержимым:

```
using System.ComponentModel.DataAnnotations.Schema;
namespace MvcDataApp.Models;
public record Job
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int JobId { get; set; }
    public string JobTitle { get; set; }
    [Column(TypeName = "decimal(8, 2)")]
    public decimal Compensation { get; set; }
    public ICollection<Employee> Employees { get; set; }
}
```

В качестве базового типа мы использовали `record` (Запись). Это ключевое слово было добавлено в C# 10. Запись похожа на класс, но в первую очередь предназначена для представления записи в БД. Однако этот код будет работать и с классом.

Данный код содержит некоторые атрибуты аннотаций, которые мы добавили для демонстрации возможностей EF7. Первый из них — `DatabaseGeneratedAttribute`, который мы поместили над свойством `JobId`. Этот атрибут не является строго необходимым, поскольку данное свойство будет распознано как столбец идентификатора первичного ключа на основе соглашений об именовании. Но мы поместили его туда, чтобы продемонстрировать, как можно обозначить любой произвольный столбец в качестве идентификатора первичного ключа.

Для десятичного типа данных в SQL-сервере требуется указать точность и масштаб. Точность — это общее число цифр, которое мы можем хранить. Масштаб — это число цифр после запятой. А чтобы задать эту информацию, мы добавили атрибут `Column`.

Объекты Entity Framework представляют отношения "один-ко-многим" в виде коллекций. В нашем случае, поскольку у нас есть отношение "один-ко-многим" между сущностью `Job` и записями `Employee`, в записи `Job` имеется коллекция объектов `Employee`.

Далее мы добавим определение записи о сотруднике. Создадим в папке Models файл Employee.cs и добавим в него следующий код:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace MvcDataApp.Models;
public record Employee
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int EmployeeId { get; set; }
    [StringLength(20)]
```

```
    public string FirstName { get; set; }
    [StringLength(20)]
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public DateTime StartDate { get; set; }
    public int JobId { get; set; }
    public Job { get; set; }
    public ICollection<Shift> Shifts { get; set; }

    [NotMapped]
    public string FullName => FirstName + " " + LastName;
}
```

В этом коде мы добавили еще несколько полезных атрибутов. Атрибут `stringLength` устанавливает ограничение на содержимое столбца БД, ограничивающее его определенным количеством символов. Атрибут `NotMapped` добавляется к свойствам, которые могут быть полезны в коде, но не должны представлять столбец БД.

Поскольку каждая сущность `Employee` может быть связана только с одной сущностью `Job`, мы добавили только одно свойство `Job`. Но т. к. одной сущности `Employee` могут быть сопоставлены несколько сущностей `Shift`, мы предусмотрели свойство `Shifts`, представляющее коллекцию сущностей `Shift`.

Далее мы добавим в папку Models файл Factory.cs со следующим содержимым:

```
using System.ComponentModel.DataAnnotations.Schema;
namespace MvcDataApp.Models;
public record Factory
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int FactoryId { get; set; }
    public string FactoryName { get; set; }
    public string Location { get; set; }
    public ICollection<Shift> Shifts { get; set; }
}
```

Наконец, добавим следующий объект, представляющий собой сущность `Shift`:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace MvcDataApp.Models;
public record Shift
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int ShiftId { get; set; }
    [Range(1, 7)]
    public int WeekDay { get; set; }
    public TimeSpan StartTime { get; set; }
    public TimeSpan EndTime { get; set; }
    public int FactoryId { get; set; }
```

```
    public int EmployeeId { get; set; }
    public Employee Employee { get; set; }
    public Factory Factory { get; set; }
}
```

Теперь у нас есть объекты-сущности, представляющие все таблицы БД, которые мы описали ранее. Свойства этого класса соответствуют типам данных SQL. Мы также можем установить дополнительные ограничения для этих типов данных, как это сделано с помощью атрибута `Range` (Диапазон) для свойства `WeekDay` (День недели). В нашем примере в соответствующий целочисленный столбец можно вносить только значения от 1 до 7.

Теперь нам нужно сообщить Entity Framework, что эти объекты представляют собой сущности в таблицах БД. Для этого нам нужно будет добавить так называемый контекст базы данных и зарегистрировать его.

Добавление контекста базы данных

Контекст базы данных — это пользовательский класс, производный от базового класса `DbContext` в Entity Framework, позволяющий зарегистрировать все объекты-сущности, которые мы собираемся сопоставить с таблицами БД.

В нашем случае мы дадим нашему классу контекста БД имя `FactoryManagerContext`. Для его создания мы добавим в корневую папку проекта папку Data и создадим в ней файл FactoryManagerContext.cs со следующим содержимым:

```
using Microsoft.EntityFrameworkCore;
using MvcDataApp.Models;
namespace MvcDataApp.Data;
public class FactoryManagerContext : DbContext
{
    public FactoryManagerContext(DbContextOptions<FactoryManagerContext> options) :
                                                                        base(options)
    {
    }
    public DbSet<Employee> Employees { get; set; }
    public DbSet<Factory> Factories { get; set; }
    public DbSet<Job> Jobs { get; set; }
    public DbSet<Shift> Shifts { get; set; }
}
```

Мы добавили свойство `DbSet` для каждого из типов сущностей, созданных нами ранее. Теперь необходимо зарегистрировать этот объект контекста БД и связать его с определенной строкой подключения к экземпляру SQL Server. Мы сделаем это в файле Program.cs, в начале которого будет содержаться следующий код:

```
var builder = WebApplication.CreateBuilder(args);
// Добавим сервисы в контейнер.
builder.Services.AddControllersWithViews();
var app = builder.Build();
```

```
CreateDbIfNotExists(app);
// Настроим конвейер HTTP-запросов.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // По умолчанию значение HSTS составляет 30 дней.
    // Для реальных сценариев это значение можно изменить,
    // см. https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.Run();
static void CreateDbIfNotExists(IHost app)
{
    using var scope = app.Services.CreateScope();
    var services = scope.ServiceProvider;
    try
    {
        var context = services.GetRequiredService<FactoryManagerContext>();
        DbInitializer.Initialize(context);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "Failed to create the DB.");
    }
}
```

Начнем с того, что добавим в файл Program.cs операторы `using`:

```
using Microsoft.EntityFrameworkCore;
using MvcDataApp.Data;
```

Затем добавим следующие строки в любом месте перед `builder.Build()`:

```
builder.Services
        .AddDbContext<FactoryManagerContext>(options => options
        .UseSqlServer(builder
        .Configuration
        .GetConnectionString("DefaultConnection")));
    builder.Services.AddDatabaseDeveloperPageExceptionFilter();
```

Здесь мы зарегистрировали контекст БД со строкой подключения к БД, имеющей ключ `DefaultConnection`. Вызов метода `AddDatabaseDeveloperPageExceptionFilter` поможет

нам найти и устранить любые ошибки при настройке БД, выводя их в виде HTML. Однако наличие этой строки в рабочей версии кода нежелательно.

Чтобы соединение с БД работало, необходимо добавить в настройки соответствующий код. Для этого мы добавим в файл appsettings.json секцию `ConnectionStrings` с элементом `DefaultConnection`. В следующем примере показано содержимое файла appsettings.json со встроенным соединением `LocalDB`:

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "DefaultConnection": "Server=(localdb)\\
           mssqllocaldb;Database=FactoryManager;Trusted_
           Connection=True;MultipleActiveResultSets=true"
    }
}
```

Теперь использование Entity Framework в нашем приложении полностью настроено. Далее мы добавим некоторый дополнительный код, чтобы в полной мере реализовать преимущества подхода "сначала код".

Добавление скрипта создания базы данных

Если мы применим подход "сначала код" к Entity Framework, то при первом запуске приложения в месте, указанном строкой подключения, будет создана БД. Чтобы сделать это, добавим класс `DbInitializer` в папку Data, наполнив его следующим содержимым:

```
using MvcDataApp.Models;
namespace MvcDataApp.Data;
public class DbInitializer
{
    public static void Initialize(FactoryManagerContext context)
    {
        context.Database.EnsureCreated();
        if (context.Jobs.Any())
        {
            return;
        }
    }
}
```

Метод `Initialize` создаст БД, если она еще не существует. Это делается путем вызова метода `EnsureCreated` для свойства `Database` объекта контекста БД. Если в таблице `Jobs` БД уже есть некоторые данные, выполнение этого кода завершается. Это необ-

ходимо для того, чтобы можно было пропустить шаги добавления данных, если они уже были выполнены.

Чтобы реализовать шаги добавления данных, нам потребуется вставить в метод `Initialize` еще немного кода. Поскольку это будет достаточно длинный скрипт, мы будем добавлять его поэтапно. Сначала вставим следующий код, чтобы добавить несколько записей в таблицу `Jobs`:

```
var jobs = new Job[]
{
new Job
    {
        JobTitle = "Manager", Compensation = 50000
    },
    new Job
        {
            JobTitle = "Laborer", Compensation = 25000
        }
};
foreach (var j in jobs)
{
    context.Jobs.Add(j);
}
context.SaveChanges();
```

Здесь мы создаем коллекцию объектов `Job`. Затем добавляем каждый из них в набор данных `Jobs` в контексте БД. А после этого вызываем метод `SaveChanges`.

Вызов метода `SaveChanges` после добавления данных критически важен. Именно этот метод выполняет фактическое занесение данных в БД. До его вызова все данные будут находиться только в памяти.

Далее, добавим некоторые данные в таблицу `Employees`:

```
var employees = new Employee[]
{
new Employee
    {
        FirstName="John",
        LastName="Smith",
        DateOfBirth=DateTime.Parse("1992-10-01"),
        StartDate=DateTime.Parse("2020-09-01"),
        JobId = 1,
    },
new Employee
    {
        FirstName="Alexander",
        LastName="Marshall",
        DateOfBirth=DateTime.Parse("1982-09-12"),
        StartDate=DateTime.Parse("2017-09-01"),
```

```
        JobId = 2,
    },
new Employee
    {
        FirstName="Michael",
        LastName="Davidson",
        DateOfBirth=DateTime.Parse("1989-05-11"),
        StartDate=DateTime.Parse("2010-09-01"),
        JobId = 2,
    },
};
foreach (var e in employees)
{
    context.Employees.Add(e);
}
context.SaveChanges();
```

После этого добавим несколько сущностей `Factories`:

```
var factories = new Factory[]
{
new Factory
    {
        FactoryName = "Best Cookies",
        Location = "New York"
    }
};
foreach (var f in factories)
{
    context.Factories.Add(f);
}
context.SaveChanges();
```

Затем мы можем применить тот же принцип для добавления любого количества произвольных данных о сменах в таблицу `Shifts`, как показано в следующем примере:

```
var shifts = new Shift[]
{
new Shift
    {
        WeekDay = 1,
        StartTime = new TimeSpan(9,0,0),
        EndTime = new TimeSpan(17,0,0),
        FactoryId = 1,
        EmployeeId = 1
    },
new Shift
    {
        WeekDay = 2,
```

```
        StartTime = new TimeSpan(9,0,0),
        EndTime = new TimeSpan(17,0,0),
        FactoryId = 1,
        EmployeeId = 1
    },
};
foreach (var s in shifts)
{
    context.Shifts.Add(s);
}
context.SaveChanges();
```

Обратите внимание, что мы не добавляем вручную столбцы с идентификаторами. Это связано с тем, что БД сделает это за нас автоматически. Если мы добавляем данные во вновь созданную БД, то значение идентификатора первой записи в каждой таблице будет равно 1. Следующим будет 2, и для каждой записи значение идентификатора будет последовательно увеличиваться на единицу. Именно на этом принципе были основаны значения столбцов внешних ключей. Например, т. к. у нас есть только две сущности `Job`, в столбце `JobId` таблицы `Employees` могут находиться только значения 1 или 2.

Теперь осталось добавить код, который будет вызывать этот скрипт при запуске приложения. Для этого вставим в файл Program.cs следующий метод:

```
static void CreateDbIfNotExists(IHost app)
    {
    using var scope = app.Services.CreateScope();
    var services = scope.ServiceProvider;
    try
    {
        var context = services.GetRequiredService<FactoryManagerContext>();
        DbInitializer.Initialize(context);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "Failed to create the DB.");
    }
}
```

Затем вызовем этот метод, передав ему переменную `app`:

```
CreateDbIfNotExists(app);
```

Теперь мы готовы запустить приложение и посмотреть, как формируется база данных.

Создание базы данных путем запуска приложения

Мы можем запустить наше приложение, выполнив команду `dotnet run` из консоли, или сделать это через IDE. Поскольку мы еще не создали БД, следует ожидать появления в консоли различных операторов SQL. Затем, когда приложение будет запущено, мы можем открыть обозреватель объектов SQL Server и посмотреть на только что созданную БД. На рис. 5.3 показано, как должны выглядеть данные в таблице `Jobs`.

JobId	JobTitle	Compens...
1	Manager	50000.00
2	Laborer	25000.00

Рис. 5.3. Данные в таблице Jobs (Должности)

На рис. 5.4 показано содержимое таблицы `Factories`.

FactoryId	FactoryName	Location
1	Best Cookies	New York

Рис. 5.4. Данные в таблице Factories (Заводы)

Данные в таблице `Employees` показаны на рис. 5.5.

EmployeeId	FirstName	LastName	DateOfBirth	StartDate	JobId
1	John	Smith	01/10/199...	01/09/202...	1
2	Alexander	Marshall	12/09/198...	01/09/201...	2
3	Michael	Davidson	11/05/1989...	01/09/201...	2

Рис. 5.5. Данные в таблице Employees (Сотрудники)

И, наконец, в зависимости от того, какие записи мы добавили в таблицу `Shifts`, данные в ней будут выглядеть примерно так, как показано на рис. 5.6.

ShiftId	WeekDay	StartTime	EndTime	FactoryId	EmployeeId
1	1	09:00:00	17:00:00	1	1
2	2	09:00:00	17:00:00	1	1
3	3	09:00:00	17:00:00	1	1
4	4	09:00:00	17:00:00	1	1
5	5	09:00:00	17:00:00	1	1
6	1	09:00:00	17:00:00	1	2
7	2	09:00:00	17:00:00	1	2
8	3	09:00:00	17:00:00	1	2
9	4	09:00:00	17:00:00	1	2
10	5	09:00:00	17:00:00	1	2
11	1	09:00:00	17:00:00	1	3
12	2	09:00:00	17:00:00	1	3
13	3	09:00:00	17:00:00	1	3
14	4	09:00:00	17:00:00	1	3
15	5	09:00:00	17:00:00	1	3

Рис. 5.6. Данные в таблице Shifts (Смены)

На этом обзор подхода к созданию базы данных на основе кода ("сначала код") в EF7 завершен. Но можно поступить и наоборот. Объекты-сущности можно создать автоматически на основе анализа существующей БД (подход "сначала база данных"). Этот подход мы рассмотрим далее.

Подход "сначала база данных" в EF7

Инструменты командной строки EF7 позволяют создать весь код Entity Framework, используя существующее соединение с БД. Эти инструменты несовершенны, и очень часто возникает необходимость переработки кода автоматически созданных классов и удаления из них лишнего. Но полученные таким образом объекты-сущности, как правило, все равно будут представлять таблицу БД с достаточной степенью точности.

Создание моделей EF7 из существующей БД

Прежде чем приступить к реконструированию существующей БД, необходимо добавить в проект, в котором мы хотим это сделать, следующий пакет NuGet:

```
Microsoft.EntityFrameworkCore.Design
```

Затем следует добавить пакет NuGet, представляющий провайдер БД, который мы собираемся использовать. Например, для выбора провайдера SQL Server необходимо добавить следующий пакет NuGet:

```
Microsoft.EntityFrameworkCore.SqlServer
```

Затем, чтобы запустить процесс реконструирования, необходимо выполнить из папки проекта следующую команду:

```
dotnet ef dbcontext scaffold <строка подключения к базе данных> <имя NuGet-пакета провайдера базы данных>
```

Эта команда запустит процесс формирования кода, который заключается в чтении схемы БД и автоматической генерации кода на основе этой схемы. Объекты EF, которые мы создавали вручную на основе подхода "сначала код", такие как контекст БД и модели данных, создаются автоматически.

Предусмотрены некоторые дополнительные параметры, которые можно применить. Например, несколько раз можно указать параметр `--Table`, если требуется включить только определенные таблицы. Или можно задать параметр `--schema` для включения объектов только из определенной схемы. Например, если требуется реконструировать созданную нами ранее БД SQL Server и включить в нее только объекты из стандартной схемы `dbo`, это можно сделать с помощью следующей команды:

```
dotnet ef dbcontext scaffold "Server=(localdb)\
mssqllocaldb;Database=FactoryManager;Trusted_
Connection=True;MultipleActiveResultSets=true"
Microsoft.EntityFrameworkCore.SqlServer --schema dbo
```

Мы также можем повысить безопасность этой команды путем извлечения строки подключения из секретов .NET вместо открытого текстового значения. Для этого сначала нужно добавить секрет, выполнив следующую команду:

```
dotnet user-secrets set ConnectionStrings:Default
"Server=(localdb)\mssqllocaldb;Database=FactoryManager;Trusted_
Connection=True;MultipleActiveResultSets=true"
```

И тогда мы можем изменить нашу исходную команду, указав не фактическое значение, а ключ секрета:

```
dotnet ef dbcontext scaffold Name=ConnectionStrings:Default
                                    Microsoft.EntityFrameworkCore.SqlServer --schema dbo
```

Теперь мы можем создать еще один проект веб-приложения, добавить в него необходимые пакеты NuGet и выполнить одну из указанных ранее команд в папке проекта.

Просмотр автоматически сгенерированного кода

Выполнив эти команды в папке нового проекта, следует ожидать, что в ней появятся все объекты сущностей, а также контекст БД, как показано на рис. 5.7.



Рис. 5.7. Автоматически сгенерированный код в папке проекта

Если мы откроем один из файлов, содержащих объекты-сущности, то он будет выглядеть примерно так, как показано на рис. 5.8.

Как видно, может потребоваться некоторая переработка кода и удаление лишних элементов. Например, мы видим, что в предыдущем примере нам не нужен ни один из операторов `using`. Но такой подход все равно позволит сэкономить много времени, поскольку нам не придется писать классы сущностей вручную.

На этом мы завершим обзор основных возможностей Entity Framework и рассмотрим новые возможности, которые были добавлены в EF7.

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

namespace DatabaseFirstDataApp
{
    public partial class Employee
    {
        public Employee()
        {
            Shifts = new HashSet<Shift>();
        }

        public int EmployeeId { get; set; }
        public string FirstName { get; set; } = null!;
        public string LastName { get; set; } = null!;
        public DateTime DateOfBirth { get; set; }
        public DateTime StartDate { get; set; }
        public int JobId { get; set; }

        public virtual Job Job { get; set; } = null!;
        public virtual ICollection<Shift> Shifts { get; set; }
    }
}
```

Рис. 5.8. Автоматически сгенерированный класс сущности сотрудника

Новейшие функции EF7

Седьмая версия Entity Framework, как и все версии, выпущенные до нее, имеет свой набор полезных особенностей. Они выходят за рамки простого исправления ошибок и повышения производительности.

Подход "сначала база данных" и управление с помощью шаблонов T4

Шаблоны T4 в течение многих лет применялись в различных технологиях Microsoft. Они предназначены в основном для генерации текста по определенным правилам. Форматирование шаблонов T4 позволило разработчикам реализовать сложную смесь текста и управляющей логики для генерации документов любой сложности. Теперь шаблоны T4 можно использовать для определения сложных конфигураций при генерации кода на основе существующей БД с помощью EF7.

Не все БД представляют собой простую совокупность таблиц, объединенных единой схемой. Существует множество корпоративных БД, в которых есть большое количество таблиц, несколько схем и сложные бизнес-правила. Иногда представлять их все в коде ORM не требуется. Именно в этом случае шаблоны T4 могут быть особенно удобны.

Защищенный ключ

По умолчанию объекты-сущности Entity Framework, предназначенные для представления свойства первичного ключа `identity`, имеют тип данных `int`. У этого свой-

ства должен быть как геттер, так и сеттер, что может вызвать некоторые проблемы. Несмотря на то что значение в столбце `identity` должно быть неизменяемым в БД, оно является всего лишь доступным для записи целочисленным значением в объекте, представляющем запись БД. Поэтому ничто не мешает вам случайно изменить значения ключей и получить неожиданные результаты, например обновить не ту запись в БД.

Для защиты от подобных ситуаций в EF7 добавлена концепция защищенного ключа. Защищенный ключ — это ключ, обернутый в свой собственный класс. Это значение может быть инициализировано через конструктор, поэтому Entity Framework все равно может извлечь его из БД в объект-сущность. Однако после создания это значение изменить нельзя.

Чтобы применить защищенный ключ, мы можем внести следующее изменение в наш автоматически сгенерированный класс сущности `Employee`:

```
namespace DatabaseFirstDataApp
{
    public partial class Employee
    {
        public Employee()
        {
            Shifts = new HashSet<Shift>();
        }
        public EmployeeKey EmployeeId { get; set; };
        public string FirstName { get; set; } = null!;
        public string LastName { get; set; } = null!;
        public DateTime DateOfBirth { get; set; }
        public DateTime StartDate { get; set; }
        public int JobId { get; set; }
        public virtual Job Job { get; set; } = null!;
        public virtual ICollection<Shift> Shifts { get; set; }
    }
    public class EmployeeKey
{
    public EmployeeKey(Func<int> generator) => EmployeeId = generator();
        public EmployeeKey(int id) => EmployeeId = id;
        public int EmployeeId { get; private set; }
    }
}
```

Как видим, вместо целочисленного свойства `EmployeeId` мы имеем класс `EmployeeKey`, который выступает в качестве обертки доступного только для чтения свойства типа `int`. У этого класса есть два конструктора. Один из них принимает целочисленное значение, а другой — функцию, возвращающую значение `int`. Таким образом, он совместим с любым кодом, который может вызывать Entity Framework.

Отображение "одна таблица на конкретный тип" (TPC)

В предшественнике EF7, версии EF Core 6, предлагались два способа работы с объектами-сущностями (отображения), которые наследовали друг от друга:

- **"одна таблица на иерархию"** (**Table-Per-Hierarchy, TPH**), где каждая полная иерархия наследования была представлена одной таблицей;
- **"одна таблица на тип" (Table-Per-Type, TPT)**, при котором любые типы, в том числе и абстрактные, помещались в отдельные таблицы, которые ссылались друг на друга посредством отношений с использованием внешнего ключа.

Теперь в Entity Framework добавлен третий тип отображения — **"одна таблица на конкретный тип" (Table-Per-Concrete-type, TPC)**, создающий таблицу на основе каждого конкретного типа в иерархии.

Чтобы объяснить, как работают различные типы отображений, давайте изменим наше исходное приложение, реализующее подход с созданием БД на основе кода. Добавим в папку Models файл Person.cs и внесем в него следующий код:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace MvcDataApp.Models;
public abstract record Person
{
    public int Id { get; set; }
    [StringLength(20)]
    public string FirstName { get; set; }
    [StringLength(20)]
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    [NotMapped]
    public string FullName => FirstName + " " + LastName;
   }
```

Затем изменим нашу запись `Employee` так, чтобы она наследовала от этой базовой записи:

```
namespace MvcDataApp.Models;
public record Employee : Person
{
    public DateTime StartDate { get; set; }
    public int JobId { get; set; }
    public Job Job { get; set; }
    public ICollection<Shift> Shifts { get; set; }
}
```

После этого добавим файл `Supervisor.cs`, который будет содержать запись, наследующую от записи `Employee`:

```
namespace MvcDataApp.Models;
public record Supervisor : Employee
```

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

Вот как будут работать отображения различных типов. При использовании TPH в БД будет одна таблица, поскольку у нас имеется только одна иерархия наследования (`Person – Employee – Supervisor`). В случае TPT у нас было бы три таблицы: одна для `Person`, одна для `Employee` и еще одна для `Supervisor`. В каждой таблице будут только те поля, которые соответствуют свойствам класса исходной сущности, за исключением столбца `Id`, который присутствует во всех таблицах. Таблицы будут использовать этот столбец для связи друг с другом. Если мы задействуем добавленное в новой версии отображение TPC, то у нас будут две таблицы: `Supervisors` и `Employees`. Сущность `Person` не будет иметь связанной с ней таблицы, поскольку она представляет собой абстрактную запись. Но в каждой таблице будут присутствовать все поля, имеющиеся во всей иерархии наследования.

Чтобы продемонстрировать работу TPC, удалим существующую БД и изменим определение `FactoryManagerContext`:

```
using Microsoft.EntityFrameworkCore;
using MvcDataApp.Models;
namespace MvcDataApp.Data;
public class FactoryManagerContext : DbContext
{
    public FactoryManagerContext(DbContextOptions<FactoryManagerContext> options) :
                                                                        base(options)
    {
    }
    public DbSet<Employee> Employees { get; set; }
    public DbSet<Supervisor> Supervisors { get; set; }
    public DbSet<Factory> Factories { get; set; }
    public DbSet<Job> Jobs { get; set; }
    public DbSet<Shift> Shifts { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>().UseTpcMappingStrategy();
        modelBuilder.Entity<Employee>();
        modelBuilder.Entity<Supervisor>();
        modelBuilder.Entity<Factory>();
        modelBuilder.Entity<Job>();
        modelBuilder.Entity<Shift>();
    }
}
```

Для применения TPC нам необходимо зарегистрировать наши объекты-сущности, включая абстрактные классы, внутри переопределения метода `OnModelCreating`. Нам также необходимо вызвать метод `UseTpcMappingStrategy` в строке, регистрирующей базовый тип иерархии, которой в нашем случае является `Person`. Отображение

TPT включено по умолчанию, но его также можно активизировать явно, вызвав метод `UseTptMappingStrategy`. Отображение TPH включается путем вызова метода `UseTphMappingStrategy`.

Теперь, удалив исходную БД, мы можем применить TPC при создании новой БД. Для этого необходимо заменить скрипт добавления записей `Employee` внутри класса `DbInitializer` следующим кодом:

```
var employees = new Employee[]
{
    new Employee
    {
    Id = 1,
        FirstName="Alexander",
        LastName="Marshall",
        DateOfBirth=DateTime.Parse("1982-09-12"),
        StartDate=DateTime.Parse("2017-09-01"),
        JobId = 2,
        },
    new Employee
    {
    Id=3,
        FirstName="Michael",
        LastName="Davidson",
        DateOfBirth=DateTime.Parse("1989-05-11"),
        StartDate=DateTime.Parse("2010-09-01"),
        JobId = 2,
        },
};
foreach (var e in employees)
{
context.Employees.Add(e);
}
context.SaveChanges();
```

Затем добавим следующий скрипт для добавления записи `Supervisor`:

```
var supervisors = new Supervisor[]
{
new Supervisor
{
    Id=1,
        FirstName="John",
        LastName="Smith",
        DateOfBirth=DateTime.Parse("1992-10-01"),
        StartDate=DateTime.Parse("2020-09-01"),
        JobId = 1,
    },
};
```

```
foreach (var s in supervisors)
{
context.Employees.Add(s);
}
context.SaveChanges();
```

Теперь наша таблица `Employees` должна выглядеть так, как показано на рис. 5.9.

На рис. 5.10 показано, как должна выглядеть таблица `Supervisors`.

Теперь перейдем к последнему заметному новшеству EF7 — перехватчикам.

Id	FirstName	LastName	DateOfBirth	StartDate	JobId
2	Alexander	Marshall	12/09/198...	01/09/201...	2
3	Michael	Davidson	11/05/1989...	01/09/201...	2

Рис. 5.9. Данные в таблице Employees (Сотрудники)

Id	FirstName	LastName	DateOfBirth	StartDate	JobId	TeamSize
1	John	Smith	01/10/199...	01/09/202...	1	0

Рис. 5.10. Данные в таблице Supervisors (Руководители)

Перехватчики

Перехватчики в EF7 позволяют ввести некоторую дополнительную логику при инициировании запроса. Такой перехватчик должен реализовать интерфейс `IMaterializationInterceptor` из пространства имен `Microsoft.EntityFrameworkCore.Diagnostics`.

Рассмотрим пример. Создадим в папке Data нашего исходного проекта файл EmployeeCachingInterceptor.cs и наполним его следующим содержимым:

```
using Microsoft.EntityFrameworkCore.Diagnostics;
using MvcDataApp.Models;
using System.Collections.Concurrent;
namespace MvcDataApp.Data;
public class EmployeeCachingInterceptor : IMaterializationInterceptor
{
    private static readonly ConcurrentDictionary<string, Employee> EmployeeCache = new();
    public InterceptionResult<object> CreatingInstance(
        MaterializationInterceptionData materializationData,
        InterceptionResult<object> result)
    {
        if (materializationData.EntityType.ClrType == typeof(Employee))
        {
            var employeeName = materializationData.
                                    GetPropertyValue<string>(nameof(Employee.FullName));
```

```
            if (EmployeeCache.TryGetValue(employeeName, out var country))
            {
                Console.WriteLine($"Got employee '{country.FullName}' from the cache.");
                return InterceptionResult<object>.SuppressWithResult(country);
            }
        }
        return result;
    }
}
```

Этот перехватчик использует для кеширования данных `Employee` конкурентный словарь. Если некоторая запись `Employee` присутствует в словаре, мы извлекаем ее оттуда. В противном случае мы извлекаем ее из БД.

Теперь нам необходимо зарегистрировать наш перехватчик. Для этого мы добавим в класс `FactoryManagementContext` следующее переопределение метода `OnConfiguring`:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) =>
        optionsBuilder.AddInterceptors(new EmployeeCachingInterceptor());
```

На этом обзор EF7 и всех его новых возможностей завершен.

Заключение

В этой главе мы ознакомились с обзором реляционных БД. Мы также рассмотрели SQL — язык, используемый реляционными БД для взаимодействия с данными. Мы обсудили различные варианты применения EF7. Теперь вы знакомы со всеми способами создания реляционной БД и подключения к ней EF7, включая подходы "сначала код" и "сначала база данных".

Подход "сначала код" позволяет создать совершенно новую БД из кода. Этот подход позволяет разработчику сосредоточиться на коде, доверив фреймворку автоматическое создание хранилища данных.

Подход "сначала база данных" больше подходит для сценариев, в которых ORM необходимо подключить к существующей БД. В EF7 имеются средства, позволяющие считать схему БД и сгенерировать соответствующие абстракции в коде.

Мы также рассмотрели новейшие возможности EF7, такие как защищенные ключи, которые скрывают детали реализации логики генерации ключей и предотвращают случайную передачу в методы неправильных ключей.

В следующей главе вы узнаете, как использовать ASP.NET Core — основной фреймворк для построения веб-приложений на .NET.

Что следует помнить

- Реляционные БД определяются таблицами, которые связаны с другими таблицами посредством так называемых отношений внешних ключей.
- Назначение языка структурированных запросов (SQL) — взаимодействие с реляционными БД.

- EF7 является основным (и текущим) инструментом объектно-реляционного отображения (ORM) в .NET и имеет библиотеки, обеспечивающие его совместимость со всеми основными типами реляционных БД.
- EF7 способен как генерировать БД из кода, так и генерировать код на основе существующих БД.
- Суть подхода "сначала код" — сначала пишется код ORM, а затем на его основе создается реляционная БД.
- Суть подхода "сначала база данных" — автоматическая генерация кода ORM на основе существующей БД.

Контрольные вопросы

1. В чем разница между запросом и командой?

а) Разницы нет, и эти понятия взаимозаменяемы;

б) Запрос предназначен для получения данных, а команда — для манипулирования данными;

в) Запрос предназначен для манипулирования данными, а команда — для их получения;

г) Запросы регистрируются, а команды — нет.

2. В чем разница между подходами "сначала код" и "сначала модель" в EF7?

а) Это одно и то же;

б) В EF7 нет подхода "сначала код";

в) В EF7 нет подхода "сначала модель";

г) Подход "сначала код" позволяет сгенерировать новую БД, а подход "сначала модель" опирается на существующую БД.

3. Какой подход необходимо применить для построения модели EF7 на основе существующей БД?

а) "Сначала код";

б) "Сначала модель";

в) "Сначала схема";

г) "Сначала база данных".

4. Что такое нормализация БД?

а) Процесс устранения избыточных записей;

б) Процесс, приводящий к тому, что в каждой таблице остается минимальное количество столбцов;

в) Все вышеперечисленное;

г) Ничего из перечисленного.

Ответы

1. б.

2. в.

3. г.

4. в.

Основные термины

- **РСУБД** — реляционная система управления БД.
- **SQL Server** — РСУБД компании Microsoft.
- **Oracle Database** — коммерческая РСУБД компании Oracle.
- **MySQL** — полукоммерческая упрощенная РСУБД компании Oracle.
- **PostgreSQL** — РСУБД с расширенными возможностями и открытым исходным кодом.
- **SQL** — язык структурированных запросов, предназначенный для взаимодействия с РСУБД.
- **ORM** — инструмент объектно-реляционного отображения, сопоставляющий объекты в коде (например, классы) с объектами в БД (например, таблицами).
- **Entity Framework 7** — основной инструмент объектно-реляционного отображения платформы .NET, разрабатываемый компанией Microsoft наряду с самим .NET.
- **"Сначала код"** — подход к использованию ORM, при котором сначала пишется код, а после этого на его основе формируется БД.
- **"Сначала база данных"** — подход к использованию ORM для генерации кода на основе существующей БД.

Глава 6

Типы веб-приложений в .NET

Введение

В .NET имеется собственный фреймворк разработки веб-приложений, который называется ASP. NET Core. Первоначальная версия платформы Active Server Pages (ASP), которую сейчас называют ASP Classic, состояла из полустатических файлов, содержащих смесь HTML-разметки и серверного кода. Позднее была создана усовершенствованная версия, получившая название ASP.NET. В ней отпала необходимость совмещать в одних и тех же файлах клиентскую разметку и серверный код. Разработчики могли выделить компоненты фронтенда и бэкенда в отдельные файлы, что позволило обеспечить четкое разделение решаемых проблем и сделать код более легким для чтения и удобным для сопровождения. Новый фреймворк также содержит собственные реализации широко распространенных архитектурных патернов, таких как MVC, поэтому разработчикам больше не нужно применять их вручную.

Один из основных недостатков ASP.NET заключался в том, что этот фреймворк был доступен только под Windows, как и все остальные продукты платформы .NET того времени. Но все изменилось в 2016 году, когда на свет появилась кроссплатформенная версия .NET Core. Была создана новая версия ASP.NET, которую можно было сочетать с любой распространенной операционной системой, а не только Windows. Этот новый вариант фреймворка получил название ASP.NET Core.

ASP.NET Core — это постоянно развивающийся фреймворк для разработки веб-приложений. В этой главе мы рассмотрим все шаблоны приложений, доступные в ASP.NET Core. При этом мы также уделим внимание последним возможностям, появившимся в его седьмой версии.

Структура

В этой главе мы рассмотрим новейшие возможности ASP.NET Core 7, которые собраны в следующие темы:

- основы ASP.NET Core;
- Web API в ASP.NET Core;
- MVC в ASP.NET Core;
- Razor Pages в ASP.NET Core.

Цели

К концу этой главы вы научитесь работать со всеми основными типами веб-приложений ASP.NET Core и ознакомитесь с последними возможностями, которые были добавлены в 7-ю версию фреймворка.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из этих компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Основы ASP.NET Core

Шаблоны проектов ASP.NET Core доступны в .NET SDK по умолчанию. Однако если вы используете Visual Studio 2022 и хотите создавать проекты ASP.NET Core в графическом интерфейсе, то вам придется включить рабочую нагрузку веб-разработки через установщик Visual Studio. Для этого следует запустить приложение **Visual Studio Installer**, нажать кнопку **Modify** (Изменить) рядом с экземпляром Visual Studio 2022 и выбрать для установки рабочую нагрузку **ASP.NET and Web development**, как показано на рис. 6.1.

После завершения установки всех необходимых компонентов мы можем создать базовое приложение ASP.NET Core, для чего познакомимся с его структурой.

Структура базового приложения ASP.NET Core

Создадим проект на основе шаблона `ASP.NET Core Empty` и назовем его `BasicAspNetCoreApp`. Если вы используете .NET CLI, это можно сделать, выполнив следующую команду:

```
dotnet new web -o BasicAspNetCoreApp
```

После создания ваш проект будет иметь структуру, показанную на рис. 6.2.

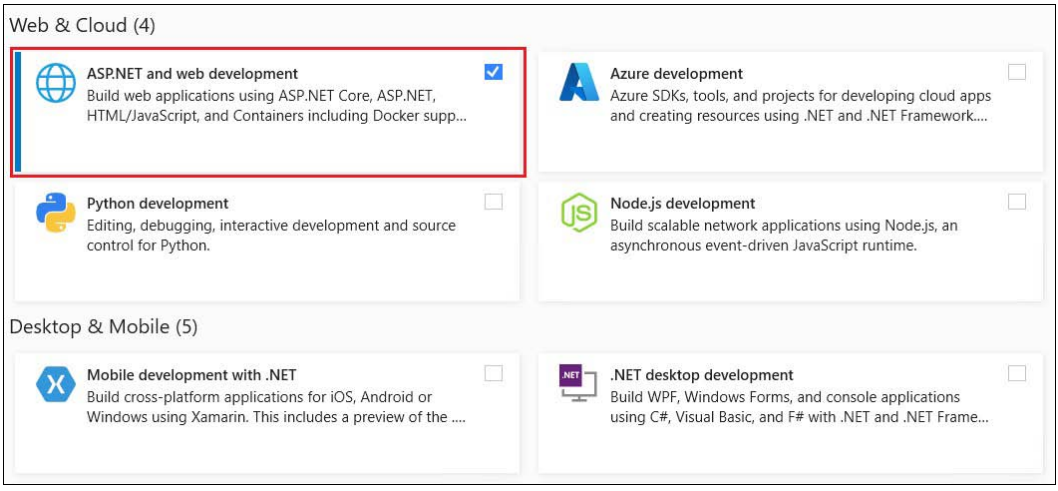


Рис. 6.1. Включение рабочей нагрузки ASP.NET Core в Visual Studio Installer

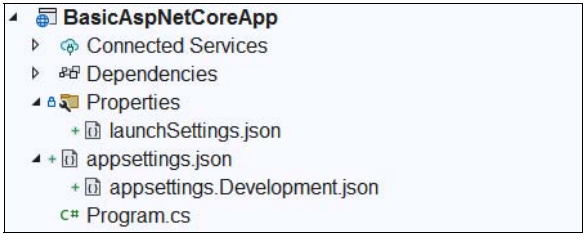


Рис. 6.2. Базовая структура проекта ASP.NET Core

Файл с именем appsettings.json — главный файл, в котором хранятся настройки приложения. Также могут существовать разновидности этого файла для конкретной среды, например appsettings.Development.json, заменяющие главный файл, если приложение выполняется на машине, на которой переменной `ASPNETCORE_ENVIRONMENT` присвоено значение, совпадающее с суффиксом в имени файла.

Также имеется папка Properties с файлом launchSettings.json. Этот файл используется в основном в среде разработки и задает правила хостинга для приложения. Например, настройки, показанные на рис. 6.3, позволяют разместить приложение на веб-сервере IIS Express и встроенном веб-сервисе Kestrel.

При хостинге на веб-сервере IIS Express приложение доступно по адресу `http://localhost:12139`, если сертификат не используется, и по адресу `https://localhost:44317`, если используется сертификат SSL. При хостинге на веб-сервисе Kestrel адреса будут соответственно следующими: `http://localhost:5051` и `https://localhost:7051`.

Чтобы запустить приложение с SSL-сертификатом на вашем рабочем компьютере, необходимо отметить сертификат разработки как доверенный. Это можно сделать через IDE или из командной строки .NET, выполнив следующую команду в Mac OS или Windows:

```
dotnet dev-certs https --trust
```



```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:12139",
      "sslPort": 44317
    }
  },
  "profiles": {
    "BasicAspNetCoreApp": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7051;http://localhost:5051",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Рис. 6.3. Пример содержимого файла launchSettings.json

При работе с Linux эта команда будет отличаться в зависимости от установленного дистрибутива.

Наконец, у нас есть файл Program.cs, который является стандартной точкой входа в приложение .NET. Его содержимое будет выглядеть следующим образом:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Welcome to ASP.NET Core!");
app.Run();
```

В этом файле есть переменная `builder`, которая позволяет настроить различные зависимости сервисов для нашего приложения. После выполнения всех необходимых настроек мы вызываем метод `Build` переменной `builder` и создаем на ее основе переменную `app`. Эта переменная позволяет нам добавлять различные действия в промежуточные средства обработки запросов. Начиная с версии .NET 6, мы также можем настраивать конечные точки HTTP для приложения непосредственно в переменной `app` в файле Program.cs. Например, мы вызываем метод `MapGet`, чтобы при обращении по корневому адресу нашего приложения возвращался определенный текст.

Чтобы посмотреть, как это работает, мы можем изменить этот текст, запустить приложение, открыть браузер и перейти по URL-адресу приложения, указанному в файле launchSettings.json.

Чтобы запустить приложение в Kestrel, достаточно выполнить команду `dotnet run` из папки application. Затем мы можем перейти по URL-адресу сервиса в браузере и убедиться, что отображается приветствие `Welcome to ASP.NET Core!`.

Теперь мы перейдем к более сложным шаблонам веб-приложений, начиная с Web API.

Web API в ASP.NET Core

Шаблон Web API в ASP.NET Core предназначен в первую очередь для создания приложений без пользовательского интерфейса. Такие приложения также известны как веб-сервисы, и их назначение — предоставлять данные другим авторизованным приложениям по их запросу.

В качестве внешнего интерфейса Web API в ASP.NET Core использует REpresentative State Transfer (REST), интегрированный с такими инструментами, как OpenAPI (также известный как Swagger), которые помогают визуализировать конечную точку REST приложения в браузере.

В седьмой версии ASP.NET Core предусмотрены два вида приложений Web API: Web API с контроллерами и Minimal API. Далее мы рассмотрим оба вида приложений по-отдельности.

Web API с контроллерами

Для определения конечных точек, доступных по протоколу HTTP, Web API с контроллерами опирается на так называемые классы контроллеров. Чтобы продемонстрировать, как это работает, создадим новое приложение на основе шаблона Web API в ASP.NET Core. Если мы создаем приложение с помощью IDE, необходимо убедиться, что включена опция **Use Controllers** (Использовать контроллеры). Если используется .NET CLI, приложение можно создать, выполнив следующую команду:

```
dotnet new webapi -o WebApiAppWithControllers
```

Теперь у нас есть приложение, для которого в файл Program.cs добавлены некоторые дополнительные параметры. Например, в нем есть вызов метода `builder.Services.AddControllers()` для добавления компонентов контроллеров, `builder.Services.AddSwaggerGen()` для включения поддержки Swagger и `app.MapControllers()` для добавления классов контроллеров в конвейер обработки запросов и подготовки их к использованию.

Этот шаблон показывает, как построить правильный конвейер промежуточных средств обработки запросов. Большинство методов переменной `app` гарантируют, что определенные действия добавляются в конвейер в том порядке, в котором вызываются эти методы. Например, вызов `UseHttpsRedirection` перед `UseAuthorization` и вызов `MapControllers` после него гарантирует, что если в исходном запросе использовался протокол HTTP, то сначала выполняется перенаправление на HTTPS. Затем мы проверяем, что клиент авторизован для дальнейшей работы. Только после этого клиент направляется к конкретному ресурсу по пути, указанному URL-адресом.

Теперь рассмотрим файл WeatherForecastController.cs, расположенный в папке Controllers. Мы знаем, что этот класс является контроллером API, поскольку в нем есть атрибут `ApiController` и он наследует от `ControllerBase`. Поскольку для него в качестве маршрута по умолчанию указано `[controller]`, базовый путь для всех конечных точек на этом контроллере будет соответствовать имени класса контроллера за вычетом суффикса `Controller`. Этот путь будет выглядеть следующим образом:

```
{Базовый URL-адрес}/WeatherForecast
```

Чтобы продемонстрировать некоторые полезные возможности ASP.NET Core, мы переработаем структуру этого кода. Мы создадим сервис, который затем добавим в контроллер с помощью внедрения зависимостей. Это позволит нам сохранить так называемый *принцип тонкого контроллера*, который предписывает, что контроллер должен лишь обрабатывать HTTP-запросы и возвращать ответы клиентам. Вся бизнес-логика должна быть реализована внутри отдельных сервисов.

Начнем с добавления в проект файла WeatherForecastService.cs со следующим интерфейсом:

```
namespace WebApiAppWithControllers;
public interface IWeatherForecastService
{
    IEnumerable<WeatherForecast> GetFiveDayForecast();
}
```

Затем мы можем добавить в тот же файл класс, реализующий этот интерфейс:

```
public class WeatherForecastService : IWeatherForecastService
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing",
        "Bracing",
        "Chilly",
        "Cool",
        "Mild",
        "Warm",
        "Balmy",
        "Hot",
        "Sweltering",
        "Scorching"
    };
    public IEnumerable<WeatherForecast> GetFiveDayForecast()
    {
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
```

```
        .ToArray();
    }
}
```

По сути, мы создали отдельный класс, выполняющий ту же работу, которая обычно возложена на контроллер. Теперь мы можем зарегистрировать этот класс в нашем контейнере внедрения зависимостей. Для этого мы добавим в файл Program.cs следующую строку в любом месте до вызова метода `Build` переменной `builder`:

```
builder.Services.AddTransient<IWeatherForecastService,
WeatherForecastService>();
```

Она сообщает базовому фреймворку, что если в конструктор какого-либо файла, на который нет прямых ссылок в коде (например, контроллер), передается параметр типа `IWeatherForecastService`, он будет автоматически преобразован в новый экземпляр класса `WeatherForecastService`.

Поскольку мы задействовали метод `AddTransient`, мы будем получать новый экземпляр этого класса при каждом запросе. Но если бы мы захотели использовать во всем приложении один и тот же экземпляр класса `WeatherForecastService`, то могли бы вместо этого применить метод `AddSingleton`.

Пока мы работаем с файлом Program.cs, мы также можем настроить ведение журнала с выводом на консоль. Для этого добавим следующие две строки перед строкой вызова метода `Build`:

```
builder.Logging.ClearProviders();
builder.Logging.AddConsole();
```

Теперь изменим наш контроллер, чтобы он выглядел следующим образом:

```
using Microsoft.AspNetCore.Mvc;
namespace WebApiAppWithControllers.Controllers;
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    private readonly IWeatherForecastService _weatherForecastService;
    private readonly ILogger<WeatherForecastController> _logger;
    public WeatherForecastController(
        ILogger<WeatherForecastController> logger,
        IWeatherForecastService weatherForecastService)
    {
        _logger = logger;
        _weatherForecastService = weatherForecastService;
    }
}
```

Затем заменим предыдущий метод следующим:

```
[HttpGet(Name = "GetWeatherForecast")]
public IActionResult Get()
```

```
{
    _logger.LogInformation("Obtaining 5-day weather forecast.");
    try
    {
        var forecast = _weatherForecastService.GetFiveDayForecast();
        return Ok(forecast);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error obtaining weather forecast;");
        throw;
    }
}
```

Атрибут `HttpGet` указывает, что этот метод вызывается, когда клиент отправляет запрос `GET HTTP`. Поскольку путь в этом методе не указан, он будет запускаться при использовании базового пути контроллера.

Этот метод полагается на реализацию интерфейса `IWeatherForecastService`, которая была внедрена в конструктор. Но можно применить внедрение зависимостей и к отдельным методам. В ASP.NET Core версии 7 это делается неявно. Чтобы проверить это, можно добавить следующий метод:

```
[HttpGet("injected-service")]
public IActionResult GetFromService(IWeatherForecastService weatherForecastService)
{
    _logger.LogInformation("Obtaining 5-day weather forecast.");
    try
    {
        var forecast = weatherForecastService.GetFiveDayForecast();
        return Ok(forecast);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error obtaining weather forecast;");
        throw;
    }
}
```

В качестве пути в этом методе указывается `injected-service`. Таким образом, полный путь для этой конечной точки будет выглядеть следующим образом:

```
{Базовый URL-адрес}/WeatherForecast/injected-service
```

Теперь мы рассмотрим еще одну возможность, которая была добавлена в 7-ю версию фреймворка — функцию `TryParse` в параметрах метода `controller`, которая позволяет изящно обрабатывать случаи, когда клиенты отправляют параметры с неверным типом данных. Чтобы продемонстрировать работу этой функции, добавим в папку Controllers контроллер TryParseDemoController.cs и наполним его следующим содержимым:

```
using Microsoft.AspNetCore.Mvc;
namespace WebApiAppWithControllers.Controllers;
[ApiController]
[Route("[controller]")]
public class TryParseDemoController : ControllerBase
{
    [HttpGet(Name = "TryParseInt")]
    public IActionResult Get([FromQuery] IntParser parser)
    {
        if (parser?.Value == null)
            return NoContent();
        return Ok(parser.Value);
    }
}
public class IntParser
{
    public int? Value { get; set; }
    public static bool TryParse(int? input, out IntParser? result)
    {
        if (input is null)
        {
            result = default;
            return false;
        }
        result = new IntParser { Value = input };
        return true;
    }
}
```

Чтобы эта функция работала, нам необходим пользовательский класс со статическим булевым методом `TryParse`, который возвращает параметр `output`. Эту роль исполняет класс `IntParser`. Теперь, если мы обратимся к этой конечной точке по следующему URL-адресу:

```
{Базовый URL-адрес}/TryParseDemo
```

то получим код ответа `204` из-за отсутствия параметра строки запроса и синтаксический анализатор установит для свойства `Value` пустое значение.

Однако если мы укажем параметр строки запроса, то получим код ответа `200` с возвращенным целым значением:

```
{base URL}/TryParseDemo?Value=2
```

Теперь мы можем запустить наше приложение и посмотреть, как выглядят добавленные нами конечные точки контроллера в Swagger. У нас уже есть все необходимые настройки для промежуточного ПО Swagger в файле Program.cs, поэтому все, что нам нужно сделать, это перейти по базовому URL-адресу приложения. При этом ожидается появление страницы, аналогичной той, что показана на рис. 6.4.

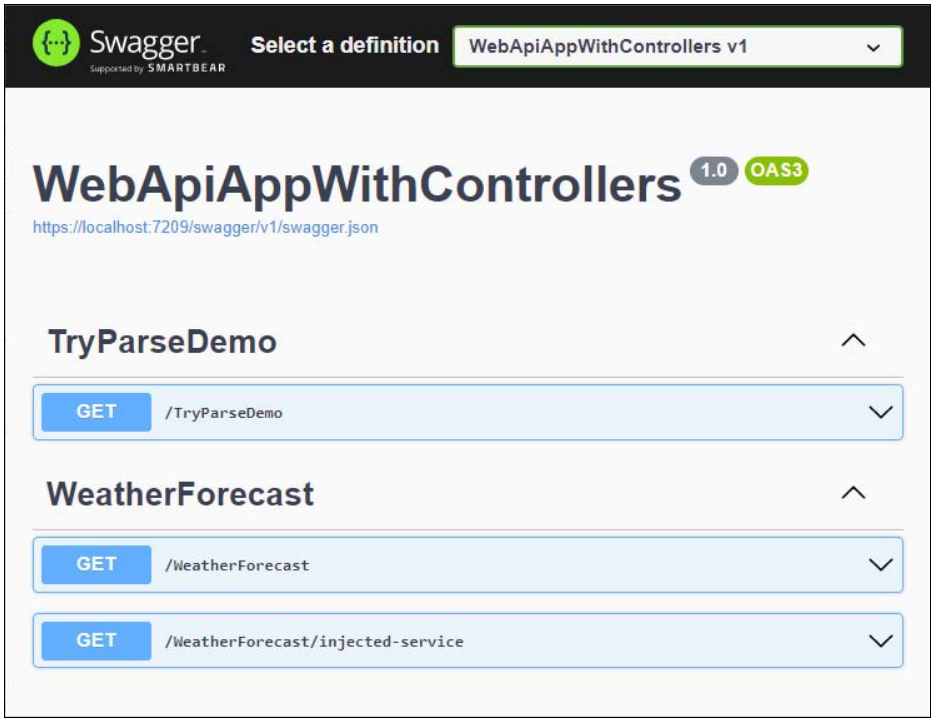


Рис. 6.4. Пользовательский интерфейс Swagger

Теперь мы рассмотрим еще один способ разработки приложений Web API в .NET 7 — Minimal API.

Конечные точки Minimal API

Minimal API — это функция, позволяющая добавлять конечные точки REST API в веб-приложение без контроллеров. Эта возможность была добавлена только в ASP.NET Core версии 6 и очень похожа на то, как настраиваются конечные точки REST в приложении Node.js.

Мы кратко рассмотрели использование Minimal API, когда создавали наше первоначальное базовое приложение ASP. NET Core. Оно вызывало метод `MapGet` для переменной `app` в файле Program.cs. Но существует и более продвинутый шаблон Minimal API. Достаточно создать приложение на основе Web API, а затем либо снять флажок **Use Controllers** (Использовать контроллеры), если вы делаете это из IDE, или добавить параметр `-minimal`, если это делается с помощью командной строки .NET. Таким образом, полная команда может выглядеть следующим образом:

```
dotnet new webapi -o WebApiAppWithMinimalApis -minimal
```

Все конечные точки HTTP созданного таким образом приложения будут отображены внутрь класса `Program.cs`. В ASP.NET Core версии 7 специально для Minimal API был добавлен целый ряд новых возможностей. Рассмотрим их подробнее.

Добавление метаданных Open API

В файле Program.cs только что добавленного проекта можно найти вызов `MapGet` переменной `app` и вставить следующее выражение перед точкой с запятой в конце полного оператора:

```
.WithDescription("The endpoint for retrieving weather forecasts.")
```

В результате это описание будет добавлено в документ Open API, который может быть отображен на странице Swagger. В качестве альтернативы можно добавить такое же описание, вставив атрибут `EndpointSummary` в главный вызов, например, следующим образом:

```
app.MapGet("/weatherforecast", [EndpointSummary("The endpoint for retrieving weather
forecasts.")]() => ...
```

Либо можно найти вызов `WithOpenApi` и изменить его следующим образом:

```
.WithOpenApi(operation => {
    operation.Summary = "The endpoint for retrieving weather forecasts.";
    return operation;
});
```

Далее мы рассмотрим недавние улучшения передачи параметров в конечные точки при использовании Minimal API.

Улучшение передачи параметров в Minimal API

Если у вас есть несколько параметров с одинаковым типом данных, то теперь можно передавать все параметры в методы конечных точек как единый массив. Но это относится только к массивам примитивных типов, массивам строк и объектам `StringValues`. Вот пример:

```
app.MapGet("/repeated-strings", (string[] names) =>
$"value 1: {names[0]} , value 2: {names[1]}, value 3: {names[2]}");
```

Аналогичным образом вместо передачи отдельных параметров в методы конечных точек можно просто создать объект, который будет представлять каждый параметр как свойство, и затем передать этот объект. Последовательность действий такова. Сначала добавим этот объект:

```
internal struct ParamsRequest
{
    public int Id { get; set; }
    public int Page { get; set; }
}
```

Затем создадим метод конечной точки, принимающий этот объект в качестве набора параметров:

```
app.MapGet("/parameters-object",
  ([AsParameters] ParamsRequest request) =>
  $"Id{request.Id}, Page: {request.Page}");
```

Этот метод автоматически распознает этот объект как набор параметров, поскольку он помечен атрибутом `AsParameters`.

Далее мы узнаем, как сделать так, чтобы Minimal API возвращал типизированные результаты.

Minimal API и типизированные результаты

Типизированные результаты позволяют возвращать объекты, имеющие строго типизированную полезную нагрузку в виде данных и различные связанные с ними поля метаданных. Для этого метод конечной точки должен возвращать реализацию интерфейса `IResult`. Чтобы продемонстрировать это, создадим файл TypedResultsDemo.cs со следующим содержимым:

```
namespace WebApiAppWithMinimalApis;
public class Data
{
    public int Id { get; set; } = 1;
    public string Name { get; set; } = "test";
}
public static class TypedResultsDemo
{
    public static void MapTypedDataApi(this IEndpointRouteBuilder routes)
    {
        routes.MapGet("/typed-data", ReturnTypedResult);
    }
    public static Task<IResult> ReturnTypedResult()
    {
        return Task.FromResult(Results.Ok(Task.FromResult(new Data())));
    }
    }
}
```

Теперь мы можем вызвать метод расширения `MapTypedDataApi` из класса `Program.cs` для регистрации данной конечной точки. Для этого нам необходимо убедиться, что в файле Program.cs имеется ссылка на пространство имен из этого вновь добавленного файла, например:

```
using WebApiAppWithMinimalApis;
```

И тогда достаточно добавить следующую строку в любом месте перед `app.Run()`:

```
app.MapTypedDataApi();
```

Теперь если мы запустим наше приложение, откроем его Swagger-страницу и вызовем конечную точку, которую мы только что добавили, то увидим, что она вернула JSON-версию нашего объекта `Data` вместе с несколькими полями метаданных, как показано на рис. 6.5.

Мы также можем настроить наши конечные точки так, чтобы они возвращали несколько типов результатов. Чтобы продемонстрировать это, мы можем добавить в класс `TypedResultsDemo` следующий метод:

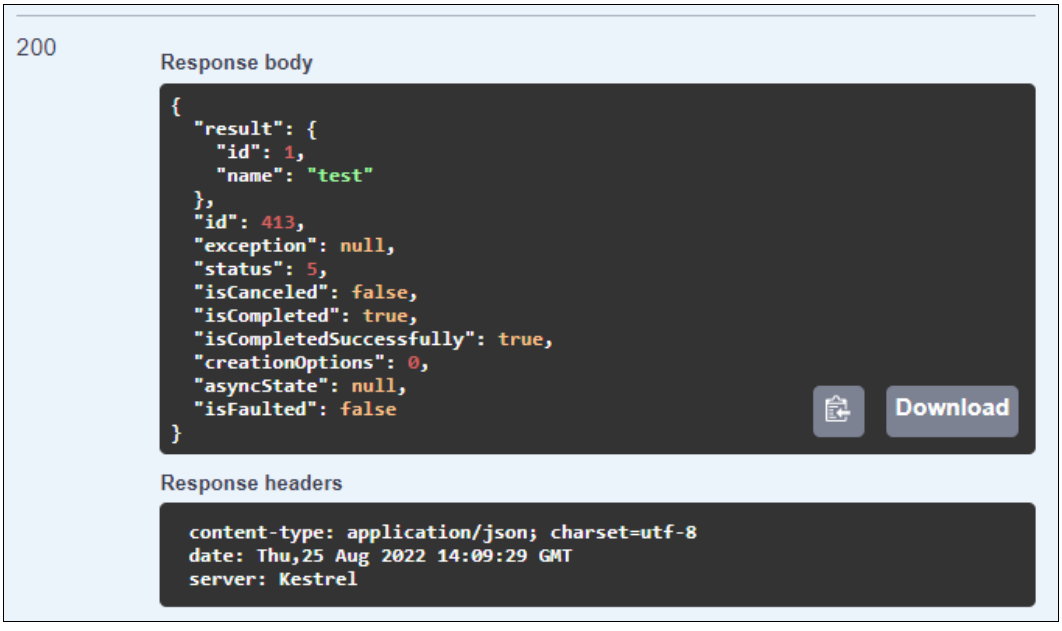


Рис. 6.5. Сериализованная JSON-версия реализации IResult

```
public static Results<Ok<Data>, NotFound> ReturnSingeItem(int id)
{
    return id == 1
        ? TypedResults.Ok(new Data())
        : TypedResults.NotFound();
}
```

Затем добавим в метод `MapTypedDataApi` следующую строку:

```
routes.MapGet("/typed-data/{id}", (int id) => ReturnSingeItem(id));
```

Объект-ответ, возвращаемый этой конечной точкой, будет выглядеть примерно так:

```
{
    "result": {
        "id": 1,
        "name": "test"
    },
    "id": 686,
    "exception": null,
    "status": 5,
    "isCanceled": false,
    "isCompleted": true,
    "isCompletedSuccessfully": true,
    "creationOptions": 0,
    "asyncState": null,
    "isFaulted": false
}
```

Теперь перейдем к другой новой возможности Minimal API: безопасной выгрузке файлов.

Выгрузка файлов в Minimal API

Для выгрузки файла на сервер требуется запрос `POST HTTP` вместо `GET`, поэтому, чтобы добавить соответствующую конечную точку, нам необходимо вызвать метод `MapPost` вместо метода `MapGet`. Эта конечная точка будет выглядеть следующим образом:

```
app.MapPost("/upload", async (IFormFile file) =>
{
    using var stream = File.OpenWrite("test.txt");
    await file.CopyToAsync(stream);
}).RequireAuthorization();
```

Здесь мы получаем данные, представляющие собой выгруженный пользователем файл, и сохраняем их в файле test.txt в той же папке, где размещено наше приложение. В реальном сценарии мы, скорее всего, передали бы данные в хранилище BLOB-объектов.

В конце мы также делаем вызов `RequireAuthorization`, который добавляется к нашим сопоставлениям конечных точек, чтобы гарантировать, что доступ к ним могут получить только авторизованные пользователи. Мы также можем передать в этот метод некоторые дополнительные параметры, чтобы задать более тонкие требования к доступу (роли, политики, утверждения и т. д.). Но чтобы все это работало, нам необходимо настроить промежуточное ПО аутентификации и авторизации.

Если говорить о промежуточном ПО, то недавно в него были добавлены некоторые новые возможности, которые мы рассмотрим далее.

Новое в промежуточном ПО для обработки запросов

В файле Program.cs можно настроить промежуточные средства обработки запросов с помощью переменной `builder` и создать для них конвейер с помощью переменной `app`. Теперь мы рассмотрим последние возможности, которые были добавлены в промежуточное ПО ASP.NET Core.

Начнем с настройки промежуточного ПО с помощью переменной `builder`. Одна из новых функций — возможность регистрации информации о пользовательских заголовках в журналах сервера, известных как журналы W3C. Это могут быть любые данные с произвольными ключами. Например, для регистрации информации о запросе, содержащем пользовательские ключи заголовков `custom-header` и `another-custom-header`, можно добавить следующий код:

```
builder.Services.AddW3CLogging(logging =>
{
    logging.AdditionalRequestHeaders.Add("custom-header");
    logging.AdditionalRequestHeaders.Add("another-custom-header");
});
```

Также можно настроить произвольную информацию о проблеме и записать ее в свой код. Для этого необходимо добавить перед `builder.Build()` следующий код:

```
builder.Services.AddProblemDetails();
```

В результате будет зарегистрирована реализация интерфейса `IProblemDetailsService`. Затем вы можете разрешить его из любого места внутри вашего приложения и вызвать его метод `WriteAsync` для записи деталей проблемы. Вот как можно разрешить этот сервис из `IHttpContext`:

```
var problemDetailsService = context.RequestServices.GetService<IProblemDetailsService>();
```

Теперь добавим следующую строку, которая зарегистрирует соответствующий провайдер, позволяющий распаковывать сжатые данные запроса:

```
builder.Services.AddRequestDecompression();
```

Далее мы рассмотрим собственно конвейер промежуточного ПО. Первое, что мы можем добавить в него, — это возможность автоматической распаковки сжатых данных запроса. Для этого вставим после `builder.Build()` следующую строку:

```
app.UseRequestDecompression();
```

Затем мы можем получить или установить значение cookie-файла, который отслеживает, дал ли пользователь согласие на использование политики cookie-файлов. Это можно сделать с помощью свойства `ConsentCookieValue` класса `CookiePolicyOptions`. Вот как мы можем установить это значение в конвейере обработки запросов:

```
app.UseCookiePolicy(new CookiePolicyOptions
{
    ConsentCookieValue = "yes"
});
```

К сопоставлениям конечных точек добавляются некоторые действия по обработке запросов. Например, можно добавить метод `CacheOutput` для кеширования результатов запроса, чтобы вызов не приводил к множеству обращений к хранилищу данных или дорогостоящим вычислениям. Например, в следующей конечной точке дата будет вычисляться только один раз. И тогда при повторном обращении того же клиента к полученному результату он просто вернет уже вычисленное значение:

```
app.MapGet("/cached-date", () => DateTime.UtcNow.ToString()). CacheOutput();
```

Кроме того, имеется возможность настраиваемой фильтрации, которую мы можем применить с помощью метода `AddEndpointFilter`. Например, в следующем примере продемонстрирована конечная точка, которая ограничивает доступ всем, кроме пользователя с определенным именем:

```
string GetGreetingMessage(string name) => $"User {name} is allowed to access reource";
app.MapGet("/filter/invocation-context/{name}", GetGreetingMessage)
    .AddEndpointFilter(async (routeHandlerInvocationContext, next) =>
    {
        var name = (string)routeHandlerInvocationContext.Arguments[0];
```

```
        if (name == "Chris Davidson")
        {
            return Results.Problem("Access is not allowed for Chris Davidson!");
        }
        return await next(routeHandlerInvocationContext);
});
```

Метод `AddEndpointFilter` также может работать с атрибутом `RouteHandlerContext` или реализацией интерфейса `IRouteHandlerFilter`.

На этом завершим обзор Web API и его новейших возможностей и рассмотрим типы веб-приложений, имеющих пользовательские интерфейсы. Начнем с шаблона MVC.

MVC в ASP.NET Core

Модель-представление-контроллер (Model-View-Controller, MVC) — это архитектурный паттерн, разработанный для эффективного манипулирования данными через пользовательский интерфейс. Эта особенность делает паттерн MVC особенно хорошо подходящим для создания корпоративных приложений.

Как следует из названия, MVC-приложение состоит из компонентов трех типов:

1. **Модель** — представляет данные в бэкенде.
2. **Представление** — реализует определенный экран в пользовательском интерфейсе.
3. **Контроллер** — обеспечивает взаимодействие между представлениями и моделями.

Контроллеры MVC и Web API концептуально похожи. Но вместо того, чтобы просто возвращать некоторые данные, они обслуживают представления для браузеров.

Мы уже рассматривали примеры моделей MVC в *главе 5*, и объект-сущность (например, `Employee`) — это модель в контексте MVC. Чтобы продемонстрировать основы работы MVC-приложений в ASP.NET Core, мы продолжим развивать приложение `MvcDataApp`, которое создали в этой главе.

Чтобы упростить демонстрацию, мы удалим все добавленные нами перехватчики. Самый простой способ сделать это — удалить переопределение `OnConfiguring` из класса `FactoryManagerContext`.

Затем мы внесем некоторые изменения в файл _Layout.cshtml, расположенный в папке Shared внутри папки Views. Этот файл содержит общую компоновку для всех представлений внутри приложения. Он позволяет реализовать такие компоненты, как общий верхний колонтитул, общая панель навигации, общий нижний колонтитул и т. д. Он написан на основе шаблона Razor, который представляет собой комбинацию необработанной HTML-разметки, языка C#, а также различных ключевых слов и операторов, которые склеивают разметку и код. Этот шаблон выполняется на сервере и формирует HTML-страницу, которая затем доставляется клиенту.

Первое, что мы сделаем, — это найдем HTML-элемент `title` и заменим его следующим:

```
<title>@ViewData["Title"] - Factory Manager</title>
```

Затем мы внесем некоторые изменения в нашу панель навигации. Для этого мы найдем элемент `Ul` с атрибутом класса `navbar-nav` и заменим его следующим:

```
<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>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-controller="Employees"
                                                           asp-action="Index">Employees</a>
    </li>
</ul>
```

Теперь у нас есть три элемента на панели навигации: `Home`, `Privacy` и `Employees`. Элементы `asp-controller` и `asp-action` указывают, какой контроллер и какой метод действия (т. е. представление) представляет каждый элемент. Для контроллеров MVC принято такое же соглашение об именовании, как и для контроллеров Web API, которые мы рассматривали ранее. В данном случае первые два элемента относятся к классу `HomeController`. Первый элемент относится к методу действия под названием `Index`, а второй — к методу действия `Privacy`. Мы можем посмотреть, как выглядят эти методы действий, открыв класс `HomeController.cs` из папки Controllers. Эти представления расположены в папке Home внутри папки Views. Третий пункт относится к методу действия `Index` класса `EmployeesController`, которого у нас еще нет. Так что добавим его.

Класс `EmployeesController` и соответствующие представления дадут нам возможность просматривать данные `Employees` из созданной нами ранее БД и управлять ими. Если вы работаете в Visual Studio 2022, то самый простой способ добавления соответствующего контроллера и представления — использование шаблона для генерации кода. Для этого необходимо щелкнуть правой кнопкой мыши на проекте в окне **Solution Explorer** (Обозреватель решений), выбрать пункт **New** (Добавить), затем выбрать пункт **Scaffolded Item** (Создать шаблонный элемент). В открывшемся окне следует выбрать опцию **MVC Controller with views using Entity Framework** (Контроллер MVC с представлениями, использующий Entity Framework), как показано на рис. 6.6.

Затем необходимо выбрать соответствующий класс контекста базы данных (**FactoryManagerContext** в нашем случае) и модель сущности (**Employee**), отметить все опции для представлений и убедиться, что для контроллера задано имя **EmployeesController**, как показано на рис. 6.7.

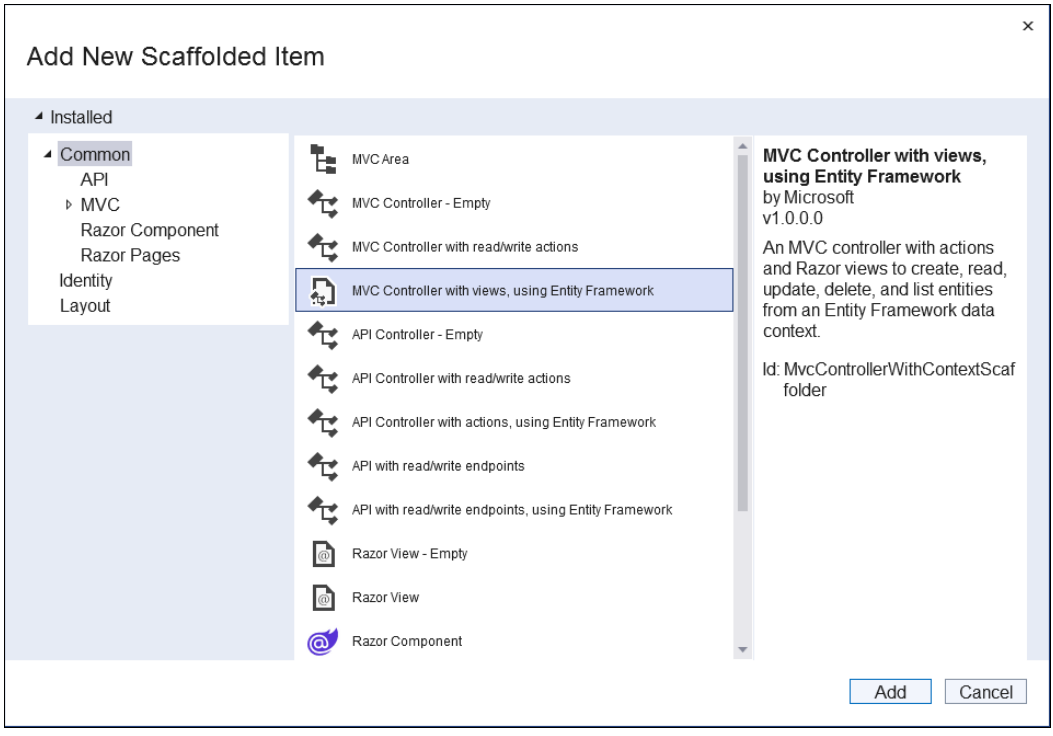


Рис. 6.6. Шаблонные представления и контроллеры на основе моделей EF

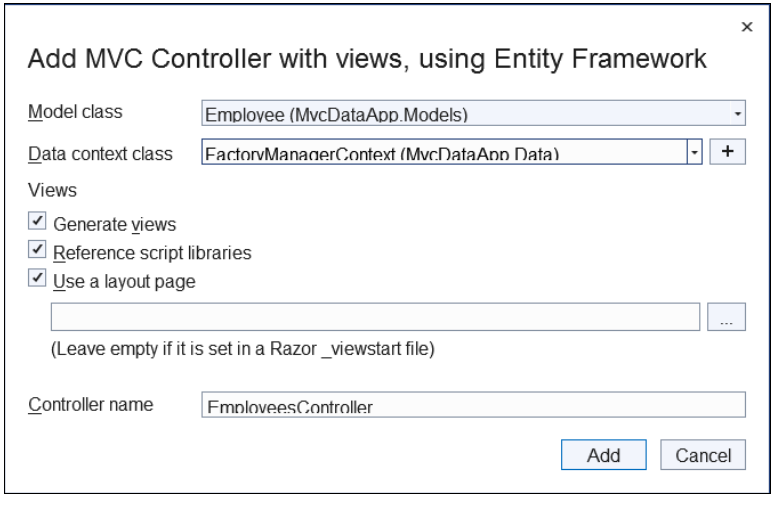


Рис. 6.7. Настройка параметров для генерации кода

Процесс генерации кода может занять некоторое время, и после его завершения в папке Controllers проекта должен появиться файл EmployeesController.cs. Также в папке Views должна появиться папка Employees с рядом CSHTML-файлов, как показано на рис. 6.8.

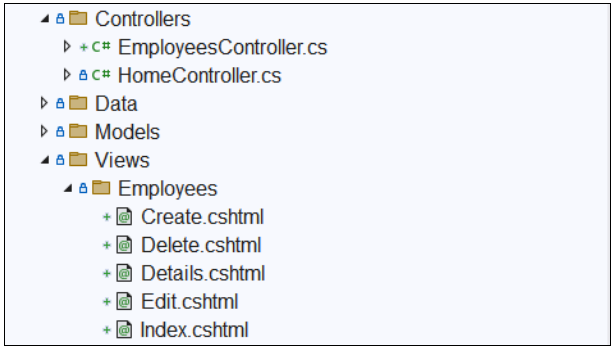


Рис. 6.8. Созданные контроллер и представления

Если вы не используете Visual Studio 2022, то можете добавить контроллер и представления вручную. Начнем с добавления файла EmployeesController.cs в папку Controllers и наполнения его следующим содержимым:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using MvcDataApp.Data;
using MvcDataApp.Models;
namespace MvcDataApp.Controllers
{
    public class EmployeesController : Controller
    {
        private readonly FactoryManagerContext _context;
        public EmployeesController(FactoryManagerContext context)
        {
            _context = context;
        }
        private bool EmployeeExists(int id)
        {
        return (_context.Employees?.Any(e => e.Id == id)).GetValueOrDefault();
    }
}
```

Затем мы добавим метод действия `Index`, который будет служить нам представлением, где перечислены все сущности `Employees`, имеющиеся в нашей БД:

```
public async Task<IActionResult> Index()
{
    var factoryManagerContext = _context.Employees.Include(e => e.Job);
    return View(await factoryManagerContext.ToListAsync());
}
```

После этого добавим метод действия `Details`, который отображает подробную информацию о любой конкретной сущности `Employee` на основе ее `Id`:

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null || _context.Employees == null)
    {
        return NotFound();
    }
    var employee = await _context.Employees
        .Include(e => e.Job)
        .FirstOrDefaultAsync(m => m.Id == id);
    if (employee == null)
    {
        return NotFound();
    }
    return View(employee);
}
```

Мы также добавим два метода действия, которые дадут нам возможность создать новую сущность `Employee`. Доступ к одному из них будет осуществляться через запрос `GET`, и этот метод даст нам форму для создания сущности. Для доступа к другому будет вызываться запрос `POST`, который позволяет отправить форму:

```
public IActionResult Create()
{
    ViewData["JobId"] = new SelectList(_context.Jobs, "JobId", "JobId");
    return View();
}
[HttpPost]
    [ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
            [Bind("StartDate,JobId,Id,FirstName,LastName,DateOfBirth")] Employee employee)
{
    if (ModelState.IsValid)
    {
        _context.Add(employee);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    ViewData["JobId"] = new SelectList(_context.Jobs, "JobId", "JobId", employee.JobId);
    return View(employee);
}
```

Теперь добавим методы действия `GET` и `POST`, которые дадут нам возможность редактировать имеющиеся сущности `Employee`. Обратите внимание, что доступ к конечным точкам действий по умолчанию осуществляется с помощью HTTP-операции `GET`. Если нужно применить другую операцию, то к этому методу следует добавить атрибут, например `HttpPost` или `HttpDelete`:

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null || _context.Employees == null)
```

```
    {
        return NotFound();
    }
    var employee = await _context.Employees.FindAsync(id);
    if (employee == null)
    {
        return NotFound();
    }
ViewData["JobId"] = new SelectList(_context.Jobs, "JobId", "JobId", employee.JobId);
    return View(employee);
}
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
        [Bind("StartDate,JobId,Id,FirstName,LastName,DateOfBirth")]
        Employee employee)
{
    if (id != employee.Id)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(employee);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!EmployeeExists(employee.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
ViewData["JobId"] = new SelectList(_context.Jobs, "JobId", "JobId", employee.JobId);
    return View(employee);
}
```

Наконец, мы добавим методы действия `GET` и `POST` для удаления сущности `Employee`:

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null || _context.Employees == null)
```

```
    {
        return NotFound();
    }
    var employee = await _context.Employees
        .Include(e => e.Job)
        .FirstOrDefaultAsync(m => m.Id == id);
    if (employee == null)
    {
        return NotFound();
    }
return View(employee);
}
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    if (_context.Employees == null)
    {
        return Problem("Entity set 'FactoryManagerContext.Employees' is null.");
    }
    var employee = await _context.Employees.FindAsync(id);
    if (employee != null)
    {
        _context.Employees.Remove(employee);
    }
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Теперь нам необходимо создать Razor-представление для каждого метода действия GET, которые мы добавили в контроллер. Для этого мы создадим в папке Views папку Employees. Сначала добавим файл Index.cshtml, первая часть которого будет выглядеть следующим образом:

```
@model IEnumerable<MvcDataApp.Models.Employee>
@{
    ViewData["Title"] = "Index";
}
<h1>Index</h1>
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
```

```
            <th>
                @Html.DisplayNameFor(model => model.Job)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.LastName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.DateOfBirth)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
```

Затем для формирования строк таблицы мы можем выполнить следующий цикл foreach:

```
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.StartDate)
            </td>
           <td>
                @Html.DisplayFor(modelItem => item.Job.JobId)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstName)
            </td>
            <td>
                    @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.DateOfBirth)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Теперь добавим файл `Create.cshtml` со следующим содержимым:

```
@model MvcDataApp.Models.Employee
@{
    ViewData["Title"] = "Create";
}
<h1>Create</h1>
<h4>Employee</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="JobId" class="control-label"></label>
                <select asp-for="JobId" class ="form-control"
                asp- items="ViewBag.JobId"></select>
            </div>
            <div class="form-group">
                <label asp-for="FirstName" class="control-label"></label>
                <input asp-for="FirstName" class="form-control" />
                <span asp-validation-for="FirstName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="LastName" class="control-label"></label>
                <input asp-for="LastName" class="form-control" />
                <span asp-validation-for="LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="DateOfBirth" class="control-label"></label>
                <input asp-for="DateOfBirth" class="form-control" />
                <span asp-validation-for="DateOfBirth" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-action="Index">Back to List</a>
</div>
```

```
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Обратите внимание, что имя каждого из этих файлов совпадает с именем одного из файлов методов действия `GET` контроллера с добавлением расширения CSHTML. Таким образом файл метода действия `Details` будет выглядеть следующим образом:

```
@model MvcDataApp.Models.Employee
@{
    ViewData["Title"] = "Details";
}
<h1>Details</h1>
<div>
    <h4>Employee</h4>
    <hr />
    <dl class="row">
        <dt class =  "col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Job)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Job.JobId)
        </dd>
        <dt class =  "col-sm-2">
            @Html.DisplayNameFor(model => model.FirstName)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.FirstName)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.LastName)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.LastName)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.DateOfBirth)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.DateOfBirth)
        </dd>
    </dl>
</div>
```

```
<div>
    <a asp-action="Edit" asp-route-id="@Model?.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Представление `Edit` будет выглядеть следующим образом:

```
@model MvcDataApp.Models.Employee
@{
    ViewData["Title"] = "Edit";
}
<h1>Edit</h1>
<h4>Employee</h4>
<hr />
```

Теперь добавим следующую форму:

```
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="JobId" class="control-label"></label>
                <select asp-for="JobId" class="form-control" asp-items="ViewBag.JobId"></select>
                <span asp-validation-for="JobId" class="text-danger"></span>
            </div>
            <input type="hidden" asp-for="Id" />
            <div class="form-group">
                <label asp-for="FirstName" class="control-label"></label>
                <input asp-for="FirstName" class="form-control" />
                <span asp-validation-for="FirstName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="LastName" class="control-label"></label>
                <input asp-for="LastName" class="form-control" />
                <span asp-validation-for="LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="DateOfBirth" class="control-label"></label>
                <input asp-for="DateOfBirth" class="form-control" />
                <span asp-validation-for="DateOfBirth" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
```

```
        </form>
    </div>
</div>
```

И завершим представление следующим нижним колонтитулом:

```
<div>
    <a asp-action="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Представление `Delete` будет выглядеть следующим образом:

```
@model MvcDataApp.Models.Employee
@{
    ViewData["Title"] = "Delete";
}
<h1>Delete</h1>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Employee</h4>
    <hr />
    <dl class="row">
        <dt class =  "col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Job)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Job.JobId)
        </dd>
        <dt class =  "col-sm-2">
            @Html.DisplayNameFor(model => model.FirstName)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.FirstName)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.LastName)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.LastName)
        </dd>
```

```
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.DateOfBirth)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.DateOfBirth)
        </dd>
    </dl>
    <form asp-action="Delete">
       <input type="hidden" asp-for="Id" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-action="Index">Back to List</a>
    </form>
</div>
```

В результате базовое MVC-приложение завершено и способно работать с данными `Employee`. Для проверки мы можем запустить приложение и перейти к вкладке `Employees`. Должна появиться страница, показанная на рис. 6.9. При переходе по ссылкам должны отображаться соответствующие страницы — Edit, Details или Delete.



Рис. 6.9. Представление `Index` контроллера `Employees`

На этом завершим обзор шаблона приложения MVC в ASP.NET Core и перейдем к последнему шаблону приложения ASP.NET Core с пользовательским интерфейсом — Razor Pages.

Razor Pages в ASP.NET Core

В отличие от MVC, приложения Razor Pages предназначены не только для работы с данными, но и обладают большей гибкостью по сравнению с MVC, поскольку каждая страница является полуавтономной единицей, имеющей собственную логику бэкенда. Поэтому, хотя Razor Pages и обеспечивают большую гибкость, с ними

не так легко работать, как с MVC, когда необходимо разработать приложение специально для выполнения операций создания, чтения, обновления, удаления (Create-Read-Update-Delete, CRUD) через пользовательский интерфейс.

Для создания приложения Razor Pages мы можем либо выбрать шаблон ASP.NET Core Razor Pages при создании нового проекта с помощью графического интерфейса, либо выполнить следующую команду через CLI:

```
dotnet new web -o RazorPagesApp
```

Структура нашего проекта будет похожа на ту, что показана на рис. 6.10.

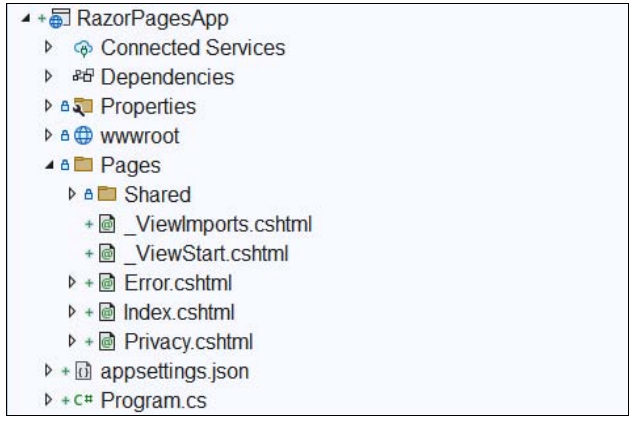


Рис. 6.10. Структура проекта Razor Pages

В проекте этого типа вместо папок Models, Views и Controllers имеется папка Pages. Как и в шаблоне MVC, здесь есть несколько общих компонентов компоновки, которые находятся в папке Shared внутри папки Pages. Как и в случае с представлениями MVC, файлы, представляющие страницы, имеют расширение cshtml и синтаксис Razor. Но на этом сходство и заканчивается. В Razor Pages страница может иметь так называемый файл кода программной части, у которого такое же имя, что и у файла, представляющего страницу, но дополнительное расширение cs. Таким образом, файл кода программной части для файла Index.cshtml имеет имя Index.cshtml.cs.

Файл кода программной части является чистым C#-файлом, поэтому в нем можно делать все то, что можно делать в любом другом классе C#, например передавать в него зависимости через конструктор и использовать любые виды бизнес-логики. В контексте Razor Pages на такой класс, наследующий от `PageModel`, можно без ограничений ссылаться из кода Razor на странице.

Когда клиент запрашивает страницу в приложении Razor Pages, в файлах кода программной части в заданном порядке запускаются определенные события. Существуют также некоторые методы, которые выполняются в соответствии с определенными соглашениями. Формат имен этих методов следующий:

```
On<обозначение HTTP-операции в стиле Pascal>
```

Если на страницу поступает запрос `GET`, то срабатывает метод `OnGet`, если поступает запрос `POST`, срабатывает метод `OnPost` и т. д. К этим методам можно применить некоторую дополнительную логику. Например, если мы заменим содержимое файла Index.cshtml.cs следующим кодом, то в журнал в виде отладочного сообщения будет записан идентификатор сеанса:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
namespace RazorPagesApp.Pages;
public class IndexModel : PageModel
{
    private readonly ILogger<IndexModel> _logger;
    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }
    public void OnGet()
    {
        _logger.LogDebug($"Homepage requested.Session id: {HttpContext.Session.Id}");
    }
}
```

Обратите внимание, что реализация параметра конструктора `Ilogger` настраивается в конвейере внедрения зависимостей. Этот объект дает нам возможность регистрировать информацию. При необходимости можно зарегистрировать и собственную реализацию.

На этом обзор шаблонов ASP.NET Core завершен. Подведем итоги тому, что мы узнали в этой главе.

Заключение

В этой главе мы изучили основы ASP.NET Core. Мы рассмотрели все доступные шаблоны проектов ASP.NET Core: пустой шаблон, MVC, Web API и Razor Pages. Мы изучили общую структуру приложений ASP.NET Core, в том числе скрипт запуска, включающий регистрацию зависимостей и промежуточное ПО для обработки запросов. Мы также рассмотрели модель хостинга ASP.NET Core и процесс реализации хостинга в среде разработки.

Мы уделили внимание шаблонам проектов MVC, которые позволяют разработчикам создавать компоненты отображения отдельно от бизнес-логики бэкенда и доступа к базе данных. Мы также рассмотрели шаблоны Web API, которые предназначены для построения веб-сервисов с конечными точками REST API. Наконец, мы рассмотрели шаблон Razor Pages, который позволяет разработчикам создавать индивидуальную логику бэкенда для каждой страницы.

Мы также рассказали о новых возможностях, добавленных в седьмую версию ASP.NET Core, таких как усовершенствование механизма кеширования.

В следующей главе мы рассмотрим технологию Blazor, которая позволяет разработчикам создавать скомпилированные .NET-приложения, запускаемые в браузерах.

Что следует помнить

- ASP.NET Core — это основной фреймворк для разработки веб-приложений, доступный на платформе .NET.
- В коде запуска любой реализации ASP.NET Core предусмотрены регистрация зависимостей и промежуточные средства конвейерной обработки запросов.
- По умолчанию ASP.NET Core поставляется с четырьмя основными шаблонами проектов: пустой шаблон, MVC, Web API и Razor Pages.
- Пустой шаблон проекта предоставляет лишь базовый набор инструментов для разработки веб-приложений, позволяя разработчикам добавлять только те компоненты, которые им действительно необходимы.
- Шаблон MVC использует архитектурный паттерн "модель-представление-контроллер".
- Шаблон Web API предназначен в первую очередь для построения интерфейсов REST API.
- Шаблон Razor Pages состоит из шаблонов веб-страниц, каждый из которых может также иметь связанный с ним так называемый файл кода программной части.

Контрольные вопросы

1. В чем разница между контроллерами в Web API и контроллерами в MVC?

а) Разницы нет, эти понятия полностью взаимозаменяемы;

б) Контроллеры MVC предназначены для обслуживания представлений, а контроллеры Web API возвращают только данные;

в) В Web API нет контроллеров;

г) Контроллеры Web API могут возвращать данные, тогда как контроллеры MVC — не могут.

2. Что такое Razor в контексте Razor Pages?

а) Язык, который можно использовать в бэкенде приложения вместо C#;

б) Имя фреймворка внедрения зависимостей, используемого Razor Pages;

в) Язык, предназначенный для написания клиентских шаблонов;

г) Это просто произвольное имя шаблона приложения.

3. Для чего используется свойство `Services` объекта `builder` в скрипте запуска ASP.NET Core?

а) Для регистрации необходимых сервисов, т. е. объектов, от которых зависят другие объекты;

б) Для предоставления промежуточных компонентов обработки запросов;

в) Для подключения приложения к любым внешним веб-сервисам;

г) Для запуска фоновых задач.

4. **Для чего используются промежуточные средства обработки запросов в ASP.NET Core?**

а) Для выполнения аутентификации;

б) Для выполнения авторизации;

в) Предоставление цепочки этапов обработки, через которые должен пройти запрос, прежде чем он достигнет цели;

г) Все вышеперечисленное.

Ответы

1. б.
2. в.
3. а.
4. г.

Основные термины

- **ASP.NET Core** — это основной набор инструментов для разработки веб-приложений на платформе .NET.
- **MVC** или **Model-View-Controller (Модель-представление-контроллер)** — архитектурный паттерн, в котором представления отвечают за логику отображения, модели — за бизнес-логику и работу с данными в бэкенде, а контроллеры — за объединение этих типов компонентов.
- **REST** или **REpresentational State Transfer (Передача состояния представления)** — тип интерфейса веб-сервисов, позволяющий клиентским приложениям обмениваться данными с веб-сервисами, как правило, с помощью JSON.
- **Web API** — шаблон приложения в ASP.NET Core, предназначенный для построения веб-сервисов с помощью REST API.
- **Minimal API** — вариант подхода к разработке приложений Web API, в котором не используются классы `Controller`.
- **Razor Pages** — шаблон приложения в ASP.NET Core, позволяющий разработчикам создавать отдельные шаблоны веб-страниц с индивидуальной логикой бэкенда.
- **Внедрение зависимостей** — процесс регистрации необходимых сервисов (например, классов) при запуске приложения.
- **Промежуточные средства обработки запросов** — настраиваемая цепочка этапов обработки, применяемых к входящему запросу в определенном порядке, прежде чем он попадет к адресату.

Глава 7

Blazor и WebAssembly в .NET

Введение

Еще совсем недавно единственным способом запустить скомпилированное высокопроизводительное приложение в браузере была установка специального плагина. Но даже в этом случае плагин мог быть совместим только с приложениями определенного типа. Например, для запуска Java-приложений необходим плагин Java, для запуска Flash-приложений — плагин Adobe Flash, а для запуска приложений .NET Framework — Microsoft Silverlight. Возможной альтернативой было выполнение кода JavaScript, который является интерпретируемым, а не компилируемым, что не позволяло достичь уровня производительности, доступного для компилируемого кода.

Но все изменилось с появлением технологии WebAssembly, которая стала стандартной функцией всех браузеров. WebAssembly — это формат двоичных инструкций для компилируемых приложений. По сути, как и в случае со стандартным языком Ассемблер, исходный код заранее компилируется в этот формат. Поскольку приложение состоит из низкоуровневых инструкций, оно выполняется быстрее, чем JavaScript, который хранится в текстовом виде и интерпретируется в исполняемые инструкции по мере чтения.

В .NET имеется собственная реализация WebAssembly под названием Blazor. Это позволяет разработчикам писать внутрибраузерные приложения на языке C#. Таким образом, располагая возможностью создавать высокопроизводительные приложения для браузеров, разработчики .NET могут заниматься полнофункциональной разработкой без необходимости изучать JavaScript и его бесчисленные современные реализации.

Но Blazor — это нечто большее, чем просто технология, облегчающая процесс преобразования .NET-кода в формат WebAssembly. Помимо этого, она предлагает простой способ реализации хостинга приложений WebAssembly внутри стандартных приложений ASP.NET Core. Существует также разновидность Blazor, известная как Blazor Server. При ее использовании ваш код будет идентичен коду Blazor WebAssembly. Но компилятор будет работать по-другому. Скомпилированный код будет выполняться на сервере, но браузер и сервер будут взаимодействовать друг

с другом в реальном времени, поэтому элементы на странице будут мгновенно реагировать на события, происходящие на сервере. А код на сервере будет мгновенно запускаться событиями в браузере.

В последующих разделах главы мы обсудим все эти концепции, а также рассмотрим новые возможности Blazor, добавленные в .NET 7.

Структура

В этой главе мы рассмотрим новейшие возможности Blazor в ASP.NET Core 7 и затронем следующие темы:

- введение в Blazor;
- обзор Blazor WebAssembly;
- хостинг приложений Blazor WebAssembly в ASP.NET Core;
- настройка Blazor Server.

Цели

К концу этой главы вы научитесь использовать .NET Blazor во всех его проявлениях. Вы познакомитесь с созданием автономных приложений Blazor WebAssembly, а также узнаете, как реализовать хостинг приложения Blazor WebAssembly внутри стандартного приложения ASP.NET Core. Вы узнаете, как и в каких случаях следует использовать Blazor Server, и ознакомитесь с новейшими возможностями Blazor.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из перечисленных компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Общие сведения о Blazor

Blazor существует в двух вариантах — Blazor WebAssembly и Blazor Server. Для каждого из них предусмотрен отдельный шаблон проекта и по-разному выполняется компиляция. Blazor Server представляет собой не что иное, как размещенное на сервере веб-приложение ASP.NET Core с некоторыми дополнительными библиоте-

ками, а Blazor WebAssembly компилируется в исполняемый файл, который целиком размещается в браузере.

Однако, несмотря на различия в структуре приложения, у обоих вариантов Blazor совершенно одинаковый синтаксис модулей, которые называют Razor Components (Razor-компонентами). Не следует путать их с Razor Pages. Оба они служат для генерации HTML-кода, который затем отображается в браузере. Но файл Razor Pages имеет расширение cshtml, а файл Razor Components — razor. Кроме того, хотя оба типа модулей основаны на синтаксисе Razor, есть некоторые ключевые слова, которые работают только в Razor Components и не работают в Razor Pages. Обратная картина наблюдается и для некоторых других ключевых слов. Например, ключевое слово `@code` применимо только к компонентам Razor, в то время как `@model` — только в Razor Pages.

Razor-компоненты могут ссылаться на другие Razor-компоненты так же, как это происходит с элементами HTML. Имеются открывающие и закрывающие теги, а внутри элементов можно разместить дополнительное содержимое, как это делается в HTML и XML. Также можно использовать свойства элементов аналогично атрибутам HTML. Но соглашения об именовании отличаются. Если HTML-элементы обычно записывают строчными буквами, то Razor-компоненты оформлены в стиле PascalCase, т. е. каждое слово начинается с прописной буквы, а все остальные буквы остаются неизменными. Например, если имеется Razor-компонент с именем `Counter` и у него есть свойство `IncrementBy`, то из других Razor-компонентов на него можно сослаться следующим образом:

```
<Counter IncrementBy="5" />
```

Теперь рассмотрим несколько примеров Razor-компонентов, чтобы понять, как они устроены.

Примеры Razor-компонентов

При создании нового проекта приложения на основе шаблона Blazor WebAssembly или Blazor Server в нем уже будут присутствовать в качестве примеров некоторые Razor-компоненты. В обоих проектах в папке Pages имеется файл Counter.razor со следующим содержимым:

```
@page "/counter"
<PageTitle>Counter</PageTitle>
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</ button>
@code {
    private int currentCount = 0;
    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Рассмотрим его по частям.

@page

Файл начинается с директивы `@page`, которая представляет путь доступа к компоненту, заданный по умолчанию. Если ввести URL-адрес базового приложения, а затем `/counter`, то браузер перейдет к этому компоненту.

Далее следует ссылка на Razor-компонент `PageTitle` с внутренним содержимым `Counter`. Этот Razor-компонент похож на элемент HTML/XML и оформлен в стиле PascalCase.

Далее у нас есть элемент параграфа, часть текста которого привязана к полю `currentCount`. Если мы посмотрим на содержимое директивы `@code`, то увидим приватное целочисленное поле с таким именем. Обратите внимание, что элементы разметки в Razor-компоненте имеют полный доступ к приватным членам его кода. Таким образом, при каждом изменении значения `currentCount` в коде содержимое элемента параграфа также будет изменено.

@onclick

Далее следует кнопка с обработчиком событий `@onclick`. Как и ранее, мы можем сказать, что это обработчик событий Razor, т. к. его имя начинается с символа `@`. Этот обработчик событий обращается к методу `IncrementCount` внутри директивы `@code`. Этот метод вызывается при нажатии кнопки.

@code

Наконец, в файле есть директива `@code`, содержащая код на языке C#, который мы можем вызвать из разметки. При наличии членов с модификатором доступа `public` и атрибутом `[Parameter]` к ним можно обращаться из других Razor-компонентов. Однако мы можем получить доступ к этому коду и из нашего обычного кода на C#. Каждый Razor-компонент представлен в виде класса, который наследует от `ComponentBase` из пространства имен `Microsoft.AspNetCore.Components`. Этот класс будет иметь то же имя, что и файл. В данном случае, поскольку файл носит имя Counter.razor, данный класс будет доступен как `Counter`. Внутри других Razor-компонентов на него можно ссылаться так: `<Counter />`.

Razor-компонент не представляет всю страницу целиком. Подобно представлениям Razor Pages и MVC, он обернут в совместно используемую компоновку. Именно эта компоновка содержит все остальные элементы страницы, такие как HTML-заголовки и меню навигации. Однако формат компоновок отличается в зависимости от того, какой шаблон проекта задан — Blazor WebAssembly или Blazor Server. Оба варианта мы рассмотрим далее в этой главе. А пока давайте пройдемся по всем ключевым словам, которые присутствуют в Razor-компонентах.

Ключевые слова Razor в Razor-компонентах

Мы уже рассмотрели директивы `@page`, `@onclick` и `@code`. Мы также обсудили, как с помощью одного символа `@` можно ссылаться на элементы внутри обычного кода

C#. Теперь давайте рассмотрим другие ключевые слова, которые можно использовать в Razor-компонентах.

@using

В контексте Razor-компонентов это ключевое слово предназначено для вставки пространств имен в блок `@code` аналогично тому, как ключевое слово `using` служит для вставки ссылок на пространства имен в классы C#. Выражения с этой директивой следуют непосредственно за директивой `@page`. Вот пример такого выражения:

```
@using HostedBlazorWasmDemo.Client.Models;
```

@implements

Эта директива используется, если вы хотите, чтобы ваш Razor-компонент реализовывал какой-либо интерфейс. Например, для реализации интерфейса `IDisposable` эту директиву можно задать следующим образом:

```
@implements IDisposable
```

@inherits

Эту директиву можно добавить, если требуется, чтобы ваш Razor-компонент наследовал от конкретного класса C#. Пример применения:

```
@inherits SomeBaseClass
```

@inject

Эта директива предназначена для добавления любых зависимостей в Razor-компонент, зарегистрированный в файле `Program.cs` при запуске приложения. В следующем примере выполняется внедрение в компонент экземпляра `NavigationManager`:

```
@inject NavigationManager
```

@layout

Эта директива задает элемент компоновки, который может быть применен к Razor-компоненту. Компоновку обычно настраивают для всего приложения, однако эта директива позволяет применить определенный набор многократно используемых элементов к конкретному Razor-компоненту. Вот пример этой директивы в начале Razor-компонента:

```
@layout CustomLayout
```

@namespace

С помощью этой директивы можно переопределить стандартное пространство имен Razor-компонента (соответствующее структуре папок проекта), задав для него другое пространство имен. Вот пример:

```
@namespace SomeNamespace
```

@preservewhitespace

По умолчанию Blazor удаляет лишние пробелы из HTML-разметки для повышения производительности. Однако если пробелы необходимо сохранить, то это можно сделать следующим образом:

```
@preservewhitespace true
```

@attributes

Эта директива позволяет динамически вставлять в HTML-элемент произвольные HTML-атрибуты из словаря C#. Например, можно задать следующий элемент:

```
<input id="mainText" @attributes="Attributes" />
```

Пример оформления поля словаря:

```
private Dictionary<string, object> Attributes { get; set; } =
        new()
        {
            { "maxlength", "10" },
            { "placeholder", "Default value" },
            { "required", "required" },
            { "class", "green-textbox" }
        };
```

@bind

Это ключевое слово позволяет привязать HTML-элемент к переменной из кода C#. При его применении изменение входной величины ведет к изменению значения переменной и наоборот. Например, если имеется переменная типа `string` с именем `name`, то ее можно привязать к элементу ввода следующим образом:

```
<input @bind="name" />
```

@ref

Эта директива требуется в тех случаях, когда необходимо передать команды определенной ссылке другого Razor-компонента. Для применения этой директивы необходимо задать целевой Razor-компонент в качестве переменной в коде, а затем вставить имя этой переменной с ключевым словом в разметку.

Например, если у вас есть Razor-компонент с именем `ChildComponent`, вы можете объявить его как приватное поле внутри директивы `@code` следующим образом:

```
private ReferenceChild? child;
```

Затем объявите его в разметке, чтобы убедиться, что вы ссылаетесь именно на это поле, а не на новый экземпляр данного компонента:

```
<ReferenceChild @ref="child" />
```

@typeparam

Эта директива позволяет использовать в Razor-компонентах обобщенные типы. Вот пример:

```
@typeparam TEntity where TEntity : class
```

На этом завершим общий обзор возможностей, применимых как к Blazor WebAssembly, так и к Blazor Server, и рассмотрим процесс настройки проекта Blazor WebAssembly.

Обзор Blazor WebAssembly

Новый проект Blazor WebAssembly можно создать либо на основе соответствующего шаблона в IDE, либо выполнив следующую команду, которая создаст проект с именем `BlazorWasmDemo`:

```
dotnet new blazorwasm -o BlazorWasmDemo
```

WASM — это распространенная аббревиатура для WebAssembly. Именно поэтому она присутствует в указанной выше команде.

После создания проекта рассмотрим его структуру. Первое, на что мы обратим внимание, — это компоновка самого верхнего уровня, которая задана в файле index.html, расположенном в папке wwwroot. Этот обычный HTML-файл задает структуру приложения Blazor WebAssembly: он используется в качестве внешней оболочки всеми Blazor-компонентами. Здесь определяются заголовки HTML, а также любые элементы-заместители, дополнительные скрипты и ссылки на стили.

Заслуживает внимания ссылка на файл скрипта _framework/blazor.webassembly.js. В этом файле содержится весь код JavaScript, необходимый для создания Razor-компонентов на стороне клиента и обеспечения взаимодействия между ними и обычным JavaScript. Поэтому, если потребуется добавить какой-либо код JavaScript, предназначенный для работы с Blazor, вам нужно будет добавить этот код (или ссылку на JS-файл) после ссылки на этот файл скрипта.

Далее, в корневой папке проекта расположен файл _Imports.razor, необходимый для перечисления всех пространств имен, которые мы хотим сделать доступными для импорта в Razor-компоненты.

Затем, имеется файл App.razor, который выступает в качестве корневого элемента самого приложения Blazor. Именно этот файл определяет, какой файл компоновки будет использоваться и что отображать, если указанный в браузере путь не совпадает ни с одним Razor-компонентом. Кроме того, именно в этот файл можно добавить собственную логику реагирования на ошибки, например, если пользователь не авторизован. Этот файл регистрируется внутри файла Program.cs и вставляется в соответствующий HTML-элемент на следующей строке:

```
builder.RootComponents.Add<App>("#app");
```

По умолчанию это будет элемент, в котором атрибут `id` соответствует приложению внутри файла index.html.

По умолчанию файл App.razor указывает на `MainLayout` как на компоновку. Эту компоновку можно найти в файле MainLayout.razor в папке Shared. Razor-компоненты используют эту компоновку по умолчанию, если она не переопределена с помощью директивы `@layout`. Эта компоновка имеет несколько HTML-элементов и Razor-компонентов верхнего уровня. Содержимое Razor-компонента, к которому мы переходим посредством навигации, заменяет директиву `@Body` в компоновке.

Далее идут отдельные Razor-компоненты. Они помещены в папку Pages и имеют расширение razor. В остальном же точкой входа в скомпилированную часть приложения Blazor WebAssembly является файл Program.cs, как и в большинстве приложений .NET.

На этом обзор базовой структуры проекта Blazor WebAssembly завершен. До сих пор мы рассматривали сценарии, в которых разметка и код помещены в один файл. Однако при наличии сложной разметки (кода) такая схема не всегда обеспечивает наилучшую читабельность. Поэтому Blazor позволяет держать разметку и код в разных файлах. Такой подход известен как "отделение кода программной части", и мы рассмотрим его далее.

Отделение кода программной части

Если вы хотите вынести код на C# из файла Razor в отдельный файл, то вам следует создать файл с тем же именем, что и у файла Razor, но присвоить ему дополнительное расширение cs. Чтобы продемонстрировать это, разделим содержимое файла FetchData.razor между двумя отдельными файлами. Для этого создадим файл FetchData.razor.cs в той же папке, что и исходный файл FetchData.razor, и наполним его следующим содержимым:

```
using Microsoft.AspNetCore.Components;
using System.Net.Http.Json;
namespace BlazorWasmDemo.Pages;
public class FetchDataBase : ComponentBase
{
    [Inject] HttpClient Http { get; set; }
    protected WeatherForecast[]? forecasts;
    protected override async Task OnInitializedAsync()
    {
        forecasts =
await Http.GetFromJsonAsync<WeatherForecast[]> ("sample-data/weather.json");
    }
    public class WeatherForecast
    {
        public DateOnly Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
        public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
    }
}
```

Мы используем наследование от класса `ComponentBase`, поскольку каждый Razor-компонент должен наследовать от `ComponentBase`. Обратите внимание, что мы также дали нашему классу имя `FetchDataBase`, отличающееся от имени исходного компонента `FetchData`. Это связано с тем, что Blazor все равно будет рассматривать эти два файла как два отдельных класса. С функциональной точки зрения отделение кода программной части реализуется посредством наследования. Именно поэтому мы изменили модификатор доступа для всех членов с `private` на `protected`.

Обратите внимание на атрибут `[Inject]`, предназначеный для внедрения сервисов в файлы кода программной части Razor-компонентов вместо внедрения посредством конструктора. Этот атрибут эквивалентен директиве `@inject` в Razor-компонентах.

В этом классе при инициализации компонента мы загружаем некоторые данные из файла weather.json, который находится в папке sample-data и имеет следующее содержимое:

```
{
    {
        "date": "2022-01-06",
        "temperatureC": 1,
        "summary": "Freezing"
    },
    {
        "date": "2022-01-07",
        "temperatureC": 14,
        "summary": "Bracing"
    },
    {
        "date": "2022-01-08",
        "temperatureC": -13,
        "summary": "Freezing"
    },
    {
        "date": "2022-01-09",
        "temperatureC": -16,
        "summary": "Balmy"
    },
    {
        "date": "2022-01-10",
        "temperatureC": -2,
        "summary": "Chilly"
    }
]
```

Теперь заменим содержимое файла FetchData.razor следующим:

```
@page "/fetchdata" @inherits FetchDataBase
<PageTitle>Weather forecast</PageTitle>
<h1>Weather forecast</h1>
```

```
<p>This component demonstrates fetching data from the server.</p>
@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}
```

Как видим, здесь полностью удален раздел `@code`. И теперь этот файл наследует от класса `FetchDataBase`. В остальном он функционально идентичен прежнему.

Еще одна интересная особенность Blazor — способность взаимодействовать с JavaScript на веб-странице, которую мы рассмотрим далее.

Взаимодействие с JavaScript

Поддержка взаимодействия с JavaScript в Blazor позволяет как коду на C# вызывать методы JavaScript, так и коду на JavaScript вызывать методы C#. Продемонстрируем, как это работает. Сначала добавим следующий элемент скрипта в файл index.html из папки wwwroot в любом месте ниже ссылки на библиотеку Blazor JavaScript:

```
<script>
window.displayAlert = () => {
    alert('Counter successfully increased');
}
</script>
```

Обратите внимание, что метод `displayAlert` добавляется в область видимости окна. Это область, с которой работает Blazor. В противном случае метод будет недоступен для кода Blazor.

Затем откроем файл Counter.razor из папки Pages и заменим его содержимое на следующее:

```
@page "/counter"
@inject IJSRuntime JS
<PageTitle>Counter</PageTitle>
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</ button>
@code {
    private int currentCount = 0;
    private async Task IncrementCount()
    {
        currentCount++;
        await JS.InvokeVoidAsync("displayAlert");
    }
}
```

Чтобы обеспечить взаимодействие с JavaScript, мы внедрили объект `IJSRuntime`, который регистрируется автоматически вместе со всеми остальными зависимостями Blazor. Этот сервис позволяет нам вызвать любой подходящий метод JavaScript. В нашем случае мы вызываем метод `displayAlert` через `InvokeVoidAsync`, т. к. он не возвращает никакого значения. Однако, если бы мы хотели получить значение из метода JavaScript, мы бы вместо этого использовали `InvokeAsync`. Кроме того, если метод JavaScript принимает какие-либо параметры, то мы можем передать их, указав после имени метода. Например, если у нас есть метод `getFullName`, принимающий два строковых параметра и возвращающий строку, мы можем вызвать его следующим образом:

```
var fullName = await JS.InvokeAsync<string>("getFullName", "John", "Smith");
```

Чтобы вызвать метод C# из JavaScript, этот метод должен быть публичной статической задачей с атрибутом `[JSInvokable]`. Затем, чтобы вызвать данный метод из JavaScript, необходимо использовать объект `DotNet`, который имеется в библиотеке Blazor JavaScript. Вызов должен быть построен следующим образом:

```
DotNet.invokeMethodAsync('{Namespace Name}', '{.NET Method Name}', {parameters});
```

Далее мы рассмотрим передачу параметров из Razor-компонента его дочерним компонентам.

Передача параметров в Razor-компоненты

Чтобы передать параметры из Razor-компонента его дочернему компоненту, последний должен иметь публичный тип и должен быть помечен атрибутом `[Parameter]`. Например, мы можем изменить логику приращения для нашего компо-

нента `Counter`. Вместо того чтобы увеличивать его на единицу, мы можем сделать так, чтобы при нажатии кнопки он увеличивался на заданное число. Для этого мы изменили содержимое компонента, добавив в него свойство `IncrementBy`, помеченное атрибутом `[Parameter]`:

```
<PageTitle>Counter</PageTitle>
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</ button>
@code {
    private int currentCount = 0;
    private async Task IncrementCount()
    {
        currentCount += IncrementBy;
        await JS.InvokeVoidAsync("displayAlert");
    }
    [Parameter]
    public int IncrementBy { get; set; } = 1;
}
```

Теперь нам остается лишь задать значение этого свойства так же, как мы задаем значения атрибутов в HTML. Чтобы продемонстрировать это, мы изменим содержимое нашего файла Index.razor следующим образом:

```
@page "/"
<Counter IncrementBy="5" />
```

Теперь, если мы запустим наше приложение, то компонент `Counter` будет отображаться на главной странице. И если мы нажмем кнопку, то значение увеличится на 5.

До сих пор мы рассматривали только те примеры, в которых использовался механизм компиляции, принятый для Blazor WebAssembly по умолчанию, — компиляция в промежуточный язык (IL) .NET, — который реализует динамическую компиляцию (JIT) в машинные инструкции при выполнении кода. Таким образом, когда приложение запускается, каждый элемент его логики интерпретируется в машинную инструкцию низкого уровня по мере считывания.

Но в Blazor WebAssembly есть и другой механизм компиляции: предварительная компиляция (Ahead-of-Time, AoT), которую мы рассмотрим далее.

Предварительная компиляция

При предварительной компиляции приложение предварительно компилируется в набор инструкций, соответствующих конкретной аппаратной платформе. При этом не нужен промежуточный язык и отсутствует интерпретация каждой инструкции. Аппаратные средства напрямую распознают содержимое исполняемого файла.

Ввиду отсутствия промежуточного этапа интерпретации выполнение такого приложения происходит быстрее. Однако есть и обратная сторона. Процесс подготовки исполняемого файла для конкретной аппаратной платформы приводит к увеличе-

нию размера файла. Следовательно, увеличится время первоначальной загрузки файла.

Чтобы подготовить среду разработки, вам необходимо установить рабочую нагрузку `wasm-tools`, что можно сделать путем выполнения на консоли следующей команды:

```
dotnet workload install wasm-tools
```

Затем нам нужно будет добавить в файл .csproj следующий раздел:

```
<PropertyGroup>
    <RunAOTCompilation>true</RunAOTCompilation>
</PropertyGroup>
```

Поскольку в результате предварительной компиляции создается автономный исполняемый файл, который не имеет каких-либо внешних зависимостей от .NET, вы не можете запустить его в режиме **Debug**. Поэтому, чтобы скомпилировать ваш проект в исполняемый файл, вам необходимо опубликовать его в режиме **Release**. Это можно сделать, выполнив следующую команду из папки проекта:

```
dotnet publish -c Release
```

Далее мы кратко рассмотрим альтернативный шаблон проекта Blazor WebAssembly, который также можно использовать.

Пустой шаблон Blazor WebAssembly

Если вы воспользуетесь заданным по умолчанию шаблоном проекта Blazor WebAssembly, то он предоставит вам все необходимые зависимости и примеры. Но затем вам придется удалить все добавленные по умолчанию Razor-компоненты, т. к. они предназначены исключительно для демонстрационных целей и не предоставляют никаких полезных функций.

Но если вы уже знакомы с Blazor, то можете просто создать пустой проект Blazor, в котором эти демонстрационные компоненты отсутствуют. Этот шаблон называ-

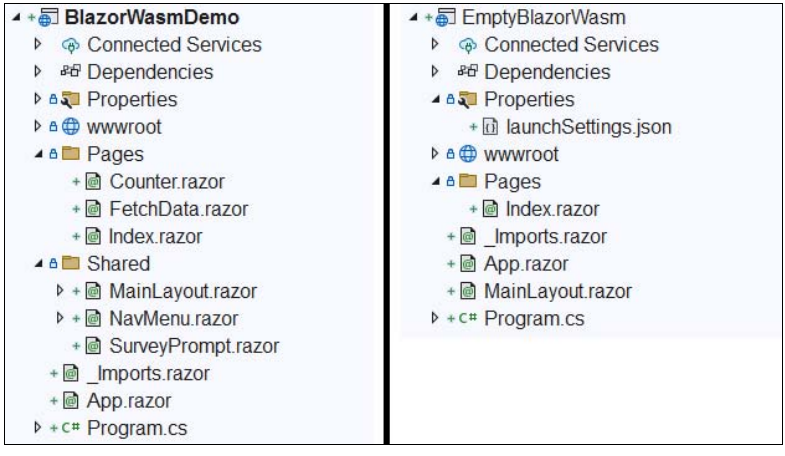


Рис. 7.1. Разница между проектом Blazor WebAssembly, принятым по умолчанию, и пустым проектом

ется `blazorwasm-empty` и может быть применен к новому проекту с помощью следующей команды:

```
dotnet new blazorwasm-empty -o EmptyBlazorWasm
```

После создания проект будет содержать меньше файлов, чем исходный проект Blazor WebAssembly. Разница продемонстрирована на рис. 7.1.

До сих пор мы рассматривали Blazor WebAssembly как отдельное автономное приложение. Но мы также можем разместить его внутри приложения ASP.NET Core и автоматически загружать как одно из представлений.

Хостинг приложений Blazor WebAssembly в ASP.NET Core

Вы можете создать проект Blazor WebAssembly с хостингом в приложении ASP.NET Core. Для этого нужно лишь выбрать соответствующую опцию при создании проекта на основе этого шаблона. Либо вы можете указать флаг `--hosted` для команды в CLI. Например, чтобы создать хостируемое приложение Blazor WebAssembly с именем `HostedBlazorWasmDemo`, можно выполнить следующую команду:

```
dotnet new blazorwasm -o HostedBlazorWasmDemo --hosted
```

Эта команда создаст решение с именем `HostedBlazorWasmDemo`. В папке решения будут размещены три папки проекта — Client, Server и Shared:

- Client — это наше приложение Blazor WebAssembly;
- Server — это приложение ASP.NET Core, реализующие хостинг приложения WebAssembly;
- Shared — это библиотека классов, содержащая компоненты, используемые обоими приложениями.

Если для приложения Blazor WebAssembly используется хостинг в ASP.NET Core, потребуется запустить проект ASP.NET Core, чтобы увидеть всю конструкцию в действии. Вам больше не нужно будет отдельно запускать приложение Blazor. В нашем конкретном случае для запуска сервера и WebAssembly необходимо запустить проект в папке Server.

Мы также можем добавить хостируемое приложение Blazor WebAssembly в существующее приложение ASP.NET Core. Рассмотрим структуру нашего проекта, чтобы узнать, как это сделать.

Добавление хостируемого приложения Blazor WebAssembly к существующему приложению ASP.NET Core

Для настройки хостинга приложения Blazor WebAssembly необходимо, чтобы в приложении ASP.NET Core был установлен следующий пакет NuGet: Microsoft. AspNetCore.Components.WebAssembly.Server.

Затем следует сослаться на проект Blazor WebAssembly из приложения ASP. NET Core, чтобы приложение WebAssembly было доступно до готовности серверного приложения. Например, в нашем файле HostedBlazorWasmDemo.Server.csproj имеется ссылка на проект HostedBlazorWasmDemo.Client.

Затем необходимо включить хостинг WebAssembly, добавив в файл Program.cs проекта ASP.NET Core следующую строку:

```
app.UseBlazorFrameworkFiles();
```

Далее, если в вашем серверном приложении не настроена индексная страница по умолчанию, то Blazor WeBAssembly возьмет на себя управление и будет выполнять функции индексной страницы приложения. Если приложение Blazor не предполагается размещать на главной странице, то вам нужно будет добавить файл *.cshtml для представления Razor Pages или MVC, в котором вы хотите разместить свое Blazor-приложение, и внести в него следующее содержимое:

```
@using { name of the Blazor application namespace that hosts the component that you need };
    <component type="typeof({ name of the Razor Component you want to host
})"
        render-mode="WebAssemblyPrerendered" />
    <script src="_framework/blazor.webassembly.js"></script>
```

На этом общий обзор хостинга приложений Blazor в проектах ASP. NET Core завершен. Поскольку в такой архитектуре присутствуют как клиентские, так и серверные компоненты, сейчас самое время поговорить о формах Blazor и их проверке.

Проверка форм в Blazor

В Blazor вы можете использовать формы для отправки редактируемых данных на сервер так же, как на стандартных HTML-страницах. Данные также можно проверить с помощью атрибутов проверки, которые представляют собой специальные классы, способные выполнять любую логику проверки. Эти классы наследуют от класса `ValidationAttribute` пространства имен `System.ComponentModel.DataAnnotations`. Начиная с .NET 7, вы также можете передавать сервисы в эти классы путем внедрения зависимостей.

Создадим файл MaxIncrementValidator.cs в папке проекта Client и наполним его следующим содержимым:

```
using System.ComponentModel.DataAnnotations;

namespace HostedBlazorWasmDemo.Client;
public class MaxIncrementValidator : ValidationAttribute
{
    protected override ValidationResult? IsValid(object? value, ValidationContext
                                                                    validationContext)
    {
        var paramValuesConfig = validationContext.GetRequiredService<ParamValuesConfig>();
        if ((int)value > paramValuesConfig.MaxIncrementValue)
```

```
            return new ValidationResult($"Values greater than
                                                    {paramValuesConfig.MaxIncrementValue}
                are not allowed!", new[] { validationContext.MemberName });
        return ValidationResult.Success;
    }
}
public class ParamValuesConfig
{
    public int MaxIncrementValue { get; set; } = 5;
}
```

В этом классе мы проверяем, что целое значение не превышает определенной величины. Величина, с которой мы ее сравниваем, извлекается из экземпляра сервиса `ParamValuesConfig` и по умолчанию равна 5. Мы разрешаем этот сервис через параметр `ValidationContext`. Чтобы зарегистрировать этот сервис, достаточно добавить в файл Program.cs следующую строку:

```
builder.Services.AddScoped<ParamValuesConfig>();
```

Далее нужно будет добавить в нашу форму модель, чтобы затем применить к ней этот атрибут. В нашем примере мы создаем в проекте папку Models и добавляем в нее файл CounterModel.cs со следующим содержимым:

```
namespace HostedBlazorWasmDemo.Client.Models;
public class CounterModel
{
    [MaxIncrementValidator]
    public int IncrementBy { get; set; } = 1;
}
```

Как видите, у нас есть целочисленное поле `IncrementBy`, к которому мы применяем данный атрибут.

Теперь заменим содержимое файла Counter.razor следующим:

```
@page "/counter"
@using HostedBlazorWasmDemo.Client.Models;
<PageTitle>Counter</PageTitle>
<h1>Counter</h1>
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</ button>
<EditForm Model="@counterModel" >
    <DataAnnotationsValidator />
    <ValidationSummary />
    <label for="incrementBy">Increment by:</label>
    <InputNumber id="incrementBy" @bind-Value="counterModel.IncrementBy"
/>
</EditForm>
```

Здесь имеется Razor-компонент `EditForm`, который представляет собой форму Blazor. Она содержит компоненты `DataAnnotationsValidator` и `ValidationSummary`, которые предоставляют возможность проверить данные в форме и отобразить ошибку про-

верки. Имеется также Razor-компонент `InputNumber`, привязанный к полю `IncrementBy` экземпляра `CounterModel`.

Наконец, нам нужно внести некоторые изменения в блок `@code`. Он должен выглядеть следующим образом:

```
@code {
    private CounterModel counterModel = new();
    private int currentCount = 0;
    private async Task IncrementCount()
    {
        currentCount += counterModel.IncrementBy;
    }
}
```

Если теперь мы соберем и запустим проект `Server` и перейдем на страницу `Counter`, то сможем протестировать логику проверки. Если мы введем в поле `IncrementBy` любое число больше 5, то должны будем увидеть ошибку проверки, как показано на рис. 7.2.



Рис. 7.2. Ошибка проверки в форме Blazor

На этом завершим обзор проверки форм Blazor и далее рассмотрим функции навигации и возможность передачи состояния от одного Razor-компонента другому.

NavigationManager и передача состояния между страницами

Blazor сопровождается классом `NavigationManager`, который дает возможность переходить от одной страницы к другой. Начиная с версии .NET 7, он позволяет также передавать информацию между страницами.

Чтобы продемонстрировать это, мы заменили содержимое файла Index.razor следующим:

```
@page "/"
@inject NavigationManager NavigationManager
<button class="btn btn-primary" @onclick="NavigateToCounterComponent">
    Go to Counter
</button>
```

```
@code {
    private void NavigateToCounterComponent()
    {
        NavigationManager.NavigateTo("counter",
        new NavigationOptions
        {
        HistoryEntryState = "Navigated here from Index page"
        });
    }
}
```

Итак, на нашей индексной странице теперь есть кнопка, которая перенаправляет пользователя на страницу `Counter`. Предполагается, что страница `Counter` получает сообщение о том, что был выполнен переход с индексной страницы.

Теперь мы внесем некоторые изменения в файл Counter.razor, чтобы проверить, как все это работает. Во-первых, мы внедрим в него экземпляр `NavgiationManager`, поместив следующее выражение непосредственно после директивы `@using`:

```
@inject NavigationManager NavigationManager
```

Затем в любом месте области разметки добавим следующий элемент:

```
<p>History entry state: @NavigationManager.HistoryEntryState</p>
```

Теперь, если запустить приложение `Server` и щелкнуть на вновь добавленной кнопке навигации на главной странице, то должен появиться элемент `Counter` с сообщением, переданным со страницы `Index`, как показано на рис. 7.3.

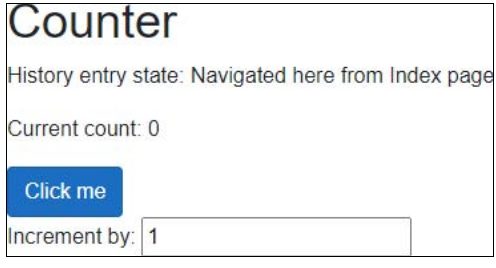


Рис. 7.3. Состояние навигации, переданное из другого Razor-компонента

На этом завершим обзор хостинга приложений Blazor WebAssembly и далее рассмотрим основы работы Blazor Server, продемонстрировав некоторые дополнительные возможности Blazor.

Настройка Blazor Server

Теперь создадим проект Blazor Server под названием BlazorServerDemo. Мы можем сделать это либо с помощью шаблона проекта Blazor Server в IDE, либо выполнив следующую команду CLI:

```
dotnet new blazorserver -o BlazorServerDemo
```

После создания проекта мы можем посмотреть, чем его структура отличается от Blazor WebAssembly. Прежде всего, следует отметить, что данный проект представляет собой не что иное, как стандартное приложение ASP.NET Core с добавленными к нему дополнительными зависимостями Blazor. Он запускается с сервера. В него можно добавлять любые стандартные компоненты ASP.NET Core, такие как компоненты MVC, Razor Pages, контроллеры API и т. д.

Настройка компонентов Blazor Server осуществляется путем добавления в файл Program.cs следующей строки:

```
builder.Services.AddServerSideBlazor();
```

Кроме того, в конвейер промежуточного ПО необходимо добавить следующие шаги:

```
app.MapBlazorHub();
app.MapFallbackToPage("/_Host");
```

Также есть и другие отличия в структуре проекта. Вы могли заметить, что в папке wwwroot нет файла index.html. Вместо этого HTML базового уровня располагается в файле _Host.cshtml, который находится в папке Pages. Имя этого файла является параметром по умолчанию внутри вызова `app.MapFallbackToPage` из файла Program.cs.

В остальном структура аналогична структуре проекта Blazor WebAssembly. Корневой элемент приложения Blazor — файл App.razor. Файл MainLayout.razor в папке Shared содержит компоновку по умолчанию и т. д.

В самих Razor-компонентах принят тот же синтаксис, что и в проекте Blazor WebAssembly. Именно поэтому на библиотеки классов Blazor можно ссылаться из приложений Blazor обоих типов. Но эти компоненты больше не компилируются в исполняемые браузером сборки. Они компилируются в классы серверной части, и компилятор выполняет дополнительную работу по генерации кода HTML и JavaScript, которые будут взаимодействовать с этими классами. В качестве механизма связи между браузером и сервером применяется SignalR. Более подробно мы рассмотрим эту технологию в *главе 8*.

Поскольку вы знакомы со структурой проектов Blazor WebAssembly и Blazor Server, мы рассмотрим возможности Blazor по генерации пользовательских HTML-элементов, которые несколько отличаются в зависимости от типа используемого приложения Blazor.

Пользовательские элементы в Blazor

Спецификация HTML позволяет добавлять в документ пользовательские элементы. Вы можете преобразовывать компоненты Blazor в чисто пользовательские HTML-элементы. Для этого необходимо установить следующий пакет NuGet:

```
Microsoft.AspNetCore.Components.CustomElements
```

Затем необходимо добавить следующую ссылку на скрипт в файл index.html в случае Blazor WebAssembly или в файл _Host.cshtml для Blazor Server:

```
<script src="/_content/Microsoft.AspNetCore.Components.CustomElements/
BlazorCustomElements.js"></script>
```

Обратите внимание, что, как и в любом другом скрипте, взаимодействующем с Blazor, эта ссылка должна быть вставлена после ссылки на основную библиотеку Blazor JavaScript.

Далее действия будут отличаться в зависимости от типа проекта Blazor. Если бы потребовалось превратить наш Razor-компонент `Counter` в HTML-элемент `custom-counter`, вот как это можно сделать в файле Program.cs приложения Blazor Server:

```
builder.Services.AddServerSideBlazor(options =>
{
    options.RootComponents.RegisterCustomElement<Counter>("custom-counter");
});
```

А вот так можно было поступить в файле Program.cs приложения Blazor WebAssembly:

```
builder.RootComponents.RegisterCustomElement<Counter>("custom-counter");
```

Затем, чтобы применить этот пользовательский HTML-элемент, следует просто вставить его в разметку любой из наших страниц, точно так же, как мы вставили бы стандартный HTML-элемент:

```
<custom-counter></custom-counter>
```

Далее мы рассмотрим жизненный цикл Razor-компонента и методы, представляющие события жизненного цикла, которые можно переопределить.

Жизненный цикл Razor-компонента

У класса `ComponentBase` есть переопределяемые методы, которые запускаются при определенных событиях жизненного цикла. Переопределив их, можно добавить некоторую пользовательскую логику на различных этапах загрузки. Эти методы могут быть синхронными или асинхронными.

Первый из таких методов — `SetParameters`. С его помощью вы получаете доступ к входному параметру `ParemeterView`, который позволяет перехватывать и изменять параметры, устанавливаемые для Razor-компонента.

Следующий метод — `OnInitialized`. Он запускается, когда код внутри компонента уже инициализирован, но компонент еще не отрисован.

Еще один метод — `OnParametersSet`. Он позволяет применять любую пользовательскую логику после того, как члены класса компонента будут заполнены значениями, переданными им в качестве параметров.

Наконец, имеется метод под названием `OnAfterRender`. На этом этапе компонент полностью готов, отрисован, и к нему можно применить любую дополнительную логику, прежде чем пользователь начнет взаимодействовать с ним.

Чтобы продемонстрировать это, мы можем передать следующий блок `@code` в файл Index.razor приложения Blazor Server. Мы воспользуемся Blazor Server, т. к. это позволит нам легко читать консольный вывод на компьютере, где запущено приложение, и видеть, в каком порядке возникают события:

```
@code {
    public override async Task SetParametersAsync(ParameterView parameters)
    {
        Console.WriteLine($"Started setting parameters at {DateTime.Now}.");
        await base.SetParametersAsync(parameters);
    }
    protected override void OnInitialized()
    {
        Console.WriteLine($"Initialized at {DateTime.Now}.");
        base.OnInitialized();
    }
    protected override void OnParametersSet()
    {
        Console.WriteLine($"Parameters set at {DateTime.Now}.");
        base.OnParametersSet();
    }
    protected override void OnAfterRender(bool firstRender)
    {
        Console.WriteLine($"Completed rendering at {DateTime.Now}.");
        base.OnAfterRender(firstRender);
    }
}
```

На этом завершим обзор основ работы Blazor Server и рассмотрим пустой шаблон Blazor Server.

Пустой шаблон Blazor Server

Как и в случае с Blazor WebAssembly, в Blazor Server можно создать проект на основе пустого шаблона. Для этого можно через CLI выполнить следующую команду:

```
dotnet new blazorserver-empty -o EmptyBlazorServer
```

Теперь рассмотрим различия между стандартным шаблоном Blazor Server и пустым шаблоном. Главное отличие состоит в том, что в пустом шаблоне нет примеров компонентов Razor. Существует лишь базовый набор основных компонентов, необходимых для создания Blazor-приложения, как показано на рис. 7.4.

На этом обзор технологии Blazor, ее основных компонентов и возможностей завершен.

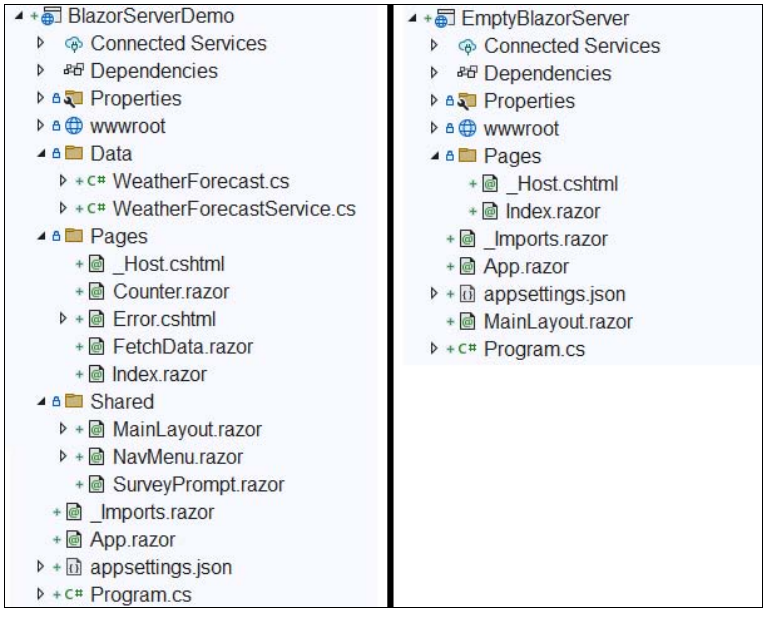


Рис. 7.4. Разница между шаблоном по умолчанию и пустым шаблоном Blazor Server

Заключение

В этой главе вы познакомились с основными возможностями платформы Blazor. Мы начали с рассказа о технологии WebAssembly и ее преимуществах. Мы также рассмотрели преимущества и недостатки метода предварительной компиляции (AoT-компиляции) в Blazor.

Теперь вы знакомы с хостингом приложений Blazor WebAssembly в стандартных приложениях ASP.NET Core и настройкой стандартных шаблонов проектов для хостируемых приложений Blazor. Вы также ознакомились с процессом добавления приложений Blazor для хостинга в уже существующие приложения ASP.NET Core.

Наконец, мы рассмотрели плюсы и минусы приложений Blazor Server, которые обеспечивают работу Blazor-кода без использования WebAssembly.

В следующей главе мы рассмотрим gRPC в .NET — эффективный механизм взаимодействия на основе HTTP/2 и HTTP/3.

Что следует помнить

- WebAssembly — это формат двоичных инструкций, позволяющий запускать скомпилированные приложения внутри браузеров.
- Blazor — это технология, позволяющая либо компилировать .NET-код в WebAssembly, либо создавать интерактивные веб-страницы.

- Blazor-приложение может быть скомпилировано в WebAssembly заранее, что делает это приложение более быстрым и уменьшает размер загружаемого файла.
- Для приложений Blazor WebAssembly также можно использовать хостинг в стандартном приложении ASP.NET Core.
- Приложения Blazor Server компилируются в отдельные компоненты для браузера и сервера, которые тесно связаны друг с другом.

Контрольные вопросы

1. **Что такое Razor-компонент?**
   а) Ключевое слово из Razor Pages;
   б) HTML-помощник из Razor Pages;
   в) Отдельный модуль внутри приложения Razor;
   г) Все вышеперечисленное.
2. **Каковы основные преимущества предварительной компиляции (AoT) в Blazor?**
   а) Уменьшение размера загружаемых файлов;
   б) Повышение скорости работы приложения;
   в) Повышение скорости запуска приложения;
   г) Все вышеперечисленное.
3. **Каковы недостатки использования Blazor Server?**
   а) Невозможность запуска приложения, когда клиент не в сети;
   б) Производительность зависит от скорости сети;
   в) Невозможность бессерверного развертывания;
   г) Все вышеперечисленное.
4. **Каковы преимущества использования Blazor Server?**
   а) Малый размер загрузки;
   б) Клиент и сервер могут разрабатываться независимо друг от друга;
   в) Возможность работы без браузера;
   г) Все вышеперечисленное.

Ответы

1. в.
2. б.
3. г.
4. а.

Основные термины

- **WebAssembly** — набор двоичных инструкций, позволяющий выполнять скомпилированный код в браузере.
- **Blazor** — технология, позволяющая либо компилировать .NET-код в WebAssembly, либо создавать интерактивные веб-приложения.
- **Razor-компонент** — отдельный модуль в приложении Blazor, наследующий от класса `ComponentBase`.
- **Blazor WebAssembly** — проект Blazor, компилируемый в WebAssembly.
- **Blazor Server** — проект Blazor, обеспечивающий взаимодействие между браузером и сервером в реальном времени.
- **Предварительная компиляция (AoT)** — предварительная компиляция проекта Blazor в чистый WebAssembly, без использования промежуточного .NET-кода.

Глава 8

SignalR и двусторонняя связь

Введение

Взаимодействие в реальном времени — обязательное условие для современных веб-приложений. Когда вы заходите на сайт социальной сети, то ожидаете получать немедленные уведомления о поступлении сообщений или появлении ответов на ваши публикации. От вас не требуется постоянно обновлять страницу. То же самое относится к приложениям для отслеживания доставки или любым другим приложениям, в которых клиент (браузер или мобильное приложение) должен получать обновления от сервера при наступлении определенных событий.

Традиционно создание полностью интерактивных приложений, в которых клиент и сервер могли бы взаимодействовать друг с другом в реальном времени, было непростой задачей. Существовало множество вариантов ее решения, но ни один из них не был легок в реализации. Это обусловлено тем, что реализуемая функция ведет себя не так, как это предусмотрено стандартной для Интернета моделью "запрос-ответ".

Обычно клиент посылает запрос на сервер, а сервер отвечает на запрос, отправляя ответ. Но в интерактивном приложении необходимо иметь возможность передавать информацию в обе стороны по мере необходимости. Когда вы наблюдаете за работой приложения для слежения за доставкой, контролируя перемещение водителя в реальном времени, именно сервер передает информацию клиенту, а не наоборот.

Существует несколько методов, но их не всегда легко реализовать. Один из таких методов — длительный опрос — заключается в отправке запроса на сервер и поддержании соединения в открытом состоянии до получения ответа. Этот метод относительно прост в реализации, но не слишком эффективен. Существует также протокол WebSocket, который позволяет клиенту и серверу устанавливать между собой постоянное соединение, обеспечивающее двусторонний обмен данными. Но написать его реализацию непросто, поскольку этот протокол работает с необработанными байтами, а код не является легкочитаемым.

Однако в ASP.NET Core есть библиотека SignalR, которая упрощает процесс создания приложений реального времени. Внутри этой библиотеки реализованы слож-

ные механизмы взаимодействия. Но она значительно упрощает программирование по сравнению с прямым использованием этих механизмов.

Библиотека SignalR не требует каких-либо пакетов NuGet, поскольку она уже включена в ASP.NET Core, и в выпуске .NET 7 получила ряд примечательных обновлений. Именно об этом мы и поговорим в этой главе.

Структура

В этой главе мы рассмотрим новейшие возможности SignalR в ASP.NET Core 7 и затронем следующие темы:

- обзор SignalR;
- создание хаба SignalR на сервере;
- JavaScript-клиент для SignalR;
- .NET-клиент для SignalR.

Цели

К концу этой главы вы узнаете, как создавать интерактивные веб-приложения реального времени с помощью библиотеки SignalR. Вы также узнаете, как использовать SignalR в браузере и в любом другом клиентском приложении.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из перечисленных компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Общие сведения о библиотеке SignalR

Поскольку SignalR — это библиотека, встроенная в ASP.NET Core, вам не нужно устанавливать какие-либо пакеты NuGet, чтобы добавить ее. Вы можете получить доступ к ней, добавив несколько строк кода в точку входа вашей программы. Функциональность библиотеки также будет доступна, если вы используете любой шаблон проекта ASP.NET Core.

SignalR использует интуитивно понятный язык для организации общения клиента и сервера, но внутри реализует ряд стандартных механизмов взаимодействия. Под-

держиваются три механизма: WebSocket, Server-Sent Events (SSE) и длительный опрос. Код клиента и сервера останется прежним независимо от того, какой из механизмов выбран. Этот код будет проще по сравнению с непосредственным программированием этих механизмов.

Эти механизмы можно настроить как на клиенте, так и на сервере. Но по умолчанию порядок выбора следующий:

- если возможно, используйте WebSocket;
- если невозможен WebSocket, применяйте SSE;
- если ни WebSocket, ни SSE недоступны, используйте длительный опрос.

В большинстве случаев в SignalR будет выбран WebSocket, поскольку для нее это механизм, принятый по умолчанию. Но существуют сценарии, в которых WebSocket может оказаться невозможным из-за конфигурации сети. То же самое относится и к SSE. Длительный опрос — это не что иное, как стандартный HTTP-запрос. Следовательно, все, что поддерживает HTTP, будет поддерживать длительный опрос.

Теперь давайте рассмотрим, что представляет собой каждый из этих транспортных механизмов.

WebSocket

WebSocket — это протокол, работающий вместе с HTTP. Но, в отличие от чистого HTTP, он устанавливает постоянное дуплексное соединение между клиентом и сервером. После установки соединения сообщения могут передаваться в обе стороны.

Этот протокол очень эффективен для двусторонней связи. "Рукопожатие" клиент-сервер выполняется только один раз при установке соединения. После этого по сети проходит лишь отправляемое время от времени диагностическое сообщение для проверки состояния подключения размером всего в пару байтов.

Протокол WebSocket достаточно эффективен, но ему также свойственны некоторые потенциальные проблемы. Например, существует ограничение на число одновременных соединений. Кроме того, не все типы клиентов поддерживают его. Вот почему это не единственный протокол, поддерживаемый SignalR.

Server-Sent Events

Технология Server-Sent Events (события, посылаемые сервером) позволяет клиенту стать подписчиком сервера и получать от него уведомления о событиях. Сначала клиент отправляет серверу обычный HTTP-запрос с типом контента `Content-Type`, равным `text/event-stream`. Этот запрос подразумевает не просто одиночный ответ. Канал ответа остается открытым, поэтому клиент может продолжать получать сообщения от сервера.

Основная причина, по которой технология SSE не так эффективна, как WebSocket, заключается в том, что она работает только в одном направлении. Если клиенту

необходимо отправить сообщение серверу, ему потребуется повторно отправить HTTP-запрос. При этом ему придется пройти через полную процедуру "рукопожатия", что увеличивает накладные расходы.

Однако не все типы клиентов способны оформить подписку на основе SSE. Поэтому в SignalR предусмотрен еще один транспортный механизм.

Длительный опрос

Длительный опрос — наименее эффективный транспортный механизм из тех, что доступны в SignalR. Но он будет работать с любыми клиентами, поддерживающими HTTP, поскольку представляет собой не что иное, как стандартный HTTP-запрос, ожидающий ответа сервера.

Его малая эффективность обусловлена именно тем, что для каждого отправляемого сообщения клиенту приходится посылать новый полный HTTP-запрос. Кроме того, чтобы прослушать очередное сообщение от сервера, требуется постоянно отправлять новые запросы.

На этом общий обзор SignalR завершен. Теперь мы создадим приложение ASP.NET Core и настроим в нем серверный хаб SignalR.

Создание хаба SignalR на сервере

В нашем примере мы будем использовать приложение Razor Pages ASP.NET Core. Однако SignalR может работать с приложениями ASP.NET Core любого другого типа.

Создадим проект приложения с именем `SignalRServer`. Для этого из CLI можно выполнить следующую команду:

```
dotnet new webapp -o SignalRServer
```

Создав проект, перейдем к созданию хаба SignalR. Хаб SignalR — это класс, который наследует от класса `Hub` пространства имен `Microsoft.AspNetCore.SignalR`. Здесь определяются конечные точки, к которым могут обращаться клиенты.

Для создания собственной реализации `Hub` создадим папку Hubs в папке нашего проекта и добавим в нее файл MessageHub.cs со следующим содержимым:

```
using Microsoft.AspNetCore.SignalR;
using System.Runtime.CompilerServices;
namespace SignalRServer.Hubs;
public class MessageHub : Hub
{
}
```

Этот код формирует базовую структуру нашего хаба SignalR. Теперь заполним ее различными методами конечных точек, чтобы продемонстрировать возможности хаба. Каждый метод конечной точки должен быть публичным. Он также может

принимать произвольное число параметров или любые сериализуемые в JSON типы, например примитивные типы данных C# и классы данных.

Первый метод конечной точки будет называться `BroadcastMessage`. Он принимает один строковый параметр. При вызове клиентом он транслирует это сообщение всем остальным подключенным клиентам, вызывая метод `SendAsync` для всех свойств свойства `Clients` базового класса `Hub`:

```
public async Task BroadcastMessage(string message)
{
    await Clients.All.SendAsync("ReceiveMessage", message);
}
```

В результате на подключенных клиентах будет инициировано событие `ReceiveMessage`, которое является первым параметром вызова `SendAsync`. Остальные параметры поступают в это событие на стороне клиента.

Затем мы можем добавить конечную точку, которая будет перенаправлять сообщение подключенным клиентам, отличным от исходного клиента, вызвавшего конечную точку. Это делается с помощью свойства `Others` свойства `Clients`:

```
public async Task SendToOthers(string message)
{
    await Clients.Others.SendAsync("ReceiveMessage", message);
}
```

Если вы хотите использовать хаб SignalR по принципу "запрос-ответ", чтобы он отправлял сообщение только себе, это можно сделать с помощью свойства `Caller`:

```
public async Task SendToSelf(string message)
{
    await Clients.Caller.SendAsync("ReceiveMessage", message);
}
```

Можно также отправить сообщение конкретному подключенному клиенту. Но для этого необходимо знать уникальный идентификатор соединения, который представляет собой GUID, автоматически генерируемый при подключении клиента. Для получения идентификатора соединения текущего клиента можно обратиться к свойству `Context.ConnectionId` базового класса `Hub`. Вот как его можно использовать для отправки сообщения конкретному клиенту:

```
public async Task SendToSpecificClient(string message, string clientId)
{
    await Clients.Client(clientId).SendAsync("ReceiveMessage", message);
}
```

Также можно отправить сообщение нескольким клиентам. Для этого можно вызвать метод `Clients.Clients`, а в качестве параметра передавать набор идентификаторов клиентов. Но более простой способ — добавить клиентов в группу и просто отправить сообщение группе. Следующий метод демонстрирует, как это можно сделать:

```
public async Task SendToGroup(string message, string groupName)
{
    await Clients.Group(groupName).SendAsync("ReceiveMessage", message);
}
```

Если клиент, вызвавший конечную точку, находится в этой группе, то мы можем с помощью метода `OthersInGroup` отправить сообщения всем остальным членам группы, за исключением вызывающего клиента. Следующий метод демонстрирует, как это можно сделать:

```
public async Task SendToOthersInGroup(string message, string groupName)
{
    await Clients.OthersInGroup(groupName).SendAsync("ReceiveMessage", message);
}
```

До сих пор мы рассматривали только одиночные сообщения. Но SignalR также поддерживает потоковую передачу, как от клиента к серверу, так и в обратном направлении. Следующий метод, например, демонстрирует чтение из потока клиента. Нам просто нужно передать набор сериализуемых в JSON объектов, реализующий интерфейс `IAsyncEnumerable`. Этот код обеспечит чтение из потока до тех пор, пока поток не будет закрыт клиентом:

```
public async Task BroadcastStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        await Clients.Caller.SendAsync($"Server received {item}");
    }
}
```

При потоковой передаче со стороны сервера клиенту все равно необходимо инициировать поток путем отправки параметризованного вызова соответствующей конечной точке. После этого можно просто продолжать добавлять объекты в поток, как демонстрирует следующий метод. Объекты будут считываться из потока клиентом, вызвавшим конечную точку:

```
public async IAsyncEnumerable<string> TriggerStream(
    int jobsToProcess,
     [EnumeratorCancellation]
        CancellationToken cancellationToken)
{
    for (var i = 0; i < jobsToProcess; i++)
    {
        cancellationToken.ThrowIfCancellationRequested();
        yield return $"Job {i} processed successfully.";
        await Task.Delay(1000, cancellationToken);
    }
}
```

Хаб SignalR также имеет некоторые переопределяемые методы, которые запускаются событиями. Например, метод `OnConnectedAsync` запускается при подключении

клиента. В следующем примере переопределения показано, как можно добавить клиента в группу `ConnectedClients` сразу после подключения:

```
public override async Task OnConnectedAsync()
{
    await Groups.AddToGroupAsync(Context.ConnectionId, "ConnectedClients");
    await base.OnConnectedAsync();
}
```

Аналогичным образом можно переопределить метод `OnDisconnectedAsync`, который срабатывает при отключении клиента. В следующем примере мы удаляем клиента из группы `ConnectedClients`, когда это происходит:

```
public override async Task OnDisconnectedAsync(Exception? exception)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, "ConnectedClients");
    await base.OnDisconnectedAsync(exception);
}
```

Наша первоначальная реализация хаба SignalR завершена. Теперь нам нужно его зарегистрировать и присвоить ему путь (URL). Для этого мы сначала зарегистрируем все зависимости SignalR, поместив приведенную далее строку кода в файл Program.cs до назначения переменной `app`:

```
builder.Services.AddSignalR();
```

Теперь необходимо зарегистрировать для класса хаба определенный путь, вызвав метод `MapHub` с переменной `app`. Прежде чем это сделать, убедитесь, что вы добавили пространство имен вашего класса `hub` с помощью оператора `using` в начале файла. В следующем примере мы сопоставляем наш класс `MessageHub` с путем `/messageHub`:

```
app.MapHub<MessageHub>("/messageHub");
```

Теперь наш хаб готов. Но мы можем его усовершенствовать. Как вы могли заметить, большинство наших методов конечных точек посылают сообщения в событие `ReceiveMessage` на подключенных клиентах. Для указания имени этого события мы используем строковый литерал, но это не обязательно. Мы можем сделать наш хаб строго типизированным, тогда клиентские события будут доступны как методы.

Строго типизированный хаб

Для реализации строго типизированного хаба SignalR необходимо добавить интерфейс, отражающий форму наших клиентов. В нашем примере мы добавим в папку Hubs файл IMessageHubClient.cs и наполним его следующим содержимым:

```
namespace SignalRServer.Hubs;
public interface IMessageHubClient
{
    Task ReceiveMessage(string message);
}
```

Далее, если мы хотим превратить класс `MessageHub` в строго типизированный хаб SignalR, нам необходимо изменить его сигнатуру на следующую:

```
MessageHub : Hub<IMessageHubClient>
```

Теперь все строки кода, в которых вызывается метод `SendAsync`, будут выдавать ошибку. Вам просто нужно будет заменить их вызовом `ReceiveMessage` и удалить имя события из его параметров, тогда ваши вызовы будут похожи на следующие:

```
await Clients.All.ReceiveMessage(message);
```

Далее мы рассмотрим, как внедрить в хаб SignalR зависимости.

Внедрение зависимостей в хаб SignalR

Как и в контроллеры MVC и Web API, в хабы SignaLR можно внедрять зависимости. Их можно внедрять как в конструктор, так и в методы отдельных конечных точек.

Например, предположим, что у нас есть класс с именем `SomeService`. Мы можем зарегистрировать ограниченный областью видимости экземпляр внутри файла Program.cs, как показано в этой строке:

```
builder.Services.AddScoped<SomeService>();
```

Затем мы можем определить приватное поле внутри класса `MessageHub`:

```
private readonly SomeService _someService;
```

Кроме того, мы можем добавить в этот класс следующий конструктор, который будет присваивать значение этого поля экземпляру, который он получает от внедрения зависимостей:

```
public MessageHub(SomeService someService)
{
_someService = someService;
}
```

Но если все, что вам нужно, — это использовать данный сервис внутри одного метода, то внедрять его в конструктор и задавать переменную, действующую в рамках класса, возможно, было бы излишним. Поэтому можно также внедрить этот сервис в методы отдельных конечных точек, сделав его последним параметром метода:

```
public async Task BroadcastMessage(string message, SomeService someService)
```

Вы также можете сделать это более явно, добавив атрибут `[FromService]`:

```
public async Task BroadcastMessage(string message,
    [FromService] SomeService someService)
```

Если вы хотите иметь возможность использовать только явную версию, то можете задать для параметра `DisableImplicitFromServicesParameter` значение "ложь" при регистрации зависимостей SignalR, как показано в следующем примере:

```
services.AddSignalR(options =>
{
    options.DisableImplicitFromServicesParameters = true;
});
```

Далее мы рассмотрим форматы сериализации сообщений, которые поддерживает SignalR.

JSON и MessagePack

По умолчанию SignalR применяет JSON для сериализации сообщений при их передаче между клиентом и сервером. Также поддерживается протокол MessagePack, который похож на JSON, но использует двоичную форму. Это означает, что при сериализации сообщения теряют удобный для восприятия человеком вид, однако при этом они имеют значительно меньший размер.

Для включения протокола MessagePack на сервере SignalR достаточно загрузить дополнительный пакет NuGet и добавить некоторые настройки. Потребуется следующий пакет NuGet:

```
Microsoft.AspNetCore.SignalR.Protocols.MessagePack
```

Чтобы включить поддержку MessagePack, необходимо вызвать метод `AddMessagePackProtocol` при регистрации зависимостей SignalR в файле Program.cs:

```
services.AddSignalR().AddMessagePackProtocol();
```

На этом завершим обзор настройки сервера SignalR и рассмотрим клиенты SignalR. Начнем с внутрибраузерного JavaScript-клиента.

JavaScript-клиент для SignalR

Чтобы JavaScript-клиент для SignalR заработал, необходимо установить библиотеку для него. Существуют следующие два основных способа ее установки:

1. Получение из сети доставки контента (CDN).
2. Сборка библиотеки из исходного кода.

Наиболее простой способ — использование CDN. В Интернете уже имеется файл JavaScript. Все, что нужно сделать, — это указать путь к нему. Откроем файл _Layout.cshtml, расположенный в подпапке Shared папки Pages. Вам необходимо найти HTML-элемент `script`, который ссылается на файл site.js, и вставить перед ним следующий элемент:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/
                                        microsoft-signalr/6.0.1/signalr.min.js"></script>
```

Ссылку на CDN очень легко настроить, но при этом вы полагаетесь на скрипт, размещенный на внешнем ресурсе. Поэтому также имеется возможность собрать библиотеку из исходного кода и затем просто скопировать ее непосредственно в папку проекта. Вот шаги, которые следует выполнить для создания библиотеки с помощью Node.js Package Manager (NPM):

1. Установите файл Node.js на своем компьютере.
2. Выполните команду `npm init -y` в любой папке.
3. Выполните следующую команду для установки последней версии библиотеки SignalR: `npm install @microsoft/signalr`.
4. Теперь внутри папки, из которой было инициировано создание проекта NPM, должна появиться папка node_modules, внутри которой можно перейти по следующему пути: `@microsoft/signalr/dist/browser`.

Эта папка содержит файлы signalr.js и signalr.min.js. Вы можете скопировать любой из этих файлов в соответствующее место внутри папки wwwroot своего проекта ASP.NET Core. Эти файлы эквивалентны. Единственное различие между ними заключается в том, что signalr.js содержит все скрипты в удобном для человека формате, тогда как signalr.min.js сильно уменьшен: удалены все лишние символы (например, пробелы) и сокращены имена переменных. Этот скрипт уже непонятен для человека, но занимает значительно меньше места.

Затем нужно просто вставить ссылку на этот файл в файл _Layout.cshtml в то место, куда бы вы вставили элемент скрипта со ссылкой на CDN (при условии, что вы поместили файл signalr.js в папку signalr внутри папки lib):

```
<script src="~/lib/signalr/signalr.js"></script>
```

Наша библиотека готова. Теперь нам необходимо создать клиент, добавив соответствующую HTML-разметку и JavaScript.

Добавление HTML-разметки для клиента SignalR

Чтобы добавить разметку для клиента SignalR, откроем файл Index.cshtml из папки Pages и заменим HTML-разметку внутри файла на следующую:

```
<div class="row" style="padding-top: 50px;">
<div class="col-md-4">
    </div>
    <div class="col-md-7">
        <p>SignalR Messages:</p>
        <pre id="signalr-message-panel"></pre>
    </div>
</div>
```

Эта разметка предоставляет область, в которую будут записываться сообщения, получаемые от хаба SignalR. В ней также имеется пустая секция, которую мы заполним различными текстовыми полями и кнопками, позволяющими отправлять сообщения SignalR. В нашем примере данная секция представлена элементом `div` с классом `col-md-4`. В эту секцию мы вставим следующий элемент, позволяющий транслировать сообщение всем подключенным клиентам:

```
<div class="control-group">
<div>
    <label for="broadcast">Message</label>
```

```
    <input type="text" id="broadcast" name="broadcast" />
</div>
    <button id="btn-broadcast">Broadcast</button>
</div>
```

Далее мы вставим следующий элемент, который позволит нам отправить сообщение всем клиентам, кроме создателя сообщения:

```
<div class="control-group">
    <div>
        <label for="others">Message</label>
        <input type="text" id="others" name="others" />
    </div>
    <button id="btn-others">Send to Others</button>
</div>
```

Затем мы добавим HTML-код, который даст нам возможность отправить сообщение самому себе:

```
<div class="control-group">
    <div>
        <label for="self">Message</label>
        <input type="text" id="self" name="self" />
    </div>
    <button id="btn-self">Send to Self</button>
</div>
```

Следующий элемент управления позволит нам отправить сообщение конкретному клиенту SignalR, если нам известен его уникальный идентификатор соединения:

```
<div class="control-group">
    <div>
        <label for="individual">Message</label>
        <input type="text" id="individual" name="individual" />
    </div>
    <div>
        <label for="connection-id">User connection id:</label>
        <input type="text" id="connection-id" name="connection-id" />
    </div>
    <button id="btn-individual">Send to Specific User</button>
</div>
```

Далее мы можем добавить еще один элемент управления для отправки сообщения определенной группе:

```
<div class="control-group">
    <div>
        <label for="group">Message</label>
        <input type="text" id="group" name="group" />
    </div>
```

```
    <div>
        <label for="group1">Group Name</label>
        <input type="text" id="group1" name="group1" />
    </div>
    button id="btn-group">Send to Group</button>
</div>
```

Следующая разметка позволит нам отправить сообщение всем клиентам в данной группе, исключив отправителя, если он окажется членом этой группы:

```
<div class="control-group">
    <div>
        <label for="others-in-group">Message</label>
        <input type="text" id="others-in-group" name="others-in-group" />
    </div>
    <div>
        <label for="group1">Group Name</label>
        <input type="text" id="group2" name="group2" />
    </div>
    <button id="btn-others-in-group">Send to Others in Group</button>
</div>
```

Затем мы добавим элемент управления, который обеспечит нам возможность отправить поток сообщений на сервер SignalR:

```
<div class="control-group">
    <div>
        <label for="broadcast">Messages</label>
        <input type="text" id="broadcast-stream" name="broadcast-stream" />
    </div>
    <button id="btn-broadcast-stream">Broadcast Stream</button>
</div>
```

Наконец, для запуска потока с сервера вставим следующий элемент управления:

```
<div class="control-group">
    <div>
        <label for="number-of-jobs">Number of Jobs</label>
        <input type="text" id="number-of-jobs" name="number-of-jobs" />
    </div>
    <button id="btn-trigger-stream">Trigger Server Stream</button>
</div>
```

Наша HTML-разметка завершена. Теперь нам остается применить к ней некоторые стили. Для этого мы вставим приведенный далее код CSS в файл site.css, расположенный в папке css внутри папки wwwroot:

```
.body-content {
     padding-left: 15px;
     padding-right: 15px;
}
```

```
.control-group {
    padding-top: 50px;
}
label {
    width: 100px;
}
#signalr-message-panel  {
    height: calc(100vh - 200px);
}
```

Теперь приступим к добавлению JavaScript для реализации JavaScript-клиента.

Применение функций SignalR в JavaScript

Для применения функций JavaScript нам потребуется открыть файл site.js, который расположен в папке js в wwwroot. Начнем с того, что добавим в него следующий скрипт:

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/messageHub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
async function start() {
    try {
        await connection.start();
        console.log('connected');
    } catch (err) {
        console.log(err);
        setTimeout(() => start(), 5000);
    }
};
connection.onclose(async () => {
    await start();
});
```

Здесь мы выстраиваем соединение SignalR. Мы устанавливаем соединение, задав путь `/messageHub` нашего приложения. Это путь, который мы ранее использовали для сопоставления нашего хаба SignalR с сервером. Нам не нужно указывать базовый URL-адрес, поскольку мы запускаем клиент из того же веб-приложения, в котором размещен хаб. Если бы мы запускали его из отдельного JavaScript-приложения, то пришлось бы указывать полный URL-адрес.

Кроме того, имеется функция запуска, которая запускает соединение и автоматически восстанавливает его при обнаружении разрыва. Эту функцию мы также вызываем внутри события `onclose`, чтобы мы могли перезапустить соединение в случае ошибки подключения и возникновения этого события.

Далее мы добавим к нашему соединению событие, которое может быть инициировано сервером SignalR. Если вы помните, при настройке хаба на стороне сервера

мы ожидали, что подключенные клиенты будут получать событие `ReceiveMessage`. Вот как это событие будет выглядеть в нашем JavaScript-клиенте:

```
connection.on("ReceiveMessage", (message) => {
    $('#signalr-message-panel').prepend($('<div />').text(message));
});
```

Когда хаб отправляет параметр сообщения этому событию, этот параметр добавляется к началу HTML-элемента с идентификатором `signalr-message-panel`.

Далее мы добавим несколько событий `onclick` к HTML-кнопкам, которые мы создали ранее. Каждое из них будет вызывать конечную точку в хабе SignalR. Обработчик событий для трансляции сообщения будет выглядеть следующим образом:

```
$('#btn-broadcast').click(function () {
   var message = $('#broadcast').val();
   connection.invoke("BroadcastMessage", message).catch(err => console.error(err.toString()));
});
```

Таким образом, в клиенте JavaScript запуск конечной точки хаба SignalR выполняется с помощью метода `invoke`. Первый параметр этого метода — это имя метода хаба SignalR, который вы хотите вызвать. Остальные являются входными параметрами для этого метода. Это очень похоже на то, как события на стороне клиента вызываются из хаба на стороне сервера, только в обратном направлении.

Теперь мы можем добавить обработчики событий ко всем остальным кнопкам, которые отправляют отдельные сообщения в хаб:

```
$('#btn-others').click(function () {
    var message = $('#others').val();
    connection.invoke("SendToOthers", message).catch(err => console. error(err.toString()));
});
$('#btn-self').click(function () {
    var message = $('#self').val();
    connection.invoke("SendToSelf", message).catch(err => console. error(err.toString()));
});
$('#btn-individual').click(function () {
    var message = $('#individual').val();
    var connectionId = $('#connection-id').val();
    connection.invoke("SendToSpecificClient", message, connectionId).
catch(err => console.error(err.toString()));
});
$('#btn-group').click(function () {
    var message = $('#group').val();
    var group = $('#group1').val();
    connection.invoke("SendToGroup", message, group).catch(err =>
                                                    console.error(err.toString()));
});
$('#btn-others-in-group-group').click(function () {
    var message = $('#others-in-group').val();
```

```
    var group = $('#group2').val();
    connection.invoke("SendToOthersInGroup", message, group).catch(err =>
                                                        console.error(err.toString()));
});
```

Далее мы реализуем некоторую функциональность для работы с потоками сообщений. Сначала добавим следующий код, который будет отправлять поток сообщений на сервер:

```
$('#btn-broadcast-stream').click(function () {
    var message = $('#broadcast-stream').val();
    var messages = message.split(';');
    var subject = new signalR.Subject();
    connection.send("BroadcastStream", subject).catch(err => console.error(err.toString()));
    for (var i = 0; i < messages.length; i++) {
        subject.next(messages[i]);
    }
    subject.complete();
});
```

Этот код разделит строку на несколько сообщений с помощью точки с запятой. Затем он создаст субъект SignalR, который будет представлять наш поток на стороне клиента. После этого, вызвав метод `send` на объекте соединения, мы получаем определенную конечную точку в хабе сервера SignalR для прослушивания потока от клиента. Далее мы помещаем отдельные сообщения в поток, вызывая метод `next` на субъекте. И наконец, после завершения отправки сообщений мы вызываем метод `complete` на субъекте, чтобы закрыть поток.

Далее мы вставим следующий код для запуска потока с сервера:

```
$('#btn-trigger-stream').click(function () {
    var numberOfJobs = parseInt($('#number-of-jobs').val(), 10);
    connection.stream("TriggerStream", numberOfJobs)
        .subscribe({
           next: (message) => $('#signalr-message-panel').prepend($('<div />').text(message))
        });
});
```

В этом коде мы запускаем на хабе SignalR конечную точку серверного потока, вызывая метод `stream` на объекте соединения. Затем мы вызываем метод `subscribe` для подписки на поток.

После настройки соединения SignalR и добавления всех необходимых обработчиков событий мы можем запустить наше соединение. В нашем случае мы можем просто вызвать функцию `start`, созданную нами ранее:

```
start();
```

На этом завершим обзор JavaScript-клиента SignalR и далее создадим клиент на основе .NET.

.NET-клиент для SignalR

Для демонстрации функциональности .NET-клиента SignalR создадим консольное приложение .NET. Но те же принципы могут быть применены и к любому другому типу приложений, даже к Blazor WebAssembly.

Сначала создадим проект консольного приложения, для чего в CLI можно выполнить следующую команду:

```
dotnet new console -o SignalRClient
```

Далее нам необходимо добавить в проект пакет NuGet для добавления функциональности клиента SignalR. Для этого нам необходимо установить следующий пакет:

```
Microsoft.AspNetCore.SignalR.Client
```

Это будет один и тот же пакет, независимо от типа .NET-приложения, для которого требуется добавить функциональность клиента SignalR.

Затем добавим в файл Program.cs следующий код:

```
using Microsoft.AspNetCore.SignalR.Client;
using System.Threading.Channels;
var url = args[0];
var hubConnection = new HubConnectionBuilder()
                        .WithUrl(url)
                        .Build();
hubConnection.On<string>("ReceiveMessage",
    message => Console.WriteLine($"Message received from the server: {message}"));
await hubConnection.StartAsync();
```

В приведенном коде мы создаем соединение SignalR от URL-адреса, который мы указали в качестве параметра `application`. Это должен быть полный URL-адрес веб-приложения, в котором размещен хаб SignalR, включая путь, сопоставленный с хабом.

Далее мы вставим следующий код, который позволит нам взаимодействовать с консольным приложением и инициировать различные действия на сервере SignalR:

```
var running = true;
while (running)
{
    var message = string.Empty;
    var groupName = string.Empty;
    Console.WriteLine("Please specify the action:");
    Console.WriteLine("0 - Broadcast a message to all");
    Console.WriteLine("1 - Send a message to others");
    Console.WriteLine("2 - Send a message to self");
    Console.WriteLine("3 - Send a message to a specific client");
    Console.WriteLine("4 - Send a to a group");
```

```
    Console.WriteLine("5 - Send a message to others in the group");
    Console.WriteLine("6 - Broadcast messages from client");
    Console.WriteLine("7 - Trigger a stream from the server");
    Console.WriteLine("exit - Terminate the program");
    var action = Console.ReadLine();
    if (action != "7")
    {
        Console.WriteLine("Please specify the message:");
        message = Console.ReadLine();
    }
    if (action == "4")
    {
        Console.WriteLine("Please specify the group name:");
        groupName = Console.ReadLine();
    }
    switch (action)
    {
    }
}
```

Теперь нам остается только вставить варианты в блок `switch`. Сначала добавим варианты, которые будут обрабатывать единичные сообщения:

```
case "0":
    await hubConnection.SendAsync("BroadcastMessage", message);
    break;
case "1":
    await hubConnection.SendAsync("SendToOthers", message);
    break;
case "2":
    await hubConnection.SendAsync("SendToCaller", message);
    break;
case "3":
    Console.WriteLine("Please specify the connection id:");
    var connectionId = Console.ReadLine();
    await hubConnection.SendAsync("SendToSpecificClient", message, connectionId);
    break;
case "4":
    await hubConnection.SendAsync("SendToGroup", message, groupName);
    break;
case "5":
    await hubConnection.SendAsync("SendToOthersInGroup", message, groupName);
    break;
```

Принцип тот же и аналогичен тому, что мы использовали в JavaScript-клиенте. Мы вызываем метод `SendAsync` на объекте соединения SignalR. Первый параметр — это имя хаба SignalR на стороне сервера. А остальные параметры — это параметры, которые принимает данный метод.

Затем мы вставим следующий вариант для работы с клиентским потоком:

```
case "6":
    var channel = Channel.CreateBounded<string>(10);
    await hubConnection.SendAsync("BroadcastStream", channel.Reader);
    foreach (var item in message.Split(';'))
    {
        await channel.Writer.WriteAsync(item);
    }
    channel.Writer.Complete();
break;
```

Здесь мы сначала создаем канал. Затем вызываем метод `SendAsync`, и в качестве одного из параметров передаем функцию чтения из канала. Затем мы записываем сообщения в канал с помощью его средства записи. По окончании передачи мы закрываем средство записи канала.

После этого добавим следующий вариант для работы с функцией потоковой передачи данных на сервер:

```
case "7":
    Console.WriteLine("How many jobs to run?");
    var numberOfJobs = int.Parse(Console.ReadLine() ?? "0");
    var cancellationTokenSource = new CancellationTokenSource();
    var stream = hubConnection.StreamAsync<string>(
        "TriggerStream", numberOfJobs, cancellationTokenSource.Token);
    await foreach (var reply in stream)
    {
        Console.WriteLine(reply);
    }
break;
```

Здесь мы создаем поток, вызывая метод `StreamAsync` на объекте `hubConnection`, а затем выполняем чтение из этого потока.

Мы можем завершить блок `switch`, добавив в него следующие варианты:

```
case "exit":
    running = false;
    break;
default:
    Console.WriteLine("Unknown action. Please try again.");
    break;
```

Теперь мы можем протестировать функциональность клиента SignalR, запустив сначала наше приложение ASP.NET Core, в котором размещен хаб SignalR, а затем выполнив следующую команду для запуска только что добавленного консольного приложения, заменив заполнитель базового URL-адреса (`base URL`) на реальный базовый URL-адрес веб-приложения:

```
dotnet run -- {base URL}/messageHub
```

На этом обзор SignalR на .NET 7 завершен. Подведем итоги тому, что мы узнали.

Заключение

В этой главе мы рассмотрели создание интерактивных веб-приложений на платформе ASP.NET Core с помощью библиотеки SignalR. Мы узнали, как обеспечить доступ к ней на стороне сервера в приложении ASP.NET Core и какие зависимости необходимо установить, чтобы получить к ней доступ на клиентах.

Мы также узнали, что SignalR реализует любой из трех механизмов взаимодействия, которые могут быть настроены. Но независимо от выбранного механизма код приложения, использующего SignalR, останется неизменным, поскольку библиотека абстрагируется от конкретных реализаций своих транспортных механизмов.

Мы также узнали, что поддерживаются клиенты SignalR любого типа. Существует несколько JavaScript-библиотек, которые помогут вам построить клиент SignalR внутри веб-страницы или в отдельном приложении Node.js. Но доступны также пакеты .NET NuGet, которые позволяют превратить любое .NET-приложение в клиент SignalR, включая внутрибраузерное приложение Blazor WebAssembly.

В следующей главе мы рассмотрим реализацию технологии gRPC на основе .NET, которая представляет собой фреймворк для высокоэффективного взаимодействия между сервисами по сети.

Что следует помнить

- Библиотека SignalR предназначена для организации постоянного соединения между клиентом и сервером, которое позволяет им в реальном времени обмениваться между собой произвольными сообщениями.
- Серверные компоненты SignalR включены в состав ASP.NET Core, однако клиентские компоненты необходимо устанавливать отдельно.
- Внутри SignalR использует любой из трех механизмов связи: WebSocket, Server-Sent Events и длительный опрос.
- SignalR работает с клиентами любых типов, как в браузере, так и в отдельных приложениях.
- Полезная нагрузка SignalR имеет формат JSON, но может быть также в формате MessagePack, который представляет собой двоичную версию JSON и, как следствие, занимает меньше места.

Контрольные вопросы

1. Какой механизм связи не поддерживается SignalR на ASP. NET Core?

а) Длительный опрос;

б) Server-Sent Events;

в) ForeverFrame;

г) WebSocket.

2. **Каков порядок приоритетов в механизмах связи SignalR?**
   а) WebSocket, длительный опрос, Server-Sent Events;
   б) WebSocket, Server-Sent Events, длительный опрос;
   в) ForeverFrame, WebSocket, длительный опрос;
   г) Server-Sent Events, длительный опрос, WebSocket.
3. **Какие типы клиентов SignalR официально поддерживает Microsoft?**
   а) JavaScript;
   б) Blazor WebAssembly;
   в) Типовой .NET-клиент;
   г) Все вышеперечисленное.
4. **Какой пакет NuGet необходимо установить, чтобы добавить функции клиента SignalR в приложение .NET?**
   а) Microsoft.AspNetCore.SignalR.Client;
   б) Microsoft.AspNet.SignalR.JS;
   в) Microsoft.AspNet.SignalR;
   г) Любое из вышеперечисленного.

Ответы

1. в.
2. б.
3. г.
4. а.

Основные термины

- **SignalR** — библиотека, встроенная в ASP.NET Core и обеспечивающая двустороннее взаимодействие между клиентом и сервером в реальном времени.
- **WebSocket** — интернет-протокол обмена данными, позволяющий клиенту устанавливать постоянное соединение с сервером для обеспечения двусторонней связи между двумя компьютерами.
- **Server-Sent Events** (События, посылаемые сервером) — серверная push-технология, позволяющая рассылать сообщения подписавшимся клиентам.
- **Длительный опрос** — HTTP-запросы, при которых соединение остается открытым потенциально долгое время, пока сервер не ответит.

Глава 9
gRPC в ASP.NET Core

Введение

Существует множество способов взаимодействия сервисов по сети. Есть протокол REST, который позволяет отправлять HTTP-запросы конечной точке Web API, о чем мы рассказывали в *главе 6*. С помощью SOAP можно также определить более строго типизированную схему сообщений. Можно использовать язык GraphQL и добавить некоторые расширенные возможности по составлению запросов.

Все эти способы взаимодействия хороши для тех целей, для которых они были разработаны, но у них есть существенный недостаток. Они работают по протоколу HTTP/1.1. Это означает, что новые возможности HTTP/2 и HTTP/3 для них недоступны. Например, ни один из этих механизмов не может обладать таким же уровнем возможностей или производительностью, как механизм, опирающийся на более современную версию протокола HTTP.

Однако существует технология, использующая преимущества новейших функций HTTP/2 и HTTP/3, абстрагируясь от сложностей их реализации, которая называется gRPC. Эту технологию широко применяют разработчики программного обеспечения на большинстве языков и платформ. С 2019 года она является полноправным резидентом ASP.NET Core, т. е. доступна в качестве официального расширения для ASP.NET Core.

Часть *RPC* названия *gRPC* означает *Remote Procedure Calls — удаленные вызовы процедур*. И это в достаточной степени объясняет, как работает данная технология. Несмотря на то что разные части программного обеспечения (клиент и сервер) обмениваются сообщениями по сети, написанный код выглядит так, как будто вы просто вызываете некоторую процедуру внутри одного приложения. И по мере рассмотрения примеров в этой главе, вы увидите, как это работает.

Структура

В этой главе мы рассмотрим использование gRPC в ASP.NET Core 7 и затронем следующие темы:

- обзор gRPC;
- настройка сервера gRPC;
- настройка клиента gRPC;
- обзор типов данных gRPC.

Цели

К концу этой главы вы узнаете, как получить доступ к фреймворку взаимодействия gRPC и на клиенте, и на сервере. Вы также изучите основы используемого в gRPC протокола сериализации сообщений Protobuf.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из перечисленных компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Общие сведения о технологии gRPC

Изначально технология gRPC была разработана компанией Google и служила в качестве одного из ее внутренних транспортных механизмов. Но в 2016 году она была выпущена в виде фреймворка с открытым исходным кодом. С тех пор она была адаптирована для множества языков программирования и программных платформ, что обеспечило высокий уровень ее стандартизации. Поэтому практически в любом популярном языке программирования имеется своя реализация этой технологии.

По сути gRPC опирается на новейшие возможности протокола HTTP/2. Именно благодаря этому она отличается высокой эффективностью по сравнению с другими механизмами взаимодействия. Например, функция HTTP/2, известная под названием "мультиплексирование", позволяет gRPC поддерживать несколько одновременных потоков обмена данными в рамках одного соединения. Это обусловливает эффективность gRPC с точки зрения использования пропускной способности сети и отсутствия необходимости открывать множество отдельных соединений.

Новые возможности HTTP/2 не всегда удобны для написания кода под них. Но gRPC скрывает от разработчиков всю эту сложность реализации. Поэтому эту технологию часто называют *протоколом-оболочкой*. Основная функциональность

gRPC отвечает за выполнение сложной логики управления соединениями. Но все, с чем сталкиваются разработчики, — это весьма удобные API. С точки зрения разработчика, выполнение вызовов gRPC по сети ничем не отличается от вызова метода внутри приложения. Это облегчается структурой протокола сериализации сообщений gRPC, который известен под названием Protocol buffers или, просто, Protobuf.

Protobuf как основной протокол сериализации сообщений

При создании Protobuf преследовались две цели:

1. Упрощение генерации абстракций в коде.
2. Упрощение генерации сообщений, которые могут быть эффективно переданы по сети.

В результате достижения первой цели gRPC теперь очень легко реализовать в коде. Protobuf опирается на файлы определения сервисов. Эти файлы, имеющие расширение proto, определяют структуру сервисов gRPC, вызываемых процедур, а также сообщений запросов и ответов. Но все это организовано таким образом, что легко переводится в структуры, обычно поддерживаемые языками программирования, такие как классы, объекты и т. д.

Последнее достигается за счет сериализации сообщений в двоичную форму с сохранением только того объема информации, который необходим. Например, здесь типы данных Protobuf имеют значения по умолчанию, и если значение конкретного поля совпадает с его значением по умолчанию, то нет необходимости передавать это поле по сети. Аналогично, несмотря на то, что имена полей используются в Protobuf для удобства чтения, они не передаются по сети. Каждому полю назначается целый порядковый номер, который занимает гораздо меньше места, чем произвольное имя. По этому номеру получатель сообщения может точно определять поля.

Рассмотрим структуру сообщений Protobuf. Для этого создадим базовое приложение gRPC Service, которое будет выступать в роли сервера для входящих сообщений.

Настройка сервера gRPC

.NET SDK уже содержит шаблон проекта ASP.NET Core со всеми необходимыми для сервиса gRPC зависимостями, а также несколько базовых примеров кода. Теперь воспользуемся этим шаблоном для создания проекта и назовем его `GrpcServiceApp`. Если вы используете IDE, то для создания проекта можете взять шаблон `ASP.NET Core gRPC Service`. В качестве альтернативы можно создать проект, выполнив на консоли следующую команду:

```
dotnet new grpc -o GrpcServiceApp
```

После создания проекта рассмотрим его структуру.

Структура проекта ASP.NET Core gRPC

Создаваемый проект напоминает базовое приложение ASP.NET Core. Однако у него есть ряд дополнительных компонентов. Один из них — это папка Protos, которая содержит файлы с расширением proto. Другой — папка Services, содержащая классы, реализующие определения сервисов из файлов *.proto. Структура проекта показана на рис. 9.1.



Рис. 9.1. Структура проекта gRPC Service

Здесь мы имеем возможность более подробно рассмотреть структуру типичного файла *.proto. Если мы откроем файл greet.proto из папки Protos, то увидим, что он начинается со следующего выражения:

```
syntax = "proto3";
```

Это говорит о том, что мы используем версию 3 Protobuf, которая на момент написания книги является самой последней версией протокола. Затем идет следующее выражение:

```
option csharp_namespace = "GrpcServiceApp";
```

Ключевое слово `option` указывает на то, что мы имеем дело с настраиваемой опцией, которая в данном случае носит имя `csharp_namespace`. Эта опция применима только к приложениям на языке C# и переопределяет пространство имен для автоматически генерируемых файлов кода.

Следующее выражение имеет вид:

```
package greet;
```

Это имя пакета gRPC — аналог пространства имен в C#. Если бы у нас не было упомянутой ранее опции `csharp_namespace`, то пространство имен C# представляло бы собой версию имени пакета в стиле Pascal.

Далее приведено следующее определение сервиса:

```
service Greeter {
    rpc SayHello(HelloRequest) returns (HelloReply);
}
```

Это сигнатура сервиса gRPC, который в нашем примере называется `Greeter`. В нем имеется один RPC с именем `SayHello`. В качестве запроса этот RPC использует

сообщение `HelloRequest`, а в качестве ответа — сообщение `HelloReply`. Каждое определение RPC должно содержать как тип запроса, так и тип ответа. Однако для обоих этих типов допускается отсутствие полей.

Сообщение `HelloRequest` имеет следующее определение:

```
message HelloRequest {
    string name = 1;
}
```

В нем указано одно поле типа `string`. Цифра 1 рядом с ним — это его порядковый номер. Каждое поле в определении сообщения должно иметь связанный с ним порядковый номер. Причем каждый порядковый номер должен быть уникальным.

Сообщение `HelloReply` имеет следующее определение:

```
message HelloReply {
    string message = 1;
}
```

Когда мы переведем это определение Protobuf в код, каждое сообщение и сервис будут представлены классом. Класс, представляющий сервис, будет абстрактным, т. к. мы захотим применить к его методам собственную логику, переопределив их. Определение каждого RPC в рамках сервиса будет представлено методом. Мы можем увидеть, как это транслируется в код C#, открыв класс `GreeterService`, который расположен в папке Services. Сигнатура класса будет выглядеть следующим образом, т. к. мы переопределяем класс, который был автоматически сгенерирован из файла proto:

```
public class GreeterService : Greeter.GreeterBase
```

Далее представлена реализация RPC `SayHello`. Дополнительный параметр `ServerCallContext` служит для хранения метаданных, относящихся к конкретному сеансу:

```
public override Task<HelloReply> SayHello(
    HelloRequest request, ServerCallContext context)
```

Чтобы получить доступ к функциям gRPC из нашего приложения ASP.NET Core, необходимо установить следующий пакет NuGet:

```
Grpc.AspNetCore
```

Кроме того, в нашем файле формата CSPROJ должен находиться XML-код Protobuf, указывающий на каждый файл proto, который мы хотим использовать. У нас это определено в файле GrpcServiceApp следующим образом:

```
<ItemGroup>
    <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>
```

Обратите внимание, что для параметра `GrpcServices` задано значение `Server`. Это ограничит генератор кода созданием только серверных gRPC-компонентов для указанного файла *.proto. Мы также можем установить для этого атрибута значение `Client`. Если ему вообще не присваивать значение, то генератор кода будет выдавать и клиентские, и серверные компоненты.

Прежде чем начать использовать gRPC внутри нашего приложения, необходимо сделать еще кое-что. Если мы откроем файл Program.cs, то найдем следующую строку:

```
builder.Services.AddGrpc();
```

Это выражение добавит в наше приложение все необходимые зависимости gRPC. Затем нам также необходимо зарегистрировать каждую реализацию сервиса gRPC, которую мы собираемся использовать. Это делается путем вызова метода `MapGrpcService` с переменной `app`, в качестве типа которой указана реализация сервиса. В нашем коде есть следующий пример:

```
app.MapGrpcService<GreeterService>();
```

На этом завершим обзор базовой настройки сервиса gRPC и рассмотрим типы вызовов и типы данных, поддерживаемые gRPC.

Типы вызовов и типы данных gRPC

До сих пор мы рассматривали только один тип RPC, известный как одиночный вызов. Вызов этого типа принимает один запрос и возвращает один ответ. Однако gRPC также поддерживает следующие типы вызовов:

- потоковый серверный вызов;
- потоковый клиентский вызов;
- потоковый двунаправленный вызов.

Чтобы продемонстрировать их все, мы изменим определение сервиса `Greeter` в файле greet.proto следующим образом:

```
service Greeter {
    rpc SayHello (HelloRequest) returns (HelloReply);
    rpc RequestManyReplies (HelloRequest) returns (stream HelloReply);
    rpc SendManyRequests (stream HelloRequest) returns (HelloReply);
    rpc InitiateBidirectionalStreaming (stream HelloRequest) returns (stream HelloReply);
}
```

Таким образом, для определения любых потоковых вызовов достаточно поместить ключевое слово `stream` либо перед типом входа, либо перед типом выхода, либо перед обоими типами. В данном примере RPC `RequestManyReplies` является потоковым серверным вызовом, RPC `SendManyRequests` — потоковым клиентским вызовом, и RPC `InitiateBidirectionalStreaming` — потоковым двунаправленным вызовом.

Мы также рассмотрим некоторые встроенные типы данных. Для этого изменим определение сообщения `HelloReply`, чтобы оно приобрело следующий вид:

```
message HelloReply {
    string message = 1;
    int32 message_id = 2;
    bytes content_in_bytes = 3;
    float message_size_in_kilobytes = 4;
    bool reply_processed = 5;
}
```

Более подробно встроенные типы данных Protobuf мы рассмотрим позже. А в данном примере задействованы следующие типы:

- `int32` — 32-битовое целое число, транслируемое в C# в стандартный тип `int`;
- `bytes` — действует как массив байтов и представлен типом данных `ByteString` из пространства имен `Google.Protobuf` в C#;
- `float` — прямой эквивалент типа `float` в C#;
- `bool` — прямой эквивалент типа `bool` в C#.

Теперь внесем некоторые изменения в класс `GreeterService`, чтобы увидеть, как работают эти типы данных. Первое, что нам нужно сделать, это добавить в него следующее выражение `using`:

```
using Google.Protobuf;
```

Затем мы добавим следующий приватный метод, который будет использоваться всеми нашими реализациями RPC для генерации ответа:

```
private HelloReply GenerateReplyMessage(HelloRequest request)
{
    var message = "Hello " + request.Name;
    return new HelloReply
    {
        Message = message,
        MessageId = 1,
        ContentInBytes = ByteString.CopyFrom(Encoding.ASCII.GetBytes(message)),
        MessageSizeInKilobytes =
                (float)ByteString.CopyFrom(Encoding.ASCII.GetBytes(message)).Length / 1024,
        ReplyProcessed = false
    };
}
```

Как видим, есть некоторые типы данных, которые мы можем применять напрямую. Также существует тип `ByteString`, который мы можем заполнить из стандартной строки байтов, вызвав метод `CopyFrom`. Тогда наш оригинальный метод `SayHello` будет выглядеть следующим образом:

```
public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
{
    return Task.FromResult(GenerateReplyMessage(request));
}
```

Реализация метода серверного потока `RequestManyReplies` будет выглядеть так:

```
public override async Task RequestManyReplies(
    HelloRequest request,
    IServerStreamWriter<HelloReply> responseStream,
    ServerCallContext context)
{
        var i = 3;
```

```
        while (i > 0)
        {
            await responseStream.WriteAsync(GenerateReplyMessage(request));
            i--;
        }
}
```

Здесь мы помещаем в поток ответа три сообщения. Метод не имеет возвращаемого типа, т. к. поток заполняется до того, как мы возвращаемся из метода. Клиент сможет читать данные из потока по мере его заполнения.

Добавим следующую реализацию для метода `SendManyRequests`:

```
public override async Task<HelloReply> SendManyRequests(
    IAsyncStreamReader<HelloRequest> requestStream,
    ServerCallContext context)
{
        var reply = new HelloReply();
        await foreach (var request in requestStream.ReadAllAsync())
        {
            reply = GenerateReplyMessage(request);
        }
        return reply;
}
```

В этом методе мы читаем из клиентского потока до тех пор, пока клиент не закроет свой поток.

Наконец, метод двунаправленного потока будет выглядеть следующим образом:

```
public override async Task InitiateBidirectionalStreaming(
    IAsyncStreamReader<HelloRequest> requestStream,
    IServerStreamWriter<HelloReply> responseStream,
    ServerCallContext context)
{
        await foreach(var request in requestStream.ReadAllAsync())
        {
            await responseStream.WriteAsync(GenerateReplyMessage(request));
        }
}
```

В вызове этого типа мы не возвращаем ответ. Вместо этого в качестве входных параметров мы имеем два потока: поток клиента и поток сервера. Мы можем читать из потока клиента и записывать в поток сервера.

Вскоре мы рассмотрим настройку gRPC-клиента в .NET. Но перед этим обратим внимание на функцию, которая была добавлена в gRPC в ASP.NET Core 7 и называется JSON-перекодированием.

JSON-перекодирование в gRPC

JSON-перекодирование позволяет разработчикам подключаться к серверам gRPC по стандартному протоколу REST. Для этого выполняется преобразование сообщений Protobuf в объекты JSON.

Существуют ограничения на перекодирование JSON. Во-первых, эта функция по-прежнему работает по стандартному протоколу HTTP/1.1, поэтому преимущества HTTP/2 теряются. Во-вторых, она не поддерживает потоковую передачу данных от клиентов. Однако она все равно полезна, поскольку разработчикам больше не придется писать отдельные реализации gRPC и REST.

Мы даже можем использовать вместе с этой функцией Swagger. Для этого нам необходимо установить следующий пакет NuGet:

```
Microsoft.AspNetCore.Grpc.Swagger
```

Затем следует открыть файл Program.cs и изменить вызов `AddGrpc`, чтобы он выглядел следующим образом:

```
builder.Services.AddGrpc().AddJsonTranscoding();
```

Далее мы можем добавить стандартные зависимости Swagger:

```
builder.Services.AddGrpcSwagger();
builder.Services.AddSwaggerGen(c =>
{
c.SwaggerDoc("v1",
new OpenApiInfo { Title = "REST API to gRPC", Version = "v1" });
});
```

После добавления всех зависимостей мы можем зарегистрировать промежуточное ПО Swagger, добавив следующий код:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
});
```

Теперь, чтобы перекодирование JSON заработало, нам необходимо скопировать некоторые файлы Protobuf в папку нашего проекта. Для начала нам нужно создать папку с именем google в корне нашего проекта. Затем внутри этой папки создадим папку api. После этого добавим в эту папку файл http.proto со следующим содержимым:

```
syntax = "proto3";
package google.api;
option cc_enable_arenas = true;
option go_package = "google.golang.org/genproto/googleapis/api/annotations;annotations";
option java_multiple_files = true;
option java_outer_classname = "HttpProto";
```

```
option java_package = "com.google.api";
option objc_class_prefix = "GAPI";
message Http {
    repeated HttpRule rules = 1;
    bool fully_decode_reserved_expansion = 2;
}
message HttpRule {
    string selector = 1;
    oneof pattern {
        string get = 2;
        string put = 3;
        string post = 4;
        string delete = 5;
        string patch = 6;
        CustomHttpPattern custom = 8;
    }
    string body = 7;
    string response_body = 12;
    repeated HttpRule additional_bindings = 11;
}
message CustomHttpPattern {
    string kind = 1;
    string path = 2;
}
```

Затем в той же папке создадим файл annotations.proto и наполним его следующим содержимым:

```
syntax = "proto3";
package google.api;
import "google/api/http.proto";
import "google/protobuf/descriptor.proto";
option go_package = "google.golang.org/genproto/googleapis/api/annotations;annotations";
option java_multiple_files = true;
option java_outer_classname = "AnnotationsProto";
option java_package = "com.google.api";
option objc_class_prefix = "GAPI";
extend google.protobuf.MethodOptions {
    HttpRule http = 72295728;
}
```

Теперь откроем наш исходный файл greet.proto и добавим в него следующее выражение непосредственно перед определением пакета:

```
import "google/api/annotations.proto";
```

Далее модифицируем наши RPC `SayHello` и `RequestManyReplies`, придав им следующий вид:

```
rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
        get: "/v1/greeter/{name}"
    };
}
rpc RequestManyReplies (HelloRequest) returns (stream HelloReply) {
    option (google.api.http) = {
        get: "/v1/greeter/{name}/stream"
    };
}
```

Теперь эти RPC будут связаны с путями REST API. Мы можем проверить эти пути, запустив наше приложение и перейдя на его Swagger-страницу, которая по умолчанию доступна по адресу /swagger. Как видно на рис. 9.2, теперь нам доступны эти пути.

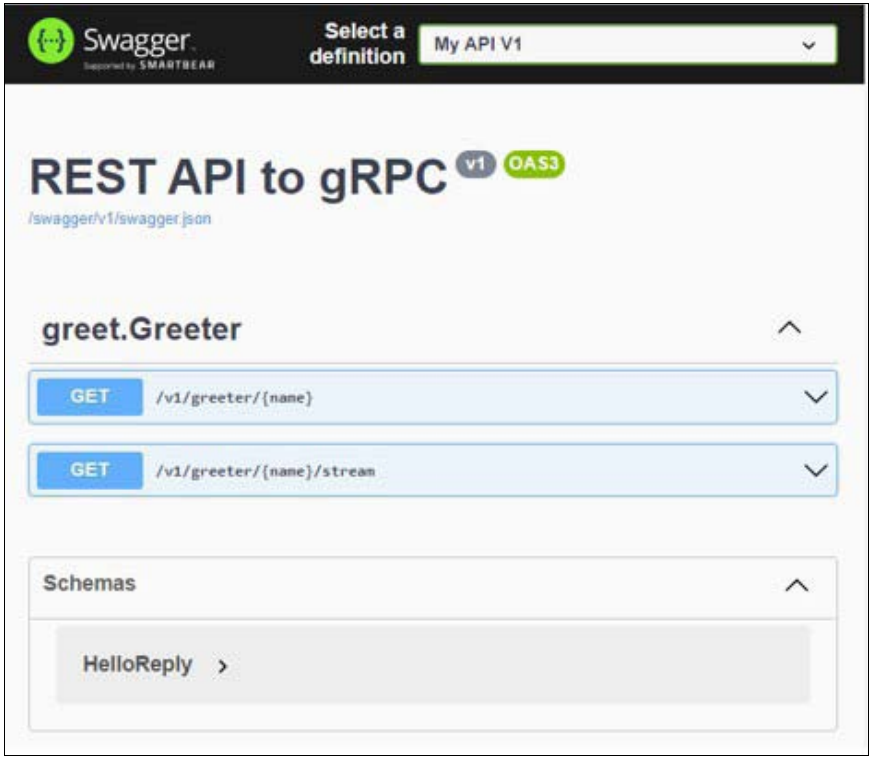


Рис. 9.2. Swagger-страница с конечными точками REST, которые сопоставлены с методами gRPC

Когда мы применяем перекодирование JSON, одиночные вызовы gRPC преобразуются в стандартные конечные точки запроса-ответа, а потоковые серверные вызовы возвращают коллекции JSON.

На этом обзор настройки сервера gRPC завершен. Теперь поговорим о настройке клиента.

Настройка клиента gRPC

Функции клиента gRPC можно добавить к .NET-приложению любого типа, включая самое простое консольное приложение. Поэтому для демонстрации функций клиента мы создадим консольное приложение с именем `BasicGrpcClient`. Это можно сделать через CLI, выполнив следующую команду:

```
dotnet new console -o BasicGrpcClient
```

Чтобы получить доступ к функциям клиента, нам необходимо установить следующие три пакета NuGet:

- Google.Protobuf;
- Grpc.Net.Client;
- Grpc.Tools.

Далее, поскольку мы будем использовать то же определение Protobuf, что и наш сервер, нам нужно скопировать папку Protos из проекта `GrpcServiceApp` в проект `BasicGrpcClient`. Однако на этот раз нам не нужна функция перекодирования JSON. Поэтому, после того как мы удалим из него все лишние ссылки, наш файл Greeting.proto в проекте `BasicGrpcClient` должен выглядеть следующим образом:

```
syntax = "proto3";
option csharp_namespace = "GrpcServiceApp";
package greet;
service Greeter {
    rpc SayHello (HelloRequest) returns (HelloReply);
    rpc RequestManyReplies (HelloRequest) returns (stream HelloReply);
    rpc SendManyRequests (stream HelloRequest) returns (HelloReply);
    rpc InitiateBidirectionalStreaming (stream HelloRequest) returns
(stream HelloReply);
}
message HelloRequest {
    string name = 1;
}
message HelloReply {
    string message = 1;
    int32 message_id = 2;
    bytes content_in_bytes = 3;
    float message_size_in_kilobytes = 4;
    bool reply_processed = 5;
}
```

Далее нам нужно будет зарегистрировать этот proto-файл в файле BasicGrpcClient.csproj путем добавления в него следующей разметки:

```
<ItemGroup>
    <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

На этот раз для атрибута `GrpcServices` установлено значение `Client`, поскольку нас интересует только импорт функций клиента.

Далее, заменим содержимое фала Program.cs следующим:

```
using Grpc.Core;
using Grpc.Net.Client;
using GrpcServiceApp;
using var channel = GrpcChannel.ForAddress("https://localhost:7222");
```

Здесь мы добавляем необходимые пространства имен. Затем создаем разовый экземпляр типа `GrpcChannel`. Внутри вызова `ForAddress` мы указываем адрес нашего серверного приложения gRPC, который можно найти в файле launchSetting.json, расположенном в папке Properties. В данном примере это адрес `https://localhost:7222`. Но в вашем случае, возможно, будет по-другому.

Важно отметить, что gRPC по умолчанию применяет безопасный протокол HTTPS. Это связано с тем, что незащищенный URL-адрес HTTP по умолчанию сопоставляется с протоколом HTTP/1.1, который не поддерживает gRPC. Но мы можем внести некоторые изменения в код нашего сервера, чтобы разрешить незащищенный доступ к HTTP/2, что можно сделать, если вы не хотите использовать TLS-сертификат разработки. Если вы работаете с Mac, это необходимо сделать, поскольку macOS не поддерживает сопоставление TLS с HTTP/2 на уровне ОС. Для этого нам нужно будет вставить следующий код в файл Program.cs нашего приложения сервера gRPC, где мы заменяем `{port number}` фактическим номером порта, который мы хотим сопоставить:

```
builder.WebHost.ConfigureKestrel(options =>
{
    options.ListenLocalhost({port number}, o => o.Protocols = HttpProtocols.Http2);
});
```

Продолжим настройку функций нашего клиента. У нас уже открыт канал gRPC. Теперь нам нужно перейти к приглашению консоли и создать экземпляр класса `HelloRequest` на основе ввода с консоли. Затем мы можем создать экземпляр клиента:

```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();
var request = new HelloRequest()
{
    Name = name
};
var client = new Greeter.GreeterClient(channel);
```

Теперь мы можем использовать экземпляр клиента для выполнения одиночного вызова:

```
var reply = await client.SayHelloAsync(request);
Console.WriteLine("Message received from a unary call: " + reply.Message);
```

Обратите внимание, что имя метода на стороне клиента, представляющего собой одиночный gRPC-вызов, имеет суффикс `Async`. Это связано с тем, что мы применяем

асинхронную версию. Но существует и синхронная версия метода, которая не имеет этого суффикса.

Затем мы можем выполнить потоковый серверный вызов. Это делается путем открытия разового потока и чтения из него:

```
using var serverStreamingCall = client.RequestManyReplies(request);
while (await serverStreamingCall.ResponseStream.MoveNext(CancellationToken.None))
{
    Console.WriteLine("Message received from a server streaming call: " +
        serverStreamingCall.ResponseStream.Current.Message);
}
```

Далее мы создадим потоковый клиентский вызов. Здесь мы откроем поток, запишем в него несколько сообщений, закроем поток и будем ждать ответа:

```
using var clientStreamingCall = client.SendManyRequests();
for (var i = 0; i < 3; i++)
{
    await clientStreamingCall.RequestStream.WriteAsync(request);
}
await clientStreamingCall.RequestStream.CompleteAsync();
var response = await clientStreamingCall;
Console.WriteLine("Message received from a client streaming call: " + response.Message);
```

После этого мы добавим двунаправленный потоковый вызов. Здесь мы создадим разовый объект, который будет представлять как клиентский, так и серверный поток. Запись в клиентский поток и чтение из серверного потока осуществляются с помощью асинхронной задачи. Затем мы закрываем клиентский поток и ждем завершения этой задачи:

```
using var bidirectionalCall = client.InitiateBidirectionalStreaming();
var readTask = Task.Run(async () =>
{
    await foreach (var response in bidirectionalCall.ResponseStream.ReadAllAsync())
    {
       Console.WriteLine("Message received from a bidirectional call: " + response.Message);
    }
});
await bidirectionalCall.RequestStream.WriteAsync(request);
await bidirectionalCall.RequestStream.CompleteAsync();
await readTask;
```

Наконец, мы можем добавить следующие строки, чтобы гарантировать, что терминал не закроется после того, как приложение выполнит свою логику:

```
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

На этом обзор основных функций клиента gRPC завершен. Однако в реальной жизни клиентом gRPC часто является другое веб-приложение или хостируемый

веб-сервис. Хорошая новость состоит в том, что ASP.NET Core поддерживает внедрение зависимостей клиентских компонентов gRPC. Этот подход мы рассмотрим далее.

Использование фабрики клиентов gRPC и внедрение зависимостей

Теперь создадим MVC-приложение ASP.NET Core с именем `GrpcMvcClient`. Это можно сделать, выполнив следующую команду из CLI:

```
dotnet new mvc -o GrpcMvcClient
```

Затем можно добавить все функции клиента, включая внедрение зависимостей, установив следующий пакет NuGet:

```
Grpc.AspNetCore.Server.ClientFactory
```

Если вы хотите иметь возможность использовать свое приложение как клиент gRPC и сервер gRPC, то вместо этого можно просто установить пакет Grpc.AspNetCore. Он содержит все необходимые клиентские пакеты, включая предыдущий.

Затем нам нужно скопировать папку Protos из созданной ранее папки проекта BasicGrpcClient в папку GrpcMvcClient. После этого следует зарегистрировать файл прототипа в файле GrpcMvcClient.csproj. Разметка может быть такой же, как и ранее в файле BasicGrpcClient.csproj, или альтернативной:

```
<ItemGroup>
    <None Update="Protos\greet.proto">
        <GrpcServices>Client</GrpcServices>
    </None>
</ItemGroup>
```

Далее мы зарегистрируем наш клиент gRPC для внедрения зависимостей. Для этого мы сначала добавим в начало файла Program.cs следующее выражение:

```
using GrpcServiceApp;
```

Затем добавим следующий код в любое место перед созданием переменной `app`:

```
builder.Services.AddGrpcClient<Greeter.GreeterClient>(o =>
{
    o.Address = new Uri("https://localhost:7222");
});
```

Обратите внимание, что входными данными конструктора класса `Uri` является URL-адрес серверного приложения gRPC. Номер порта в вашем экземпляре, возможно, будет другим.

Теперь поместим файл ResponseModel.cs в папку Models и наполним его следующим содержимым:

```
namespace GrpcMvcClient.Models;
public class ResponseModel
```

```
{
    public string Message { get; set; }
}
```

Затем добавим следующее выражение в файл HomeController.cs, находящийся в папке Controllers:

```
using GrpcServiceApp;
```

Далее добавим в класс `HomeController` следующее приватное поле:

```
private readonly Greeter.GreeterClient _client;
```

После этого заменим конструктор класса следующим:

```
public HomeController(
    ILogger<HomeController> logger,
    Greeter.GreeterClient client)
{
    _logger = logger;
    _client = client;
}
```

Здесь мы внедряем экземпляр клиента gRPC, который мы ранее зарегистрировали в файле Program.cs. Теперь мы можем использовать этот экземпляр клиента. Для этого мы заменим метод `Index` следующим:

```
public async Task<IActionResult> Index()
{
    var reply = await _client.SayHelloAsync(
        new HelloRequest
        {
            Name = "User"
        });
    var response = new ResponseModel
    {
        Message = reply.Message
    };
    return View(response);
}
```

Здесь мы делаем одиночный вызов gRPC. Далее извлекаем сообщение, полученное от сервера gRPC, и помещаем его в модель представления. Чтобы отобразить его значение, нам нужно найти файл Index.cshtml, который находится в подпапке Home папки Views. Заменим его содержимое следующим:

```
@model ResponseModel
@{
    ViewData["Title"] = "Home Page";
}
<div class="text-center">
    <h1 class="display-4">@Model.Message</h1>
</div>
```

Когда мы запустим это приложение и откроем его домашнюю страницу в браузере, то увидим сообщение, которое мы получили от серверного приложения gRPC, если оно также запущено, и мы правильно настроили наше соединение.

На этом завершим обзор настройки сервера gRPC в .NET и рассмотрим типы данных, поддерживаемые gRPC.

Обзор типов данных gRPC

Примитивные типы данных в gRPC, которые также известны как типы скалярных значений, не допускают значения `NULL`. Вместо этого для каждого из них предусмотрено значение по умолчанию. Например, значение по умолчанию для любого числового типа данных равно нулю, а для типа `bool` — `false` (ложь). Это очень похоже на эквивалентные типы данных в C#.

Но, в отличие от C#, Protobuf поддерживает разные типы целых чисел. Это дает дополнительную гибкость и эффективность при передаче сообщений по сети. Все встроенные типы данных, поддерживаемые Protobuf, перечислены в табл. 9.1.

Таблица 9.1. *Типы данных Protobuf*

Тип	Описание	Значение по умолчанию	Эквивалент в C#
`int32`	Целое число, которое занимает до 32 битов, но может занимать меньше, если значение невелико. Поддерживает как положительные, так и отрицательные значения	0	`int`
`int64`	Целое число, которое занимает до 64 битов, но может занимать меньше, если значение невелико. Поддерживает как положительные, так и отрицательные значения	0	`long`
`uint32`	Целое число, которое занимает до 32 битов, но может занимать меньше, если значение невелико. Поддерживает только положительные значения	0	`uint`
`uint64`	Целое число, которое занимает до 64 битов, но может занимать меньше, если значение невелико. Поддерживает только положительные значения	0	`ulong`
`sint32`	Целое число, которое занимает до 32 битов, но может занимать меньше, если значение невелико. Поддерживает как положительные, так и отрицательные значения, но особенно эффективен для кодирования отрицательных значений	0	`int`
`sint64`	Целое число, которое занимает до 64 битов, но может занимать меньше, если значение невелико. Поддерживает как положительные, так и отрицательные значения, но особенно эффективен для кодирования отрицательных значений	0	`int`
`fixed32`	Целое число, которое всегда занимает ровно 32 бита. Этот тип особенно эффективен для кодирования больших чисел. Поддерживает только положительные значения	0	`uint`

Таблица 9.1 *(окончание)*

Тип	Описание	Значение по умолчанию	Эквивалент в C#
fixed64	Целое число, которое всегда занимает ровно 64 бита. Этот тип особенно эффективен для кодирования больших чисел. Поддерживает только положительные значения	0	ulong
sfixed32	Целое число, которое всегда занимает ровно 32 бита. Этот тип особенно эффективен для кодирования больших чисел. Поддерживает как положительные, так и отрицательные значения	0	int
sfixed64	Целое число, которое всегда занимает ровно 64 бита. Этот тип особенно эффективен для кодирования больших чисел. Поддерживает как положительные, так и отрицательные значения	0	long
float	Числовой тип данных, поддерживающий десятичную запятую	0	float
double	Числовой тип данных, поддерживающий десятичную запятую и имеющий двойную точность	0	double
string	Произвольный текст	""	string
bytes	Коллекция байтов	Пустой массив байтов	Google.Protobuf.ByteString
bool	"Истина" или "ложь"	"ложь"	bool

В дополнение к этому Protobuf поддерживает перечисления enum, поля repeated, поля map и ключевое слово oneof. Далее мы кратко рассмотрим их.

Перечисления Protobuf

Перечисления в Protobuf аналогичны перечислениям в C#. Даже синтаксис схож. Но есть несколько моментов, которые следует помнить. Например, поскольку каждый встроенный тип данных в Protobuf имеет значение по умолчанию, перечисления тоже имеют такое значение. Для любого типа перечисления (enum) это значение равно нулю. Следовательно, в каждом типе перечисления (enum) должно быть предусмотрено значение, соответствующее нулю. Кроме того, в Protobuf каждому ключу перечисления должно быть сопоставлено числовое значение.

Вот как может выглядеть определение enum:

```
enum DemoEnum {
    VALUE_ONE = 0;
    VALUE_TWO = 1;
    VALUE_THREE = 2;
}
```

Далее мы рассмотрим ключевое слово repeated, которое используется для представления коллекций Protobuf.

Использование коллекций с помощью ключевого слова *repeated*

Если требуется, чтобы какое-либо из полей сообщения имело вид коллекции, то для этого перед определением поля достаточно поместить ключевое слово `repeated`. Таким образом, вместо одного значения определенного типа данных у нас будет коллекция значений этого типа данных.

В Protobuf также поддерживается функциональность, подобная словарю. Для этого предусмотрено ключевое слово `map`, которое мы рассмотрим далее.

Функциональность словаря в Protobuf

В Protobuf ключевое слово `map` позволяет реализовать функциональность, подобную словарю. При применении это поле становится хранилищем пар "ключ-значение", где каждый ключ должен быть уникальным. Далее приведен пример использования этого ключевого слова. В этом примере у нас есть 32-битовое целое число в качестве ключа и строка в качестве значения:

```
map<int32, string> map_example = 1;
```

Далее мы рассмотрим ключевое слово `oneof`, которое не имеет эквивалента в C#.

Использование ключевого слова *oneof* в Protobuf

При использовании в сообщении Protobuf ключевого слова `oneof` у нас может быть коллекция полей, в которой одновременно может быть заполнено только одно поле. Когда мы задаем значение для любого из полей, любое поле, которое было установлено ранее, автоматически сбрасывается.

Пример ключевого слова `oneof` имеется в файле http.proto, расположенном в папке google/api в проекте `GrpcServiceApp`. У нас есть следующий блок в определении сообщения `HttpRule`:

```
oneof pattern {
    string get = 2;
    string put = 3;
    string post = 4;
    string delete = 5;
    string patch = 6;
    CustomHttpPattern custom = 8;
}
```

На этом обзор встроенных типов данных и ключевых слов, которые поддерживает Protobuf, завершен. Важно, что мы не зациклены на этих типах. Мы можем создавать собственные типы сообщений и добавлять любые необходимые нам функции. К счастью, некоторые из этих типов уже предоставлены нам в базовой библиотеке Google. Их относят к так называемым *общеизвестным типам*.

Общеизвестные типы данных

Чтобы продемонстрировать, как работают общеизвестные типы, добавим файл nullable_greet.proto в папку Protos проекта GrpcServiceApp. Затем наполним этот файл следующим содержимым:

```
syntax = "proto3";
option csharp_namespace = "GrpcServiceApp";
import "google/protobuf/wrappers.proto";
package greet;
service NullableGreeter {
    rpc SayHello (NullableHelloRequest) returns (NullableHelloReply);
    rpc RequestManyReplies (NullableHelloRequest) returns (stream NullableHelloReply);
    rpc SendManyRequests (stream NullableHelloRequest) returns (NullableHelloReply);
    rpc InitiateBidirectionalStreaming (stream NullableHelloRequest) returns (stream
                                                                    NullableHelloReply);
}
message NullableHelloRequest {
    google.protobuf.StringValue name = 1;
}
message NullableHelloReply {
    google.protobuf.StringValue message = 1;
    google.protobuf.Int32Value message_id = 2;
    google.protobuf.BytesValue content_in_bytes = 3;
    google.protobuf.FloatValue message_size_in_kilobytes = 4;
    google.protobuf.BoolValue reply_processed = 5;
}
```

Этот файл эквивалентен файлу Greeting.proto, с которым мы работали ранее. Но на этот раз все поля допускают значение NULL. Это обусловлено тем, что мы импортировали пакет google/protobuf/wrappers.proto, который содержит все типы, допускающие значение NULL, такие как StringValue, Int32Value и т. д. Но это не единственный встроенный пакет, который мы можем импортировать, и общеизвестные типы не ограничиваются только лишь типами, допускающими значение NULL. Основные общеизвестные типы перечислены в табл. 9.2.

Таблица 9.2. *Общеизвестные типы данных из библиотеки Protobuf Google*

Тип данных	Имя пакета	Описание
Any	google/protobuf/any.proto	Тип, который может хранить данные любого типа. Эквивалент dynamic в C#
BoolValue	google/protobuf/wrappers.proto	Логическая величина, допускающая значение NULL
StringValue	google/protobuf/wrappers.proto	Строка, допускающая значение NULL
BytesValue	google/protobuf/wrappers.proto	Байты, допускающие значение NULL
FloatValue	google/protobuf/wrappers.proto	Величина с плавающей запятой (float), допускающая значение NULL

Таблица 9.2 *(окончание)*

Тип данных	Имя пакета	Описание
`DoubleValue`	google/protobuf/wrappers.proto	Величина с плавающей запятой двойной точности (`double`), допускающая значение `NULL`
`Int32Value`	google/protobuf/wrappers.proto	Целая величина (`int32`), допускающая значение `NULL`
`Int64Value`	google/protobuf/wrappers.proto	Целая величина (`int64`), допускающая значение `NULL`
`UInt32Value`	google/protobuf/wrappers.proto	Целая величина (`uint32`), допускающая значение `NULL`
`UInt64Value`	google/protobuf/wrappers.proto	Целая величина (`uint64`), допускающая значение `NULL`
`Duration`	google/protobuf/duration.proto	Эквивалент `TimeSpan` в C#
`Timestamp`	google/protobuf/timestamp.proto	Эквивалент `DateTime` в C#
`Empty`	google/protobuf/empty.proto	Заполнитель пустых запросов и ответов

Есть еще несколько типов данных, которые можно импортировать из библиотек Google. Но здесь перечислены основные, которых будет более чем достаточно для большинства практических случаев.

На этом обзор gRPC в ASP.NET Core завершен. Подведем итоги тому, что мы узнали.

Заключение

В этой главе мы рассмотрели технологию gRPC и ее реализацию в .NET. Теперь вы знакомы с тем, как использовать шаблоны проектов gRPC Service со всеми предустановленными зависимостями и как модифицировать зависимости gRPC в существующих приложениях ASP.NET Core. Мы рассмотрели настройку этой технологии как на клиенте, так и на сервере.

Мы увидели, как легко реализовать в своей программе gRPC и что писать вызовы между клиентом и сервером почти так же просто, как вызывать метод внутри отдельного приложения. В своих внутренних механизмах gRPC использует некоторые расширенные возможности HTTP/2 и HTTP/3, но мы никогда не видим их среди потребителей библиотек gRPC. Все это абстрагировано, что значительно упрощает процесс написания кода.

Мы также рассмотрели протокол Protobuf — основной протокол сериализации сообщений в gRPC. Мы изучили его синтаксис, его фундаментальную структуру и типы данных. Мы также научились расширять его возможности, обращаясь к внешним пакетам. Например, таким образом можно добавить в определения сообщений так называемые общеизвестные типы.

В следующей главе мы рассмотрим платформу ML.NET, которая представляет собой фреймворк на базе .NET для построения моделей машинного обучения.

Что следует помнить

- gRPC — это строго типизированный фреймворк взаимодействия, разработанный специально для использования новейших возможностей протокола HTTP/2.
- В gRPC в качестве основного протокола сериализации сообщений принят протокол Protobuf.
- В gRPC предусмотрены четыре типа вызовов: одиночные, потоковые клиентские, потоковые серверные и двунаправленные потоковые.
- Одиночный вызов предполагает один запрос и возвращает один ответ.
- Потоковый клиентский вызов ожидает от сервера поток сообщений и возвращает один ответ.
- Потоковый серверный вызов предполагает один запрос и возвращает поток сообщений.
- Потоковый двунаправленный вызов использует поток как от клиента к серверу, так и в обратном направлении.
- Типы данных по умолчанию в Protobuf не допускают значения `NULL`.
- Каждый RPC-вызов в gRPC должен иметь как тип сообщения запроса, так и тип сообщения ответа, хотя любой из них может быть помещен в поток.

Контрольные вопросы

1. Что такое тип данных fixed32 в Protobuf?

а) 32-битовое целое значение фиксированного размера, которое может хранить отрицательные числа;

б) 32-битовое целое значение фиксированного размера, которое может хранить только отрицательные числа;

в) 32-битовое значение с плавающей запятой фиксированного размера;

г) 32-битовое целое значение фиксированного размера, которое может хранить только положительные числа.

2. Что такое одиночный вызов в gRPC?

а) Вызов, принимающий одно сообщение запроса и возвращающий одно сообщение ответа;

б) Вызов, принимающий одно сообщение запроса и способный возвращать либо одно сообщение ответа, либо поток;

в) Вызов, который принимает либо одно сообщение запроса, либо поток запросов, но всегда возвращает один ответ;

г) Вызов, при котором потоковая передача может осуществляться в обе стороны.

3. Что из перечисленного является примером общеизвестного типа данных?

а) Int32;

б) double;

в) StringValue;

г) Все вышеперечисленное.

4. Какой пакет NuGet необходимо установить, чтобы добавить функции клиента gRPC?

а) Grpc.Net.Client;

б) Google.Protobuf;

в) Grpc.Tools;

г) Все вышеперечисленное.

Ответы

1. г.

2. а.

3. в.

4. г.

Основные термины

- **gRPC** — технология, первоначально разработанная компанией Google, которая была предназначена для организации эффективного взаимодействия между сервисами.
- **Protobuf** — основной протокол сериализации сообщений, используемый в gRPC.
- **Сервис gRPC** — структура Protobuf, содержащая сигнатуры конечных точек, к которым могут обращаться клиенты и которые могут реализовывать серверы.
- **Сообщение gRPC** — структура Protobuf, содержащая данные, которыми обмениваются сервер и клиент.
- **Одиночный вызов gRPC** — вызов RPC, который принимает один запрос и возвращает один ответ.
- **Потоковый клиентский вызов gRPC** — вызов RPC, который принимает от клиента поток сообщений и возвращает одно ответное сообщение.
- **Потоковый серверный вызов gRPC** — вызов RPC, который принимает от клиента одно сообщение запроса и возвращает поток ответных сообщений.
- **Потоковый двунаправленный вызов gRPC** — вызов RPC, который принимает поток сообщений от клиента и возвращает поток сообщений от сервера.
- **Общеизвестные типы данных** — типы данных Protobuf, которые существуют во всех популярных языках программирования, но должны быть добавлены в Protobuf посредством ссылок на внешние пакеты.

Глава 10

Машинное обучение с помощью ML.NET

Введение

Исторически *искусственный интеллект* (ИИ) и *машинное обучение* (МО) были областями интереса тех, кто являлся экспертом по статистике и высшей математике, например по математическому анализу. Кроме того, Python и R фактически были единственными языками программирования, на которых специалисты по анализу данных создавали модели МО. Но появление ML.NET изменило правила игры. С помощью этого инструмента любой .NET-разработчик может взять на себя роль специалиста по анализу данных. Им больше не нужно изучать R и Python. Хотя знание реальных математических формул для различных конкретных алгоритмов по-прежнему полезно, оно уже не является необходимым.

ML.NET — это инструмент, который помогает разработчикам строить модели МО, просто задавая комбинацию простых и интуитивно понятных параметров. Все, что нам нужно сделать, — это выполнить простую команду, указав инструменту, какую задачу мы хотим решить. Инструмент принимает входные данные, пытается понять, как с ними работать на основе заданных параметров, и генерирует код, который мы можем использовать в своих приложениях, когда нам потребуется решить задачу аналогичного характера. Например, мы можем определить, представляет ли запись с определенными свойствами человека моложе 24 лет, предсказать цену дома на основе некоторой информации о нем и т. д.

Модели МО потребовали бы сложного кода, если бы мы строили их с нуля. И их невозможно построить, если не знать, какой алгоритм лучше всего работает с тем или иным типом задачи и по какой формуле он работает. Но в ML.NET ни то ни другое не требуется. Этот инструмент не только сгенерирует код, но и попробует задействовать различные алгоритмы для выполнения поставленной задачи, чтобы определить наиболее подходящий из них. Разумеется, автоматически сгенерированный код может быть дополнительно оптимизирован человеком, обладающим знаниями в области МО. Но все же гораздо проще оптимизировать существующую модель, чем писать ее с нуля.

Чтобы дополнительно повысить удобство пользователя, для построения моделей ML.NET можно использовать визуальный малокодовый редактор. Вместо того что-

бы писать команду для построения модели, все необходимые параметры можно задать в форме ввода. Затем различные компоненты можно просто добавлять в модель, перетаскивая их.

Структура

В этой главе мы рассмотрим действия по установке и запуску ML.NET на всех поддерживаемых операционных системах и затронем следующие темы:

- основы ML.NET;
- выбор задачи для МО;
- обучение и оценка модели;
- использование малокодового построителя моделей.

Цели

К концу этой главы вы будете знать основы МО. Вы также узнаете, как использовать ML.NET и строить с его помощью модели, как из командной строки, так и в визуальном малокодовом редакторе.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- 64-разрядный компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из перечисленных компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Основы ML.NET

Прежде чем начать разговор об основах ML.NET, необходимо рассмотреть основы машинного обучения. Несмотря на то что для работы с ML.NET не обязательно быть специалистом по изучению данных или хорошо знать математику, все же необходимо понимать некоторые базовые концепции МО.

На самом базовом уровне машинное обучение — это возможность автоматической корректировки логики принятия решений на основе предоставленных данных, заложенная при написании программного обеспечения. Такой подход заметно отличается от традиционного способа принятия решений с помощью программных

средств, когда соответствующие функции просто жестко кодируются в виде условий с указанием ключевых слов `if` и `switch`. МО особенно полезно при решении сложных задач реального мира, когда логика принятия решений может быть скорректирована на основе постоянно меняющихся наборов данных или когда такая логика слишком сложна, чтобы ее можно было жестко закодировать в виде набора выражений `if`.

Существуют различные алгоритмы МО, которые имеют разную внутреннюю логику и основаны на разных формулах. Но у них есть ряд общих характеристик. На базовом уровне модель МО внутренне инициализируется некоторыми случайными числовыми параметрами, которые позволяют модели получать конкретные результаты. Затем выполняется процесс обучения путем прогона определенного набора входных данных через модель в серии многократных итераций. На каждой итерации по формуле, заложенной в модель, проводится корректировка значений внутренних параметров в зависимости от того, насколько результаты на выходе отличались от ожидаемого результата. В случае обучения без учителя эти параметры перенастраиваются для улучшения классификации элементов данных.

То, что происходит внутри модели, когда она адаптируется для получения лучших результатов, называется обучением. Процесс настройки процесса обучения называется тренировкой. Точность итоговой модели в этом случае будет зависеть от следующих факторов:

- объем данных, используемых для обучения;
- разнообразие представленных точек данных;
- количество итераций обучения;
- выбор алгоритма обучения;
- выбор конфигурации, характерной для данного алгоритма.

При этом приходится идти на некоторые компромиссы. Мы не можем применить бесконечный набор итераций и использовать миллиарды точек данных для обучения модели. Существует ограничение на объем дискового пространства, производительность процессора и объем памяти. Поэтому наша цель — сделать модель достаточно точной для той задачи, для которой мы ее строим, а не достичь полной безупречности.

Точность модели всегда можно определить в процессе обучения по тому, сколько результатов на выходе модели оказались верными или близкими к ожидаемому значению. Затем мы продолжаем обучение модели до тех пор, пока либо точность не станет приемлемой, либо алгоритм обучения не достигнет своего предела, когда дальнейшие итерации не ведут к увеличению точности.

После того как модель обучена, мы можем прогнать ее по записям, с которыми она раньше не сталкивалась. Затем, в зависимости от задачи, для которой мы построили модель, наша модель должна присвоить этой записи определенную категорию, сделать некоторые прогнозы и т. д.

Типы машинного обучения

Всего существует три типа машинного обучения: обучение с учителем, обучение без учителя и обучение с подкреплением. Рассмотрим каждую из этих категорий.

Обучение с учителем

При этом типе обучения входные данные используются вместе с ожидаемыми значениями. Например, если у нас есть запись с различными полями, такими как должность, уровень дохода и тип недвижимости, которой владеет человек (если таковая имеется), то в качестве ожидаемого выходного значения мы можем выбрать возраст человека. По мере обработки множества таких записей, модель будет постоянно корректировать свои внутренние параметры, пока не приблизится к ожидаемым результатам для как можно большего числа обучающих значений.

Обучение с учителем применяется для задач классификации и прогнозирования. Хороший пример — классификация изображений. В этом случае в модель вводят данные об изображении, а также о категории, к которой его следует отнести. Затем модель продолжает адаптироваться до тех пор, пока не сможет ассоциировать определенную комбинацию пикселов и цветов с определенной категорией.

Обучение без учителя

При обучении без учителя ожидаемые значения не задаются. Вместо этого данные и характеристики рассматриваются в некоторой собственной классификации. Целями могут быть объединение точек данных с общими характеристиками в кластер либо выявление отклонений в данных.

Обучение с подкреплением

Обучение с подкреплением — наиболее сложный из всех видов обучения. При обучении этого типа модель получает оценку за выполнение определенного вида действий. Чем более желательным является действие, тем выше оценка. Иными словами, в данном методе обучения сочетаются поощрения и наказания. В реальной жизни обучение с подкреплением применяется для обработки естественного языка, создания самоуправляемых автомобилей, обучения искусственного интеллекта видеоиграм и т. д.

Обучение с подкреплением не поддерживается ML.NET в исходной конфигурации. Однако эту функциональность можно добавить в ML.NET путем создания собственных алгоритмов, дополнения существующих классов новыми методами или изменения кода модели, которая была сгенерирована для другого типа задач.

Далее мы рассмотрим процесс установки инструментов ML.NET в нашей среде и их использование для построения нашей первой модели машинного обучения. Этот процесс будет несколько отличаться в зависимости от установленной ОС.

Начало работы с ML.NET

Прежде чем приступить к построению моделей МО с помощью ML.NET, необходимо установить необходимый набор инструментов для нашей среды.

Установка инструментов ML.NET

Команда для установки инструментов ML.NET через CLI будет несколько отличаться в зависимости от операционной системы и архитектуры процессора в компьютере, который вы используете для разработки. Мы рассмотрим все поддерживаемые сценарии.

- **Установка ML.NET в Windows**

  На компьютерах под управлением Windows с процессорами Intel/AMD для установки инструментов ML.NET можно выполнить следующую команду:

  ```
  dotnet tool install -g mlnet-win-x64
  ```

  Если в компьютере используется процессорная архитектура ARM, то команда будет выглядеть следующим образом:

  ```
  dotnet tool install -g mlnet-win-arm64
  ```

  Кроме того, Visual Studio IDE под Windows позволяет устанавливать эти инструменты через графический интерфейс. Для этого нам необходимо открыть установщик Visual Studio, выбрать пункт **.NET desktop development**, отметить вариант **ML.NET Model Builder** и нажать кнопку **Install** (Установить), как показано на рис. 10.1.

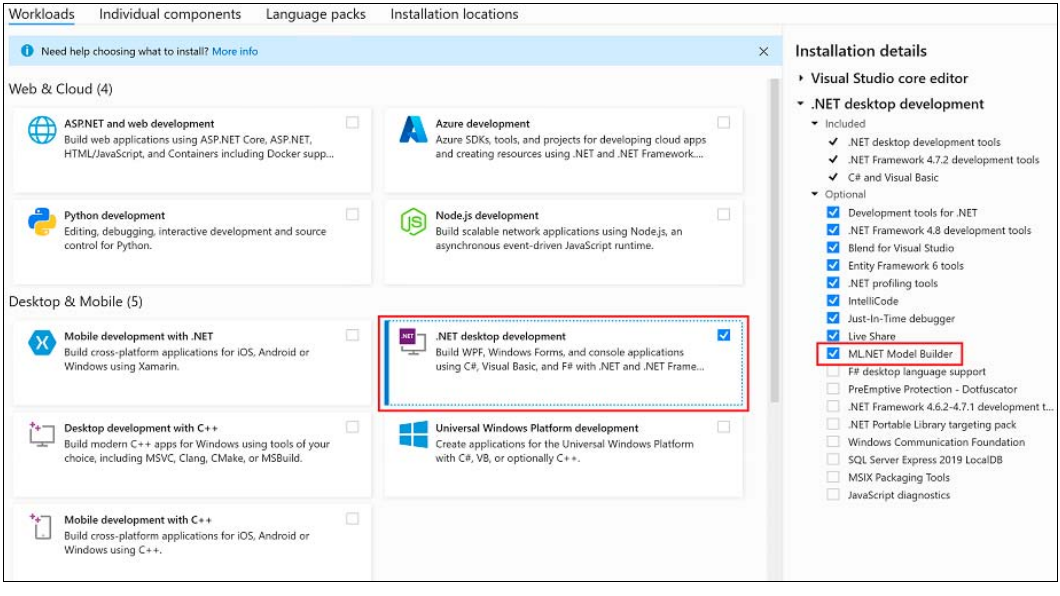


Рис. 10.1. Выбор ML.NET Model Builder в установщике Visual Studio

Visual Studio позволяет не только устанавливать инструменты ML.NET в малокодовом виде, но и настраивать модели МО с помощью графического интерфейса. Этот процесс мы рассмотрим далее в этой главе.

- **Установка ML.NET в Linux**

  На Linux-компьютерах, работающих на процессорах Intel/AMD (×64), нужно выполнить следующую команду установки:

```
dotnet tool install -g mlnet-linux-x64
```

  Для компьютеров на базе ARM команда установки будет выглядеть следующим образом:

```
dotnet tool install -g mlnet-linux-arm64
```

- **Установка ML.NET в MacOS**

  На компьютере Mac, работающем на процессорной архитектуре ×64, команда установки будет выглядеть следующим образом:

```
dotnet tool install -g mlnet-osx-x64
```

  Если процессор устройства Mac основан на архитектуре ARM64, то следует использовать следующую команду:

```
dotnet tool install -g mlnet-osx-arm64
```

  Теперь, когда инструменты ML.NET установлены, мы можем начать использовать их для построения модели МО.

Использование ML.NET для создания первой модели МО

Мы начнем с создания папки для нашей модели МО. Затем загрузим обучающий набор данных, на основе которого будем строить нашу модель. Мы возьмем известный набор данных, который часто применяется для подобных целей. Его можно получить по следующему URL-адресу: **https://archive.ics.uci.edu/ml/machine-learning-databases/00331/sentiment%20 labelled%20sentences.zip**.

Затем после загрузки zip-файла скопируем файл yelp_labelled.txt из этого архива в только что созданную папку. Этот текстовый файл содержит данные, разделенные символами табуляции, где первый столбец содержит входные данные, а второй — так называемую метку. В машинном обучении метка — это значение, которое выдает модель на основе входных данных. В данном случае мы имеем два целочисленных значения, отражающих "тональность" предложения. Здесь 1 означает положительное настроение, а 0 — отрицательное.

После копирования файла в указанную папку, можно выполнить следующую команду из этой папки:

```
mlnet classification --dataset "yelp_labelled.txt" --label-col 1 --has-
header false --train-time 60
```

В качестве задачи мы выбираем классификацию, а файл, который мы только что скопировали, — в качестве набора данных для обучения. Мы полагаем, что второй

столбец (столбец с индексом 1) — это наш столбец с метками, а также сообщаем модели, что данные не имеют заголовка и что первая строка должна рассматриваться как данные. Наконец, мы устанавливаем время обучения равным 60 секундам. Чем больше времени мы ему уделим, тем точнее будет наша модель.

В процессе обучения консоль показывает, какие алгоритмы обучения оцениваются, сколько времени затрачено на оценку каждого алгоритма и какова точность каждого алгоритма. Точность определяется тем, сколько предложений модель правильно классифицировала как 1 или 0 после просмотра исходных данных. Результат обучения будет выглядеть примерно так, как показано на рис. 10.2.

	Trainer	MicroAccuracy	MacroAccuracy	Duration	#Iteration
0	SdcaMaximumEntropyMulti	0.4838	0.5000	2.7	0
1	SdcaMaximumEntropyMulti	0.4838	0.5000	1.4	1
2	FastForestOva	0.7184	0.7185	6.7	2
3	FastTreeOva	0.6488	0.6473	3.3	3
4	SdcaLogisticRegressionOva	0.4838	0.5000	2.4	4
5	LightGbmMulti	0.6889	0.6855	2.9	5
6	LbfgsMaximumEntropyMulti	0.8060	0.8071	2.4	6
7	LbfgsLogisticRegressionOva	0.8047	0.8057	2.8	7
8	FastForestOva	0.7545	0.7548	14.0	8
9	SdcaMaximumEntropyMulti	0.6915	0.6937	1.4	9
10	LightGbmMulti	0.7142	0.7149	2.7	10
11	LbfgsMaximumEntropyMulti	0.7905	0.7914	4.6	11
12	FastTreeOva	0.7485	0.7489	6.3	12
13	SdcaLogisticRegressionOva	0.7292	0.7294	2.4	13

Рис. 10.2. Выходные данные процесса машинного обучения

После завершения процесса обучения внутри папки, в которой мы проводили обучение, создается новая папка. По умолчанию эта папка носит имя `SampleClassification`, которое совпадает с именем нашей модели. Но мы можем изменить это имя на любое другое, добавив параметр `--name` к команде `mlnel classification`, упомянутой ранее. Содержимое папки представляет собой консольное приложение с автоматически сгенерированным кодом для нашей модели.

Файл Program.cs в этой папке является точкой входа в приложение. Его следует выполнить для запуска модели. Файл {имя модели}.consumption.cs содержит входные и выходные данные модели. Он также содержит метод `Predict`, который предназначен для использования модели. Файл {имя модели}.mbconfig содержит подробную конфигурацию модели и результаты обучения. Файл {имя модели}.training.cs содержит код конвейера преобразования данных, который используется в процессе обучения. Наконец, папка {имя модели}.zip содержит собственно модель МО, используемую в данном коде.

Внутренний код в С-файлах структурирован таким образом, чтобы он был максимально интуитивно понятен опытным .NET-разработчикам, даже если они не имеют опыта в области машинного обучения. Например, класс, представляющий вход модели, будет выглядеть следующим образом (что также похоже на типичную абстракцию модели базы данных в таких фреймворках, как MVC и Entity Framework):

```
public class ModelInput
{
    [ColumnName(@"col0")]
    public string Col0 { get; set; }
    [ColumnName(@"col1")]
    public float Col1 { get; set; }
}
```

Основная часть функциональности, связанной с МО, доступна через класс `MLContext`, который концептуально аналогичен классу `DbContext`, используемому в Entity Framework. Конвейер преобразования, задействованный в процессе обучения, концептуально похож на конвейер обработки запросов в ASP.NET Core.

Наши обучающие данные представлены реализацией интерфейса `IDataView`, который передается в метод `RetrainPipeline`. Конвейер обучения, состоящий из нескольких этапов преобразования, реализует интерфейс `IEstimator<ITransformer>`. Нам необходимо несколько этапов преобразования входных данных, т. к. каждый алгоритм обучения может работать с данными только в определенной форме, поэтому мы не можем просто взять необработанные исходные данные. После того как мы построили конвейер, мы можем создать на его основе нашу модель, вызвав метод `Fit` для объекта конвейера, передав ему нашу реализацию `IDataView`. Таким образом, наш метод `RetrainPipeline` будет выглядеть следующим образом:

```
public static ITransformer RetrainPipeline(MLContext mlContext, IDataView trainData)
{
    var pipeline = BuildPipeline(mlContext);
    var model = pipeline.Fit(trainData);
    return model;
}
```

Мы построили нашу первую простую модель МО. Давайте посмотрим, что еще можно сделать с помощью ML.NET.

Выбор задачи для МО

Здесь приведен полный список задач, с которыми способна справиться платформа ML.NET. Для каждой из них мы сделаем общий обзор, разберемся, как построить команду для нее, и рассмотрим внутренние алгоритмы обучения, которые могут использовать задачи этого типа.

Бинарная классификация

Бинарная классификация — это процесс определения принадлежности конкретной записи в наборе данных к одной из двух категорий. Это самый простой вид классификации, который можно реализовать с помощью МО, поскольку категории могут быть определены либо двоичными (1 и 0), либо булевыми значениями ("истина" или "ложь").

Этот алгоритм подойдет для определения состояния занятости (работающий/безработный), наличия или отсутствия жилья у человека и во многих других ситуациях, когда требуется принять решение или представить прогноз, выбрав один из двух вариантов. Двоичная классификация — это один из видов обучения с учителем.

Задача классификации, которую мы решали ранее при выполнении команды `mlnet classification`, представляет собой пример двоичной классификации, поскольку у нас есть только две категории настроений: 1 и 0.

Модель бинарной классификации можно создать, выполнив следующую команду с соответствующими параметрами:

```
mlnet classification
```

Далее приведен список параметров, поддерживаемых следующей командой:

```
--dataset <dataset> (ОБЯЗАТЕЛЬНЫЙ)
Путь к файлу одиночного набора данных или обучающего набора данных для подходов
обучения/тестирования.

--label-col <label-col> (ОБЯЗАТЕЛЬНЫЙ)
Имя или индекс (начинающийся с нуля) для столбца метки (целевого объекта), который нужно
спрогнозировать.

--cache <Auto|Off|On>
Укажите [On|Off|Auto], чтобы включить или отключить кеш либо определять это автоматически
[по умолчанию: Auto].

--has-header
Укажите [true|false] в зависимости от того, есть ли в файлах наборов данных строка
заголовка. Используйте автоматическое обнаружение, если этот флаг не установлен.

--ignore-cols <ignore-cols>
Задайте столбцы, которые будут пропущены в указанном наборе данных. Укажите имена столбцов
или индексы (начинающиеся с нуля), разделив их пробелами.

--log-file-path <log-file-path>
Путь к файлу журнала.

--name <name>
Имя создаваемого выходного проекта или решения. По умолчанию используется
SampleClassification.

-o, --output <output>
Расположение созданных выходных данных. По умолчанию используется текущий каталог.

--test-dataset <test-dataset>
Путь к файлу для тестового набора данных в подходах обучения/тестирования.
```

```
--train-time <train-time>
Максимальное время (в секундах) для исследования моделей с наилучшей конфигурацией.
Время по умолчанию - 100 секунд. [по умолчанию: 100]


-v, --verbosity <verbosity>
Варианты уровня детализации выходных данных: q[uiet] (без вывода данных), m[inimal] (минимум
данных, используется по умолчанию) и diag[nostic] (диагностические данные).
[по умолчанию: m]
```

Список алгоритмов, пригодных для бинарной классификации, выглядит следующим образом:

```
AveragedPerceptronTrainer
SdcaLogisticRegressionBinaryTrainer
SdcaNonCalibratedBinaryTrainer
SymbolicSgdLogisticRegressionBinaryTrainer
LbfgsLogisticRegressionBinaryTrainer
LightGbmBinaryTrainer
FastTreeBinaryTrainer
FastForestBinaryTrainer
GamBinaryTrainer
FieldAwareFactorizationMachineTrainer
PriorTrainer
LinearSvmTrainer
```

Многоклассовая классификация

Многоклассовая классификация концептуально схожа с бинарной классификацией. Однако, как правило, в этом случае рассматриваются ситуации, когда категорий более двух. Это существенно усложняет задачу такого типа по сравнению с двоичной классификацией, поскольку категории уже не могут быть представлены ни двоичными, ни булевыми значениями.

К числу применений данного алгоритма можно отнести любые ситуации, где требуется определить принадлежность какой-либо записи к той или иной категории. Например, мы можем определять, к какой возрастной группе относится конкретный человек, на основании его интересов, дохода, должности и владения недвижимостью. Поскольку многоклассовая классификация концептуально схожа с бинарной, она относится к категории обучения с учителем.

Модель многоклассовой классификации можно создать, выполнив команду `mlnet classification`.

Это задача поддерживает следующие алгоритмы:

```
LightGbmMulticlassTrainer
SdcaMaximumEntropyMulticlassTrainer
SdcaNonCalibratedMulticlassTrainer
LbfgsMaximumEntropyMulticlassTrainer
NaiveBayesMulticlassTrainer
```

```
OneVersusAllTrainer
PairwiseCouplingTrainer
```

Регрессия

Регрессия — это попытка предсказать изменение конкретного признака на основе изменений связанных с ним признаков. При обучении модели на основе алгоритма регрессии мы пытаемся получить математическую функцию, которая будет давать достаточно точный результат при изменении значений признаков.

Регрессия позволяет прогнозировать изменение цен на жилье в будущем на основе характеристик жилья и исторических данных о рынке, а также прогнозировать изменение цен на акции. Регрессия — один из видов обучения с учителем.

Регрессионную модель можно создать путем выполнения следующей команды:

```
mlnet regression
```

Поддерживаемые аргументы:

```
--dataset <dataset> (ОБЯЗАТЕЛЬНЫЙ)
Путь к файлу одиночного набора данных или обучающего набора данных для подходов
обучения/тестирования.

--label-col <label-col> (ОБЯЗАТЕЛЬНЫЙ)
Имя или индекс (начинающийся с нуля) для столбца метки (целевого объекта), который нужно
спрогнозировать.

--cache <Auto|Off|On>
Укажите [On|Off|Auto], чтобы включить или отключить кеш либо определять это автоматически
[по умолчанию: Auto].

--has-header
Укажите [true|false] в зависимости от того, есть ли в файлах наборов данных строка
заголовка. Используйте автоматическое обнаружение, если этот флаг не установлен.

--ignore-cols <ignore-cols>
Задайте столбцы, которые будут пропущены в указанном наборе данных. Укажите имена столбцов
или индексы (начинающиеся с нуля), разделив их пробелами.

--log-file-path <log-file-path>
Путь к файлу журнала.

--name <name>
Имя создаваемого выходного проекта или решения. По умолчанию: SampleRegression.

-o, --output <output>
Расположение созданных выходных данных. По умолчанию используется текущий каталог.

--test-dataset <test-dataset>
Путь к файлу для тестового набора данных в подходах обучения/тестирования.
```

```
--train-time <train-time>
Максимальное время (в секундах) для исследования моделей с наилучшей конфигурацией.
Время по умолчанию - 100 секунд. [по умолчанию: 100]

-v, --verbosity <verbosity>
Варианты уровня детализации выходных данных: q[uiet] (без вывода данных), m[inimal] (минимум
данных, используется по умолчанию) и diag[nostic] (диагностические данные).
[по умолчанию: m]
```

Задача регрессии поддерживает следующие алгоритмы:

```
LbfgsPoissonRegressionTrainer
LightGbmRegressionTrainer
SdcaRegressionTrainer
OlsTrainer
OnlineGradientDescentTrainer
FastTreeRegressionTrainer
FastTreeTweedieTrainer
FastForestRegressionTrainer
GamRegressionTrainer
```

Рекомендации

Цель этой задачи — составить список рекомендуемых элементов на основе предшествующих данных. Например, мы можем предлагать пользователю рекомендации по товарам, основываясь на том, какие товары он покупал в прошлом.

Этот тип задач относится к категории обучения без учителя, поскольку у нас нет однозначного ответа, на основе которого можно было бы обучить модель. Вместо этого алгоритм сам устанавливает взаимосвязи в данных.

Модель МО для рекомендаций можно создать путем выполнения следующей команды:

```
mlnet recommendation
```

Поддерживаются следующие параметры:

```
--dataset <dataset> (ОБЯЗАТЕЛЬНЫЙ)
Путь к файлу одиночного набора данных или обучающего набора данных для подходов
обучения/тестирования.

--item-col <item-col> (ОБЯЗАТЕЛЬНЫЙ)
Имя или индекс (начинающийся с нуля) для столбца элемента. Элементы рекомендуются
пользователям.

--rating-col <rating-col> (ОБЯЗАТЕЛЬНЫЙ)
Имя или индекс (начинающийся с нуля) для столбца рейтинга (целевого объекта), который нужно
спрогнозировать.

--user-col <user-col> (ОБЯЗАТЕЛЬНЫЙ)
Имя или индекс (начинающийся с нуля) для пользовательского столбца. Пользователи получают
рекомендованные элементы.
```

```
--cache <Auto|Off|On>
Укажите [On|Off|Auto], чтобы включить или отключить кеш либо определять это автоматически
[по умолчанию: Auto].

--has-header
Укажите [true|false] в зависимости от того, есть ли в файлах наборов данных строка
заголовка. Используйте автоматическое обнаружение, если этот флаг не установлен.

--log-file-path <log-file-path>
Путь к файлу журнала.

--name <name>
Имя создаваемого выходного проекта или решения. По умолчанию: SampleRecommendation.

-o, --output <output>
Расположение созданных выходных данных. По умолчанию используется текущий каталог.

--test-dataset <test-dataset>
Путь к файлу для тестового набора данных в подходах обучения/тестирования.

--train-time <train-time>
Максимальное время (в секундах) для исследования моделей с наилучшей конфигурацией.
Время по умолчанию - 100 секунд. [по умолчанию: 100]

-v, --verbosity <verbosity>
Варианты уровня детализации выходных данных: q[uiet] (без вывода данных), m[inimal] (минимум
данных, используется по умолчанию) и diag[nostic] (диагностические данные).
[по умолчанию: m]
```

Задача поддерживает следующий алгоритм:

```
MatrixFactorizationTrainer
```

Прогнозирование

С помощью прогнозирования можно предсказывать будущие события на основе некоторых прошлых данных, например погоду или цены на фондовом рынке, или определять время, когда следует выполнить то или иное действие, например провести техническое обслуживание автомобиля.

Для прогнозирования можно применить задачи обучения с учителем, такие как классификация и регрессия. Однако для этой задачи также предусмотрены алгоритмы, предназначенные именно для прогнозирования. Также, в зависимости от типа прогнозирования, она может относиться к категории обучения без учителя. В этом случае прогнозирование будет осуществляться по закономерности, имеющейся в исторических данных.

Для обучения модели прогнозирования предназначена команда:

```
mlnet forecasting
```

которая поддерживает следующие параметры:

```
--dataset <dataset> (ОБЯЗАТЕЛЬНЫЙ)
Путь к файлу одиночного набора данных или обучающего набора данных для подходов
обучения/тестирования.

--horizon <horizon> (ОБЯЗАТЕЛЬНЫЙ)
Определяет, на сколько периодов наперед требуется прогноз. Горизонт устанавливается
в единицах частоты временного ряда. Единицы измерения основаны на интервале времени
обучающих данных, например ежемесячно, еженедельно и т. д., который должен предсказать
аналитик.

--label-col <label-col> (ОБЯЗАТЕЛЬНЫЙ)
Имя или индекс (начинающийся с нуля) для столбца метки (целевого объекта), который нужно
спрогнозировать.

--time-col <time-col> (ОБЯЗАТЕЛЬНЫЙ)
Используется для указания столбца даты/времени во входных данных, используемых
для построения временного ряда и выведения его частоты.

--cache <Auto|Off|On>
Укажите [On|Off|Auto], чтобы включить или отключить кеш либо определять это автоматически
[по умолчанию: Auto].

--has-header
Укажите [true|false] в зависимости от того, есть ли в файлах наборов данных строка
заголовка. Используйте автоматическое обнаружение, если этот флаг не установлен.

--log-file-path <log-file-path>
Путь к файлу журнала.

--name <name>
Имя создаваемого выходного проекта или решения. По умолчанию используется
SampleClassification. [по умолчанию: SampleClassification]

-o, --output <output>
Расположение созданных выходных данных. По умолчанию используется текущий каталог.

--test-dataset <test-dataset>
Путь к файлу для тестового набора данных в подходах обучения/тестирования.

--train-time <train-time>
Максимальное время (в секундах) для исследования моделей с наилучшей конфигурацией.
Время по умолчанию - 100 секунд. [по умолчанию: 100]

-v, --verbosity <verbosity>
Варианты уровня детализации выходных данных: q[uiet] (без вывода данных), m[inimal] (минимум
данных, используется по умолчанию) и diag[nostic] (диагностические данные).
[по умолчанию: m]
```

Для прогнозирования можно использовать следующий алгоритм:

```
ForecastBySsa
```

Классификация изображений

Классификация изображений — это разновидность многоклассовой классификации, но учитывающая специфику изображений. Для определения того, какие комбинации цветов и пикселов характерны для определенной категории, используется сложный алгоритм, обычно включающий многоуровневую обработку, например с помощью нейронной сети. Затем предпринимается попытка отнести новое изображение к одной из известных категорий. Поскольку классификация изображений является разновидностью многоклассовой классификации, она представляет собой пример обучения с учителем.

Обучение модели классификации изображений может быть выполнено с помощью следующей команды:

```
mlnet image-classification
```

В этой команде используются следующие параметры:

```
--dataset <dataset> (ОБЯЗАТЕЛЬНЫЙ)
Путь к локальной папке, содержащей подписанные вложенные папки со всеми изображениями.

--cache <Auto|Off|On>
Укажите [On|Off|Auto], чтобы включить или отключить кеш либо определять это автоматически
[по умолчанию: Auto].

--log-file-path <log-file-path>
Путь к файлу журнала.

--name <name>
Имя создаваемого выходного проекта или решения.
По умолчанию используется SampleImageClassification.

-o, --output <output>
Расположение созданных выходных данных. По умолчанию используется текущий каталог.

--test-dataset <test-dataset>
Путь к родительской папке, содержащей тестовый набор данных при подходах обучения
и тестирования.

-v, --verbosity <verbosity>
Варианты уровня детализации выходных данных: q[uiet] (без вывода данных), m[inimal] (минимум
данных, используется по умолчанию) и diag[nostic] (диагностические данные).
[по умолчанию: m]
```

Для классификации изображений можно использовать следующий алгоритм обучения:

```
ImageClassificationTrainer
```

Кластеризация

Кластеризация — это процесс объединения записей в кластеры на основе некоторых общих признаков. С ее помощью можно устанавливать связи между различными записями на основе их сходства. Кластеризация очень полезна в маркетинге для понимания особенностей целевой аудитории. Это пример обучения без учителя.

Для кластеризации нет собственной отдельной команды. Модель МО, предназначенную для решения подобной задачи, можно создать с помощью типовой команды:

```
mlnet train
```

со следующими параметрами:

```
--training-config <training-config> (ОБЯЗАТЕЛЬНЫЙ)
Путь к файлу конфигурации обучения

--log-file-path <log-file-path>
Путь к файлу журнала.

-v, --verbosity <verbosity>
Варианты уровня детализации выходных данных: q[uiet] (без вывода данных), m[inimal] (минимум
данных, используется по умолчанию) и diag[nostic] (диагностические данные).
[по умолчанию: m]
```

Выбор задачи и другие настройки задаются в файле с расширением mbconbfig, подобном тому, который мы сгенерировали ранее. Путь к этому файлу добавляется с помощью параметра `--training-config`.

Алгоритм обучения такой модели МО:

```
KMeansTrainer
```

Обнаружение отклонений

Обнаружение отклонений — это процесс выявления точек данных, которые выглядят неуместными в заданном наборе данных. Например, с его помощью можно определить аномальное поведение веб-трафика, чтобы выявить возможные атаки на сервер.

Обнаружение отклонений — это задача, которая может относиться к категории обучения как с учителем, так и без него. При обучении с учителем каждому элементу обучающих данных можно присвоить двоичную метку, позволяющую идентифицировать его как отклонение, что делает эту задачу похожей на задачу двоичной классификации. С другой стороны, для отделения аномальных данных от остальных можно использовать действие, подобное кластеризации, поскольку такие отклонения будут иметь другие характеристики.

Для задачи обнаружения отклонений специальная отдельная команда не предусмотрена. Вместо этого применяется общая команда `mlnet train`, а данные конфигу-

рации добавляются в файл mbconfig. Алгоритм обучения для задачи обнаружения отклонений следующий:

```
RandomizedPcaTrainer
```

Ранжирование

В этом типе задач МО каждой записи соответствует числовое значение ранга. Задача работы с моделью состоит в том, чтобы выяснить, как комбинация признаков в данных определяет этот ранг.

Это тип обучения с учителем, который концептуально похож на многоклассовую классификацию. Однако есть и существенное отличие. Хотя категории в многоклассовой классификации различны, каждое значение ранга показывает, является ли конкретная запись "лучше" или "хуже", чем другие записи. Поэтому модель, построенная для этой конкретной задачи, может служить в качестве основы для задачи обучения с подкреплением.

Для задачи ранжирования отдельная команда не предусмотрена. Вместо этого используется общая команда `mlnet train`, а данные конфигурации добавляются в файл mbconfig. Алгоритмы для задачи ранжирования следующие:

```
LightGbmRankingTrainer
FastTreeRankingTrainer
```

Обучение и оценка модели

При обучении модели МО алгоритмы, участвующие в этом процессе, пытаются выяснить взаимосвязь между входными данными (функциями) и выходными данными (метками). Затем подмножество обучающих данных используется для оценки точности модели путем передачи в нее входных значений и проверки значений на выходе. В процессе обучения, чтобы повысить точность при следующем запуске, корректируются некоторые внутренние переменные модели.

Более длительное обучение моделей обычно повышает их точность. Но при обработке сложных реальных данных модель редко достигает 100-процентной точности. Поэтому существует риск, что если мы будем слишком долго отлаживать модель, то мы просто зря потратим время и ресурсы. Также существует риск чрезмерно близкой подгонки модели к выборочным данным. В этом случае модель демонстрирует очень хорошее соответствие данным из конкретного набора, но ее конфигурация недостаточно универсальна. Когда такое происходит, модель делает точные прогнозы на основе входных данных из обучающего набора, но плохо работает с данными, с которыми она никогда раньше не сталкивалась. Вот почему следует стремиться не к 100-процентной точности, а к некоторой приемлемой степени точности. Именно поэтому нам необходимо проверять и другие показатели, а не только точность.

Показатели, которые мы можем оценить, отображаются во время обучения модели и после завершения обучения. Набор доступных показателей зависит от задачи, для выполнения которой мы обучаем нашу модель. Давайте рассмотрим их.

Показатели бинарной классификации

При обучении модели бинарной классификации отображаются следующие показатели.

- **Точность** — указывает, как часто модель дает правильный ответ. Значение может меняться от нуля до единицы. Чем ближе это значение к единице, тем выше точность. Но если оно равно единице, это указывает на потенциальную проблему чрезмерно близкой подгонки (переобучения).
- **Площадь под кривой (AUC)** — кривая формируется путем сравнения истинно положительного показателя с ложноположительным. Значение должно быть как можно ближе к единице, но при этом должно быть больше 0,5. Если значение меньше 0,5, модель бесполезна.
- **Площадь под кривой "точность–полнота" (AUCPR)** — значение, близкое к единице, указывает на то, что модель возвращает точные результаты и большинство из них положительные. Этот показатель особенно полезен при сильном дисбалансе в наборах данных.
- **Показатель F1** — величина этого показателя меняется от нуля до единицы и указывает, насколько точен используемый классификатор. Чем ближе эта величина к единице, тем лучше классификатор.

Показатели многоклассовой классификации

Для многоклассовой классификации предусмотрены свои собственные показатели, отличные от тех, которые используются для двоичной классификации.

- **Микроточность** — эта величина может меняться от нуля до единицы. Чем она ближе к единице, тем лучше. Этот показатель предпочтительнее макроточности.
- **Макроточность** — эта величина может меняться от нуля до единицы. Чем она ближе к единице, тем лучше.
- **Логарифмическая потеря** — эта величина может меняться от единицы до нуля. Чем она ближе к нулю, тем лучше.
- **Снижение логарифмической потери** — величина может меняться от отрицательной бесконечности до единицы. Величина меньше нуля указывает, что модель работает хуже, чем случайное предположение. Чем она ближе к единице, тем лучше.

Классификация изображений

Для классификации изображений предусмотрены следующие показатели.

- **Логарифмическая потеря** — эта величина может меняться от единицы до нуля. Чем она ближе к нулю, тем лучше.
- **Логарифмическая потеря для класса** — диапазон значений логарифмической потери рассчитывается для каждой категории изображений. Это помогает опре-

делить, насколько хорошо модель идентифицирует конкретные типы изображений.

Прогнозирование

Для прогнозирования предусмотрены следующие показатели.

- **Средняя абсолютная ошибка** — эта величина представляет собой среднюю разницу между фактическим и прогнозируемым значением. Чем меньше эта величина, тем лучше.
- **Среднеквадратическая ошибка** — этот показатель похож на среднюю абсолютную ошибку, но основан на квадратном корне из квадрата разности прогнозируемого и фактического значений. Данный показатель помогает выявлять значительные отдельные ошибки, поскольку возведение большого числа в квадрат дает большое среднее число.

Регрессия и рекомендации

У этих задач один и тот же набор показателей.

- **R-квадрат** — эта величина может меняться от нуля до единицы. Чем она ближе к единице, тем лучше. Однако значение, равное единице, может указывать на переобучение модели. Значение 0,5 обычно считается приемлемым.
- **Абсолютная потеря** — этот показатель имеет положительные значения. Чем он ближе к нулю, тем лучше.
- **Квадратичная потеря** — этот показатель имеет положительные значения. Чем он ближе к нулю, тем лучше. Однако относительно большое значение не обязательно указывает на проблему.
- **Потеря RMB** — этот показатель имеет положительные значения. Как и в случае с любым типом потерь, чем ближе его величина к нулю, тем лучше.

Показатели кластеризации

Точность модели кластеризации можно оценить с помощью следующих показателей.

- **Среднее расстояние** — чем ближе эта величина к нулю, тем лучше модель. Более близкая к нулю величина коррелирует с более кластеризованными данными.
- **Индекс Дэвиса — Боулдина** — чем ближе этот показатель к нулю, тем лучше. Более рассредоточенные данные дают лучший результат.
- **Нормализованная взаимная информация** — эта величина меняется между нулем и единицей. Чем она ближе к единице, тем лучше.

Показатели обнаружения отклонений

Если требуется оценить характеристики алгоритма обнаружения отклонений, следует обратить внимание на следующие показатели.

- **Площадь под кривой ROC** — эта величина может меняться от нуля до единицы. Чем она ближе к единице, тем лучше. Однако значения ниже 0,5 означают, что модель бесполезна.
- **Коэффициент обнаружения при подсчете ложноположительных результатов** — эта величина может меняться от нуля до единицы. Чем она ближе к единице, тем лучше. Это означает, что модель дает меньше ложноположительных результатов.

Показатели ранжирования

Ранжирование моделей МО можно оценить с помощью следующих показателей.

- **Дисконтированный совокупный доход** — числовое значение этого показателя не ограничено. Чем оно выше, тем лучше.
- **Нормализованный дисконтированный совокупный доход** — числовое значение этого показателя может меняться от нуля до единицы. Чем оно ближе к единице, тем лучше.

На этом обзор основных показателей для каждого типа моделей завершен. До сих пор для построения нашей модели мы использовали сочетание команд CLI и кода. Но если вы являетесь пользователем Windows, то можете создавать малокодовые модели ML.NET посредством графического интерфейса. Этот подход мы рассмотрим далее.

Использование малокодового построителя моделей

Чтобы узнать, как построить модель машинного обучения, используя только графический интерфейс, для начала требуется включить рабочую нагрузку ML.NET через программу установки Visual Studio Installer, как было описано ранее. Затем следует создать в Visual Studio проект консольного приложения. После этого щелкните правой кнопкой мыши на созданном проекте, выберите пункт **Add** (Добавить) и далее пункт **Machine Learning Model** (Модель машинного обучения), как показано на рис. 10.3.

В появившемся диалоговом окне выберите шаблон **Machine Learning Model (ML.NET)**, как показано на рис. 10.4.

На экране отображается окно, где мы сможем выбрать сценарий машинного обучения, как показано на рис. 10.5.

Затем мы можем указать, проводить ли машинное обучение локально или в Azure. Некоторые задачи допускают только один вариант, тогда как другие поддерживают

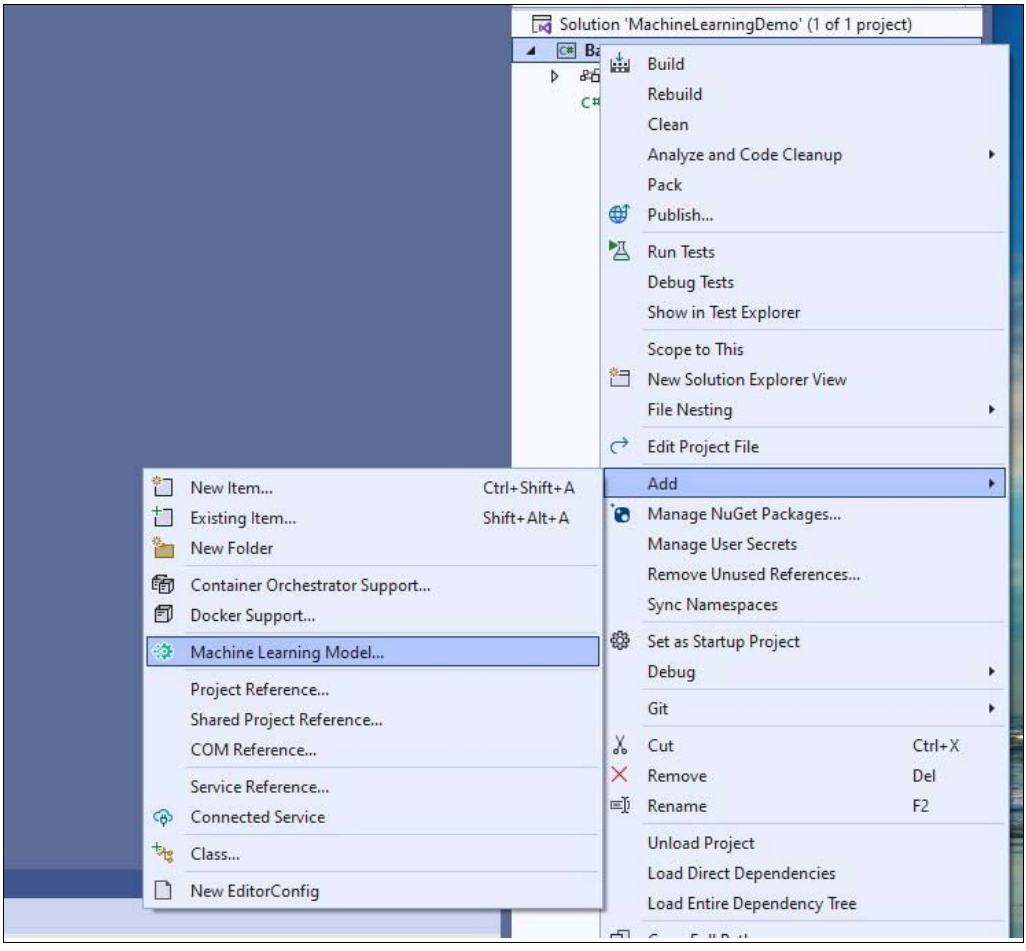


Рис. 10.3. Добавление модели машинного обучения в консольное приложение

оба. После выбора нужного варианта и нажатия кнопки **Next** (Далее) отображается следующий экран, где можно выбрать источник набора данных — БД SQL либо файл на диске. После загрузки набора данных его можно просмотреть и выбрать столбец для меток, как показано на рис. 10.6.

На следующем экране (рис. 10.7) можно настроить параметры и начать обучение нашей модели. После завершения обучения мы сможем увидеть, насколько точна наша модель. Мы также можем оценить ее, передав в нее произвольные входные данные и проверив правильность возвращаемых ответов. Затем, когда мы будем довольны полученной моделью, можно перейти к ее использованию или развертыванию приложения. Мы также можем переобучить либо улучшить нашу модель, если она не полностью нас устраивает.

Этот процесс "за кадром" создает тот же код, который мы рассматривали, когда использовали командную строку. Кроме того, во время обучения на консоль в Visual Studio будут выводиться те же данные, что и при вводе команд CLI. Оба инструмента будут создавать одинаковый код, который затем можно будет изменить.

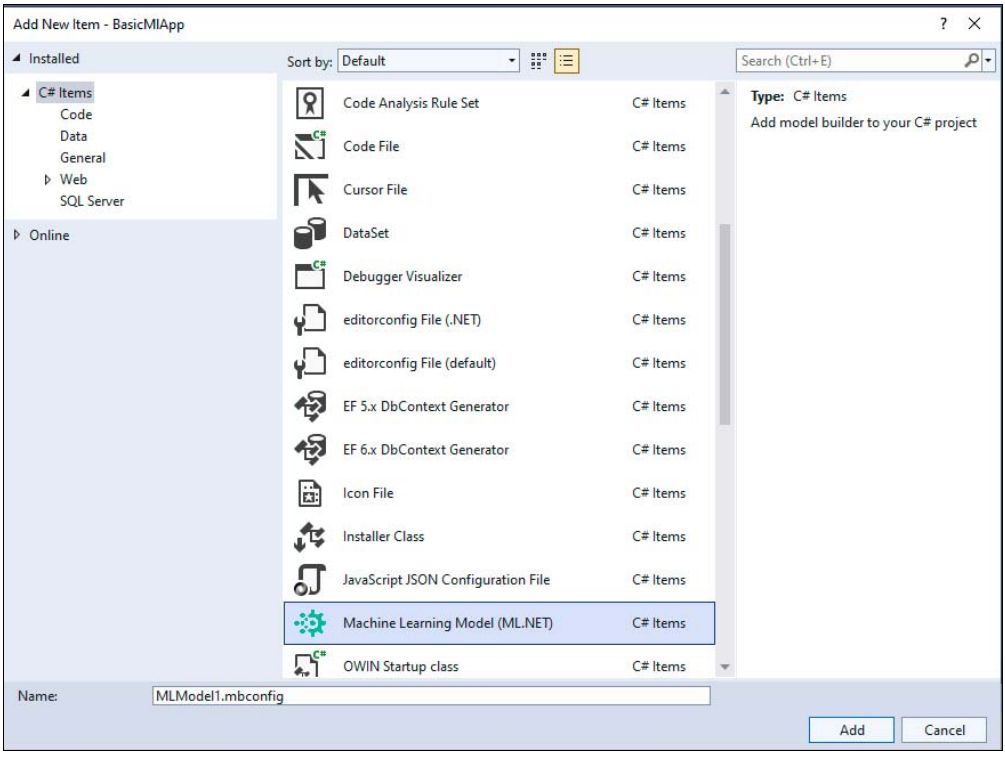


Рис. 10.4. Выбор шаблона ML.NET

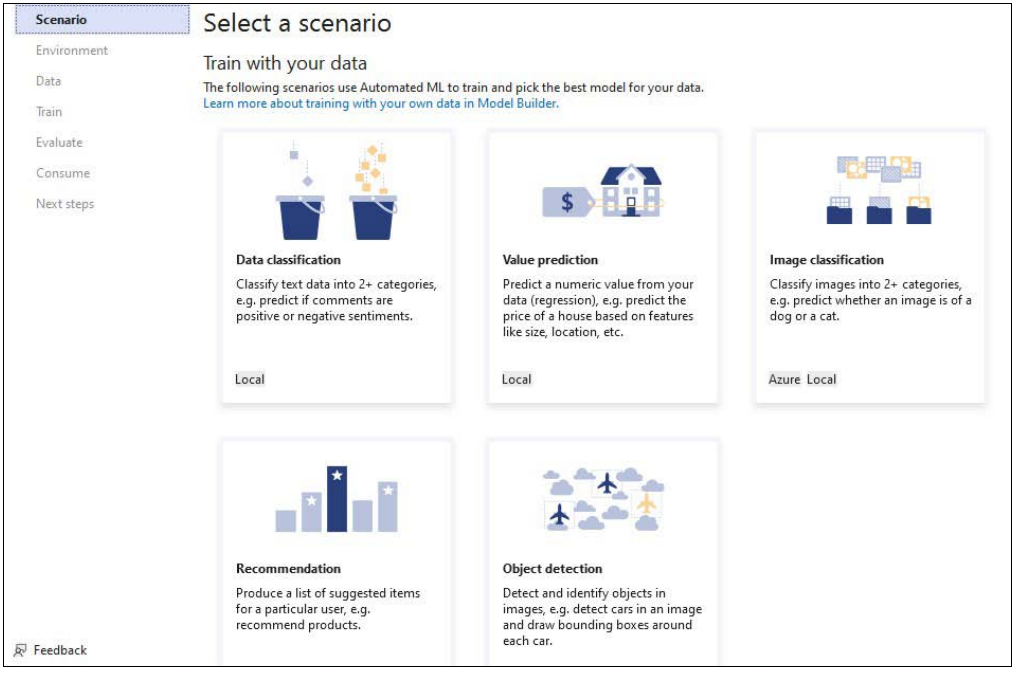


Рис. 10.5. Выбор сценария машинного обучения

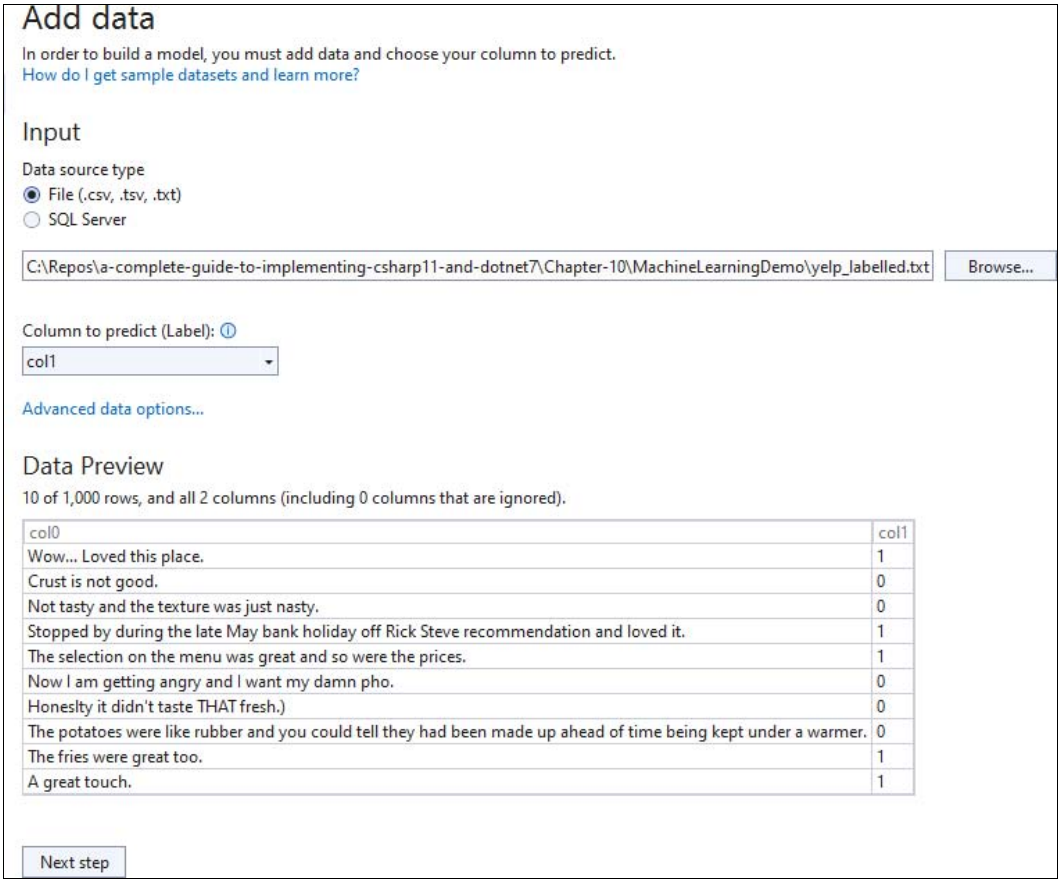


Рис. 10.6. Просмотр набора данных и выбор столбца для меток

Заключение

В этой главе мы рассмотрели основы машинного обучения, которые должен знать каждый, кто хочет применять эту технологию на практике или анализировать точность моделей машинного обучения. Мы рассмотрели типы машинного обучения и типы задач, которые помогает решать каждый тип обучения.

Мы также рассмотрели применение ML.NET — инструмента, позволяющего генерировать код модели МО путем выполнения интуитивно понятной команды с несложным набором параметров. Мы научились работать с этим инструментом как посредством команд CLI, так и с помощью визуального редактора.

Мы рассмотрели все задачи, которые поддерживает ML.NET, и все алгоритмы, которые предусмотрены в каждой из этих задач. Теперь вы знаете, что стандартная конфигурация ML.NET поддерживает множество задач обучения с учителем и без учителя. В стандартной конфигурации ML.NET обучение с подкреплением не поддерживается, но мы кратко рассмотрели, как можно решать такие задачи с по-

мощью ML.NET — путем написания дополнительных методов либо путем добавления нужных функций в сгенерированный код.

В следующей главе мы рассмотрим процесс запуска приложений .NET внутри контейнеров, чтобы их можно было развертывать где угодно вместе со средой выполнения.

Что следует помнить

- ML.NET — это инструмент на основе .NET, который помогает создавать модели машинного обучения с помощью команд CLI или через графический пользовательский интерфейс.
- Всего существуют три типа машинного обучения: с учителем, без учителя и с подкреплением. ML.NET поддерживает первые два типа в стандартной конфигурации.
- ML.NET генерирует код модели, которую он строит для решения задачи, указанной для представленного набора данных.
- Точность модели зависит от применяемого алгоритма и заданных входных параметров, таких как число итераций обучения и глубина используемой нейронной сети, если выбранные алгоритмы ее задействуют.
- ML.NET может оценивать несколько алгоритмов и выбирать тот, который дает наилучшие результаты.

Контрольные вопросы

1. Какие типы обучения можно использовать для обучения моделей МО?

а) Обучение с учителем;

б) Обучение без учителя;

в) Обучение с подкреплением;

г) Все вышеперечисленные.

2. Какова роль класса MLContext в ML.NET?

а) Он предоставляет доступ к большинству функций МО;

б) Он хранит данные, загруженные в модель;

в) Он хранит преобразованные данные;

г) Он хранит результаты оценки модели.

3. Что такое многоклассовая классификация?

а) Процесс прогнозирования наличия у заданного объекта определенного свойства;

б) Назначение входным данным одной из двух категорий;

в) Назначение входным данным одной из произвольного числа категорий;

г) Определение принадлежности конкретного изображения к определенной категории.

4. **Какая из следующих задач не поддерживается в стандартной конфигурации ML.NET?**

а) Регрессия;

б) Ранжирование;

в) Обработка естественного языка;

г) Все вышеперечисленное поддерживается.

Ответы

1. г.
2. а.
3. в.
4. в.

Основные термины

- **ML.NET** — инструмент на основе .NET, позволяющий создавать модели машинного обучения, представленные в коде C#.
- **Обучение с учителем** — тип машинного обучения, при котором необработанные входные данные предоставляются вместе с ожидаемыми результатами.
- **Обучение без учителя** — тип машинного обучения, при котором в данных ведется поиск общих закономерностей без предоставления каких-либо указаний.
- **Обучение с подкреплением** — тип машинного обучения, при котором модель побуждается к выполнению определенных действий с помощью механизма вознаграждения/наказания.
- **Бинарная классификация** — задача машинного обучения, которая определяет, принадлежит ли конкретная запись к одной из двух категорий.
- **Многоклассовая классификация** — задача машинного обучения, которая определяет, принадлежит ли конкретная запись к одной из нескольких категорий.
- **Регрессия** — попытка предсказать значение из набора связанных значений.
- **Кластеризация** — задача обучения без учителя, в которой различные записи распределяются по разным группам на основе схожих характеристик.
- **Обнаружение отклонений** — задача обучения без учителя, которая позволяет обнаружить значения, не соответствующие общей закономерности.
- **Ранжирование** — задача присвоения рангов различным точкам данных на основе некоторых примеров.

- **Рекомендации** — задача машинного обучения, представляющая пользователям рекомендации на основе данных, собранных за предыдущие периоды.
- **Прогнозирование** — задача машинного обучения, целью которой является определение значения некоторого признака на основе исторического набора значений.
- **Классификация изображений** — задача определения категории, к которой относится изображение, которая часто совпадает с задачей определения того, что на нем изображено.

Глава 11

Микросервисы и контейнеризация в .NET 7

Введение

Многим популярным веб-приложениям приходится иметь дело с большими объемами данных. Например, такие приложения, как Facebook и Twitter, в настоящее время обрабатывают огромные объемы трафика. Они должны поддерживать миллионы пользователей, одновременно участвующих в живом взаимодействии, сохраняя при этом высокую производительность.

Веб-приложения, которым приходится иметь дело с крупномасштабным использованием, обычно создаются как совокупность распределенных микросервисов, а не в виде единого монолитного приложения. Каждый микросервис отвечает только за определенную функцию или набор связанных функций. Таким образом, каждую часть системы можно масштабировать независимо, а нагрузку распределять между любым числом отдельных компонентов.

Обычно высокопроизводительные приложения реализуют два разных типа масштабирования: вертикальное (увеличение производительности компонентов) и горизонтальное (увеличение числа компонентов). Процесс вертикального масштабирования включает в себя добавление дополнительных аппаратных ресурсов, таких как ОЗУ и ЦП, чтобы сделать компьютер более производительным. Этот подход работает, но имеет свои пределы. С другой стороны, горизонтальное масштабирование предусматривает распределение нагрузки между большим количеством экземпляров приложения. При хорошо реализованной архитектуре этот процесс практически не имеет ограничений. Именно здесь микросервисы показывают свою полезность.

Чтобы сделать микросервисы еще более эффективными, была изобретена контейнеризация. Этот процесс позволяет развертывать каждое приложение вместе со средой его выполнения. Приложение будет полностью изолировано от внешнего мира, за исключением некоторых его частей, которые можно настроить соответствующим образом. В результате обеспечивается согласованное поведение приложения в любой среде, поскольку никакие операции обработки, выполняемые на хосте, не смогут оказать каких-либо непреднамеренных побочных воздействий на работу приложения. Кроме того, есть гарантия, что приложение сможет функционировать

на любом компьютере, поддерживающем определенную технологию контейнеризации, независимо от того, под управлением какой операционной системы работает компьютер и какие зависимости системного уровня на нем установлены.

Основная технология контейнеризации, используемая в ИТ-индустрии, — это Docker. Несмотря на наличие альтернатив, Docker является де-факто стандартной технологией контейнеризации, которая также интегрирована с .NET SDK. Именно об этом мы и поговорим в этой главе.

Структура

В этой главе мы рассмотрим основы контейнеров Docker и их применение в контексте .NET. Мы затронем следующие темы:

- контейнеры Docker;
- базовый образ Docker для .NET 7;
- оркестрация приложений с помощью Docker Swarm;
- оркестрация приложений с помощью Kubernetes.

Цели

Прочитав эту главу, вы узнаете, как опубликовать .NET-приложение внутри контейнера и как управлять распределенным контейнерным приложением с помощью Docker Swarm и Kubernetes.

Предварительные условия

При прочтении этой главы вам понадобится следующее:

- компьютер с ОС Windows, MacOS или Linux;
- .NET 7 SDK;
- подходящая среда разработки (IDE) или редактор кода;
- знание основ C#.

Если у вас отсутствует какой-либо из перечисленных компонентов, ознакомьтесь с указаниями по установке, приведенными в *главе 1*, которая также содержит обзор основ C#.

Контейнеры Docker

Контейнер Docker можно рассматривать как облегченную *виртуальную машину* (ВМ). Обычная виртуальная машина — это виртуальная версия компьютера, которая работает внутри операционной системы или физического компьютера и использует его ресурсы совместно с другими компонентами. По сути это полнофункциональный компьютер, изолированный от среды своего хоста. Он имеет полно-

функциональную операционную систему, доступ к нему осуществляется через пользовательский интерфейс, и в остальном его можно использовать как обычный компьютер.

Эти особенности виртуальной машины делают ее достаточно тяжеловесной. Несмотря на то что можно ограничить число аппаратных средств, к которым может получить доступ ВМ, ей все равно потребуется разумное количество ресурсов только для выполнения самых основных операций. ВМ также занимает значительный объем памяти на диске из-за необходимости иметь собственную полноценную операционную систему. Это также делает процесс перемещения виртуальной машины на другое оборудование несколько сложным и трудоемким.

Но технология Docker все это изменила. Контейнер Docker, как и виртуальная машина, также представляет собой изолированную среду, внутри которой могут запускаться приложения. Но у него нет полноценной ОС. Обычно контейнер имеет чрезвычайно облегченную версию ОС, в которой присутствуют только основные компоненты, необходимые для ее функционирования. Если приложению, которое необходимо развернуть в контейнере, требуются какие-либо конкретные инструменты, драйверы или служебные программы, их можно добавить в контейнер до помещения приложения в него. Таким образом, размер контейнера может ненамного превышать размер приложения, работающего внутри него. Именно это упрощает репликацию, развертывание и перемещение контейнеров между физическими машинами.

Контейнеры и образы контейнеров

Контейнер Docker — это активный модуль, работающий в системе. Прежде чем контейнер можно будет собрать и запустить, необходимо создать его образ. Образ контейнера представляет все данные, которыми располагает контейнер. Но они находятся в неактивном состоянии. Фактически образ можно рассматривать как файл, который можно загружать и выгружать; хоть это и не совсем файл, но концептуально он на него похож. Когда образ создан, его можно поместить в реестр образов, что эквивалентно выгрузке файла в хранилище. Затем образ можно извлечь из реестра, что аналогично загрузке файла. После извлечения образа на его основе можно запустить контейнер.

Можно создавать приватные реестры образов Docker для размещения образов конкретных систем и/или организаций. По умолчанию платформа Docker подключена к общедоступному реестру Docker Hub. Любой хранящийся в нем образ обычно можно перенести на любой компьютер, на котором работает Docker. Docker Hub доступен по следующему URL-адресу: **https://hub.docker.com/**.

Базовые образы и уровни

Каждый образ контейнера Docker создается на основе базовых образов. Каждый базовый образ, вероятно, основан на собственных базовых образах. Единственное исключение — так называемый *scratch*-образ, который является пустым.

Базовый образ предоставляет набор зависимостей для многократного использования. Например, если мы хотим запустить приложение .NET внутри контейнера, то можем создать образ на основе стандартного базового образа .NET. Этот образ будет содержать все необходимые компоненты и инструменты SDK, которые позволят нам создать и запустить наше приложение.

Для построения образа Docker предназначен скрипт под названием `Dockerfile`. Dockerfile — это стандартное имя файла, содержащего все команды, необходимые для добавления в образ всех требуемых компонентов. Мы можем копировать файлы, создавать приложения, открывать определенные порты, загружать, устанавливать недостающие компоненты ОС и т. д.

Обычно при выполнении каждой команды из Dockerfile создается определенный уровень образа, который кешируется в локальном экземпляре Docker. Это намного ускоряет последующие сборки. Если последовательность команд не изменилась и результаты этих команд детерминированы, то при необходимости перестроить образ Docker не обязательно будет выполнять все эти команды снова. Вместо этого просто извлекается уровень образа, созданный в результате выполнения предыдущих команд, и выполняются только последующие команды, которые были добавлены или цель которых была изменена (например, в результате изменений кода создаваемого приложения).

Изоляция сети и сопоставление портов

По умолчанию контейнер Docker полностью изолирован от любых других приложений, работающих в той же среде. К нему можно получить доступ из внутренней сети Docker, но никакие другие компоненты не могут с ним взаимодействовать. Однако приложения внутри контейнера могут обмениваться данными и отправлять запросы во внешний мир. Таким образом, по умолчанию приложение внутри контейнера Docker может выступать в роли клиента, но не сервера.

Чтобы приложение в контейнере Docker могло работать в качестве сервера, следует осуществить сопоставление портов. Для этого некоторый конкретный порт помечается как открытый определенной командой внутри Dockerfile во время сборки образа. Затем в команду, запускающую контейнер, добавляется сопоставление портов. Открытый порт контейнера сопоставляется с определенным портом на хосте. Это делает приложение доступным для любой сети, к которой подключен хост, включая Интернет.

Изоляция файловой системы и монтирование каталогов хоста

По умолчанию файловая система в контейнере Docker полностью изолирована. Доступ к ней можно получить с помощью Docker и путем выполнения определенных команд Docker. Но никакие другие компоненты на хосте не могут получить доступ к файлам внутри контейнера.

Однако иногда возникает потребность обеспечить легкий обмен файлами между контейнером Docker и его хостом. Таким образом удобно, например, открывать

доступ к файлам журналов. Если говорить о среде разработки, то было бы неплохо видеть, что происходит внутри контейнера в реальном времени. Docker имеет возможность сопоставлять части внутренней файловой системы контейнеров с определенными папками в ОС хоста. Существует множество способов реализовать это, но, пожалуй, самый популярный — монтирование каталогов хоста (bind-mount).

Монтирование каталогов хоста — это процесс создания двустороннего сопоставления между определенным файлом или папкой внутри контейнера Docker и эквивалентным файлом или папкой на хосте. При внесении изменений в отображаемое содержимое на хосте те же изменения происходят и внутри контейнера. То же самое верно и в обратном направлении.

Теперь перейдем к установке Docker. Этот процесс зависит от используемой операционной системы.

Установка Docker в Linux

Изначально технология Docker была разработана только для Linux, а затем стали доступны версии для Windows и Mac OS. Существует несколько способов установки Docker. Это можно сделать как через интерфейс командной строки, так и с помощью полностью управляемого приложения Docker Desktop. Также имеются некоторые различия в установке в зависимости от используемого дистрибутива Linux.

Поскольку способы установки Docker на Linux зависят от ваших личных предпочтений и имеющегося дистрибутива, лучше всего следовать официальной инструкции по установке для Linux, которую можно найти по следующей ссылке:
https://docs.docker.com/desktop/install/linux-install/.

Установка Docker на Mac

Для Mac также есть несколько вариантов установки Docker в зависимости от того, какие чипы содержит конкретный компьютер: Intel или Apple. Актуальную инструкцию по установке можно найти здесь:
https://docs.docker.com/desktop/install/mac-install/.

Установка Docker в Windows

Для установки Docker Desktop в Windows требуется технология Hyper-V, которая доступна только в 64-разрядных версиях Windows 10 или 11 Pro, Enterprise и Education. Поэтому единственный способ установить Docker на домашнюю версию Windows — применение виртуальной машины. Подробная инструкция по установке для Windows доступна по следующей ссылке:
https://docs.docker.com/desktop/install/windows-install/.

После установки Docker на компьютер для разработки можно приступить к рассмотрению интеграции этой технологии с .NET 7 SDK. Мы также рассмотрим некоторые основные команды Docker, как использующиеся внутри Dockerfile для создания образа, так и предназначенные для CLI.

Базовый образ Docker для .NET 7

Начиная с .NET 7, функциональные возможности Docker полностью интегрированы в SDK, поэтому самый простой способ создать образ Docker с вашим приложением внутри — это опубликовать его как образ Docker. Именно это мы и сделаем далее.

Нам подойдет .NET-приложение любого типа, но в качестве примера возьмем приложение ASP.NET Core Razor Pages. Создать такое приложение можно с помощью следующей команды:

```
dotnet new web -o BasicContainerApp
```

Затем нужно перейти в папку только что созданного проекта и выполнить следующую команду, чтобы добавить в проект временный пакет NuGet, необходимый для легкого создания образа Docker:

```
dotnet add package Microsoft.NET.Build.Containers
```

Далее выполним команду для создания образа Docker:

```
dotnet publish --os linux --arch x64 -c Release -p:PublishProfile=DefaultContainer
```

Параметр `--os` указывает, на какой операционной системе работает Docker. Это не то же самое, что операционная система хоста, т. к. Docker Desktop для Windows можно настроить на запуск контейнеров Linux. Параметр `--arch` указывает на архитектуру процессора и может иметь значение `x64` либо `arm64`. Этот параметр относится к архитектуре центрального процессора хоста. Параметр `-c` указывает на то, что публикуемое приложение имеет конфигурацию Release. Параметр `-p` дает возможность передать компилятору .NET любые другие параметры. В нашем случае мы настраиваем профиль публикации, который позволяет нам автоматически создать образ.

По умолчанию образ Docker, создаваемый этим процессом, получает имя, соответствующее строчной версии названия проекта в сочетании с версией программного обеспечения 1.0.0. Поэтому, если предположить, что наш первоначальный проект назывался `BasicContainerApp`, полное имя итогового образа будет следующим: `basiccontainerapp:1.0.0`.

Чтобы запустить контейнер на основе этого образа, выполните следующую команду:

```
docker run -it --rm -p 5010:80 basiccontainerapp:1.0.0
```

`docker run` — это команда, которая предназначена для запуска контейнера Docker на основе определенного образа. Флаг `-it` указывает на то, что мы запускаем его в интерактивном режиме, поэтому во время работы контейнера в консоль будет выводиться информация изнутри контейнера. Мы также можем задать флаг `-d`, если хотим запустить его в фоновом режиме. Флаг `--rm` указывает на то, что после остановки контейнер и все связанные с ним данные удаляются. Атрибут `-p` указывает на сопоставление портов. Первое число — это порт на хосте, второе — порт в контейнере, к которому он привязан. Чтобы сопоставление портов работало, порт в кон-

тейнере должен быть открытым. Порт 80 открыт по умолчанию, если мы используем именно этот способ создания контейнера. Наконец, мы указываем имя образа:

```
basiccontainerapp:1.0.0
```

После запуска контейнера в консоли начнут отображаться результаты работы приложения. Поскольку веб-приложение внутри контейнера прослушивает порт 80, который сопоставлен с портом 5010 на хосте, мы можем перейти к приложению в браузере, набрав следующий URL-адрес: **http://localhost:5010/**.

Но это не единственный способ добавить поддержку Docker в приложение .NET. Традиционный способ заключается в добавлении Dockerfile в приложение и использовании CLI Docker, а не CLI .NET для создания образа. Если у нас Windows с Visual Studio в качестве IDE, то добавлять Dockerfile вручную не потребуется. Мы можем добавить поддержку Docker при создании проекта приложения из графического интерфейса.

Создание приложения с поддержкой Docker

При создании нового .NET-приложения в Visual Studio мы можем выбрать опцию **Enable Docker** (Включить Docker). При этом нам также будет предложено выбрать **Docker OS**, которая может быть как Windows, так и Linux (рис. 11.1).

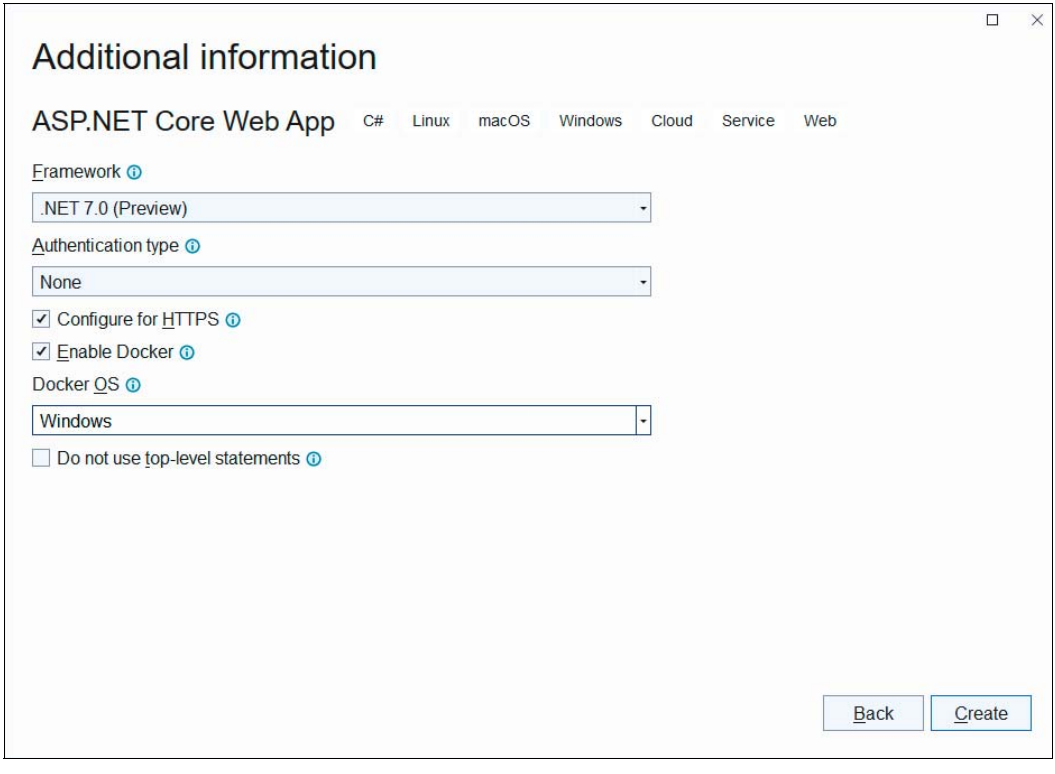


Рис. 11.1. Создание приложения с поддержкой Docker

После создания проекта приложения в папке проекта появится файл с именем Dockerfile и примерно таким содержимым:

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base WORKDIR /app
EXPOSE 80
EXPOSE 443
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src
COPY ["WindowsContainerExample/WindowsContainerExample.csproj", "WindowsContainerExample/"]
RUN dotnet restore "WindowsContainerExample/WindowsContainerExample.csproj"
COPY . .
WORKDIR "/src/WindowsContainerExample"
RUN dotnet build "WindowsContainerExample.csproj" -c Release -o /app/ build
FROM build AS publish
RUN dotnet publish "WindowsContainerExample.csproj" -c Release -o /app/ publish
/p:UseAppHost=false
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "WindowsContainerExample.dll"]
```

Содержимое этого файла будет одинаковым независимо от выбранной для Docker ОС. Отличия будут заключаться только в именах проектов и решений. В предыдущем примере мы предположили, что имя проекта — `WindowsContainerExample`.

В дополнение к этому в файл *.csproj будет добавлен следующий элемент:

```
<DockerDefaultTargetOS>Windows</DockerDefaultTargetOS>
```

Этот элемент не является строго необходимым. Образ Docker можно создать и без него. Но если мы решили задать этот элемент, то нужно настроить наш экземпляр Docker на работу с указанной здесь ОС. В противном случае Docker просто соберет образ для той ОС, которая установлена в качестве ОС локального экземпляра Docker.

Пользователи Visual Studio также могут добавить поддержку Docker в существующее приложение. Этот подход мы рассмотрим далее.

Добавление поддержки Docker в существующее приложение

Если в Visual Studio открыт проект .NET, в котором еще нет поддержки Docker, то можно щелкнуть правой кнопкой мыши на проекте и выбрать пункт **Docker Support** (Поддержка Docker), как показано на рис. 11.2.

Если вы не используете Visual Studio, то вам придется создавать Dockerfile вручную. Далее мы рассмотрим его структуру более подробно.

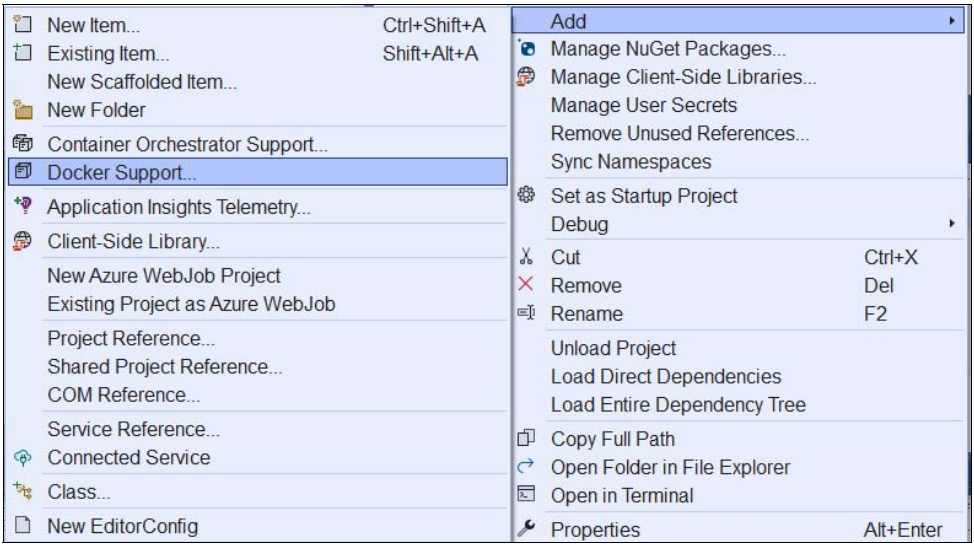


Рис. 11.2. Добавление поддержки Docker в существующее приложение

Структура файла Dockerfile

Предположим, что у нас есть проект приложения ASP.NET Core под названием `LinuxContainerExample`. Он имеет следующее содержимое, основанное на результатах работы Visual Studio:

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src
COPY ["LinuxContainerExample/LinuxContainerExample.csproj", "LinuxContainerExample/"]
RUN dotnet restore "LinuxContainerExample/LinuxContainerExample.csproj"
COPY . .
WORKDIR "/src/LinuxContainerExample"
RUN dotnet build "LinuxContainerExample.csproj" -c Release -o /app/build
FROM build AS publish
RUN dotnet publish "LinuxContainerExample.csproj" -c Release -o /app/ publish
/p:UseAppHost=false
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "LinuxContainerExample.dll"]
```

Рассмотрим его содержимое построчно. Первая строка выглядит следующим образом:

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
```

В ней мы указываем Docker использовать `mcr.microsoft.com/dotnet/aspnet:7.0` в качестве базового образа нашего контейнера. Это стандартный образ среды выполнения для ASP.NET Core 7. Образ среды выполнения содержит компоненты, необходимые только для выполнения, и не содержит SDK, поэтому с его помощью приложение можно запустить, но нельзя собрать. Далее мы будем использовать промежуточный образ. Следующие три строки:

```
WORKDIR /app
EXPOSE 80
EXPOSE 443
```

Здесь мы задаем каталог `app` в корне структуры папок контейнера в качестве рабочего каталога и открываем порты `80` и `443` для внешнего мира. Порт `80` — это порт HTTP по умолчанию, а `443` — порт HTTPS по умолчанию.

В следующих двух строках мы устанавливаем официальный образ .NET SDK в качестве основы для промежуточного образа и определяем папку `src` в корне этого образа в качестве рабочего каталога:

```
FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src
```

Затем мы копируем файл проекта в папку LinuxContainerExample в рабочем каталоге:

```
COPY ["LinuxContainerExample/LinuxContainerExample.csproj", "LinuxContainerExample/"]
```

Оригинальным путем для команды `COPY` является либо абсолютный путь на хосте, либо путь относительно того места, откуда была выдана команда `docker build`. В нашем случае мы предполагаем, что команда `docker build` выполняется из папки, расположенной непосредственно над папкой проекта.

Следующие две команды восстанавливают NuGet-зависимости из проекта внутри образа и копируют содержимое из текущей папки на хосте в рабочий каталог:

```
RUN dotnet restore "LinuxContainerExample/LinuxContainerExample.csproj"
COPY . .
```

Эта команда копирует в образ остальные файлы проекта. С помощью следующих двух команд устанавливается каталог проекта внутри образа в качестве рабочего каталога и компилируется приложение из проекта:

```
WORKDIR "/src/LinuxContainerExample"
RUN dotnet build "LinuxContainerExample.csproj" -c Release -o /app/build
```

В следующих строках создается еще один промежуточный образ из образа сборки, а также исполняемое приложение для публикации:

```
FROM build AS publish
RUN dotnet publish "LinuxContainerExample.csproj" -c Release -o /app/ publish
/p:UseAppHost=false
```

Наконец, мы копируем опубликованное приложение из образа публикации в конечный образ и запускаем приложение через CLI с помощью команды `dotnet`:

```
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "LinuxContainerExample.dll"]
```

В конечном образе содержится только опубликованное приложение. В итоге у нас есть только среда выполнения .NET и приложение, запущенное внутри нее. А промежуточные образы с .NET SDK и различными артефактами сборки были удалены.

Такова базовая структура файла Dockerfile. Теперь рассмотрим, как создать с его помощью образ и запустить из него контейнер.

Создание и запуск контейнера Docker

Чтобы создать образ контейнера Docker, выполните команду, аналогичную следующей:

```
docker build -t basicapp:1.0.0
```

`docker build` — это базовая команда, которая предназначена для создания образов. Параметр `-t` представляет тег образа. Можно выбрать любое название, если оно не занято уже существующим образом, созданным кем-то другим. Часть после двоеточия может быть любой, но обычно здесь указывают номер версии. Последняя часть команды задает рабочий каталог на хосте. Если мы используем точку в конце выражения команды, как в предыдущем примере, то каталог, из которого мы выполняем команду, является рабочим каталогом. Это означает, что текущий каталог — это место, где команда будет искать все необходимые файлы.

В предыдущем примере предполагается, что Dockerfile находится в каталоге, из которого мы выполняем команду. Однако мы можем разместить Dockerfile в любом месте и указать для него абсолютный или относительный путь, добавив параметр `-f` или `--file`, за которым следует путь к файлу.

В предыдущем примере мы создали образ с полным именем тега `basicapp:1.0.0`. Мы можем запустить контейнер из этого образа, выполнив команду, аналогичную следующей:

```
docker run -it --rm -p 5010:80 basicapp:1.0.0
```

На этом обзор создания и запуска одного контейнера Docker завершен. В реальном мире нам часто приходится запускать несколько контейнеров в качестве частей одного распределенного приложения. Также необходимо иметь возможность перезапускать контейнеры при возникновении проблем и увеличивать или уменьшать их число по мере необходимости. Для этого применяется оркестрация.

Оркестрация приложений с помощью Docker Swarm

Как следует из названия, процесс оркестрации аналогичен управлению оркестром как единым целым. Оркестр объединяет много людей, играющих на самых разных

музыкальных инструментах. Но все они действуют слаженно, воспроизводя заранее подготовленное музыкальное произведение.

Процесс оркестрации работы распределенного приложения основан на том же принципе. Приложение состоит из отдельных микросервисов. Но все они работают как единое распределенное приложение. Оркестрация обеспечивает бесперебойную работу приложения в целом. Микросервисы, которые были по тем или иным причинам остановлены, перезапускаются. Если какой-либо микросервис испытывает повышенную нагрузку, его масштабируют. Если аппаратные средства на конкретном компьютере выходят из строя, развернутые на нем сервисы удаляются и создаются заново на другом компьютере того же кластера.

Существует множество способов оркестрации контейнеров Docker. Система оркестрации промышленного уровня, поставляемая вместе с Docker, называется Docker Swarm. Мы вскоре познакомимся с ней подробнее. Но сначала рассмотрим более элементарные функции, похожие на оркестрацию, которые можно реализовать с помощью приложения Docker Compose.

Базовая оркестрация помощью Docker Compose

Docker Compose — это инструмент, включенный в состав Docker. Он использует файлы YAML для совместного запуска нескольких контейнеров Docker. Поддерживаются и другие возможности оркестрации, например возможность запускать контейнеры в определенном порядке и предварительно настраивать отдельные контейнеры на перезапуск в случае их остановки из-за ошибки. Но этим возможности Docker Compose и ограничиваются.

Чтобы задействовать Docker Compose, необходимо создать YAML-файл. По принятым соглашениям этому файлу дают имя docker-compose.yml. Но мы можем присвоить ему любое имя. Если мы сделаем это, нам придется явно указывать его в командах Docker Compose.

Если вы работаете на компьютере под управлением Windows с Visual Studio в качестве IDE, то можете добавить в свое приложение поддержку базовых функций оркестрации Docker Compose, просто щелкнув правой кнопкой мыши на своем проекте и выбрав пункты **Add | Container Orchestration Support** (Добавить | Поддержка оркестратора контейнеров), как показано на рис. 11.3.

При этом в корне решения создается несколько файлов. Основной файл, необходимый Docker Compose, — это docker-compose.yml с содержимым, подобным следующему:

```
version: '3.4'
services:
    linuxcontainerexample:
    image: ${DOCKER_REGISTRY-}linuxcontainerexample
    build:
        context: .
        dockerfile: LinuxContainerExample/Dockerfile
```

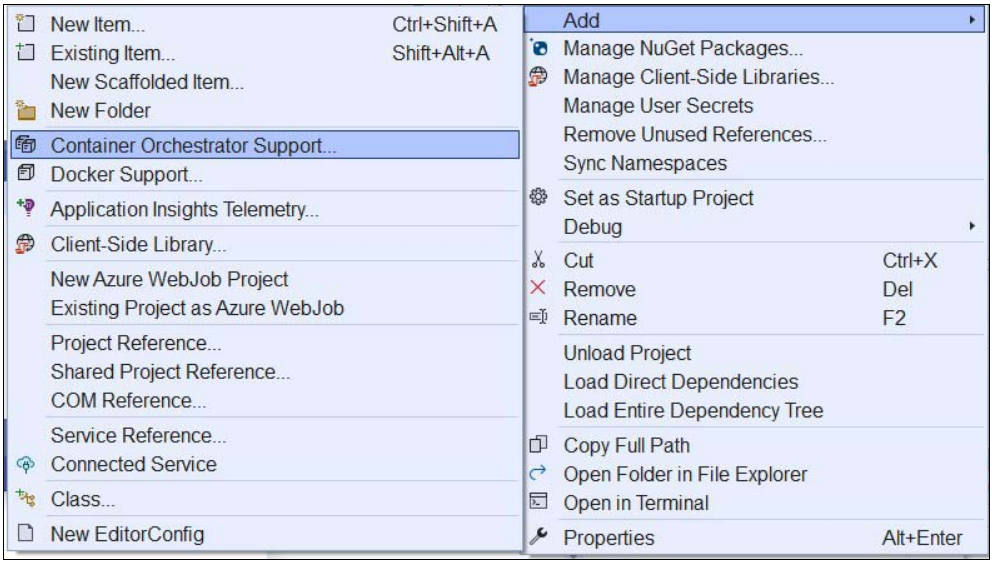


Рис. 11.3. Добавление оркестрации Docker Compose в приложение .NET

Этот файл может содержать несколько сервисов, но в предыдущем примере он содержит только один сервис. У каждого сервиса в разделе `services` есть собственное имя. В предыдущем примере нашему сервису дано имя `linuxcontainerexample`.

Затем мы даем указания по созданию образа для контейнера или извлечению существующего образа. В нашем примере мы создаем новый образ из Dockerfile, который расположен по определенному относительному пути. В поле `image` указывается имя образа. В имени образа также можно указать значения переменных окружения, например `${DOCKER_REGISTRY-}`, как показано в предыдущем примере.

Теперь, чтобы запустить распределенное приложение и создать все необходимые контейнеры, мы можем выполнить следующую команду в каталоге, где находится наш файл YAML:

```
docker-compose up -d
```

В этом примере мы запускаем все контейнеры в фоновом режиме, указывая флаг `-d`. Но можно запускать их и в интерактивном режиме, опустив этот флаг. Чтобы остановить контейнеры, можно выполнить следующую команду:

```
docker-compose down
```

На этом обзор базовых средств оркестрации в Docker Compose завершен. Этот механизм оркестрации очень ограничен и подходит только для самых простых приложений или временных разработок. Теперь мы рассмотрим более продвинутый механизм оркестрации Docker Swarm.

Запуск Docker Swarm

Как и Docker Compose, инструмент Docker Swarm поставляется вместе с Docker. Но в качестве механизма оркестрации он гораздо более продвинутый по сравнению с Docker Compose. Этот инструмент подходит для развертывания масштабируемых распределенных приложений промышленного уровня, поскольку способен работать на нескольких компьютерах, автоматически распределять контейнеры между компьютерами, обнаруживать аппаратные сбои и реагировать на них, а также масштабировать отдельные микросервисы путем их репликации.

Docker Swarm — это кластер скоординированных узлов, которые могут быть представлены как физическими, так и виртуальными машинами. Чтобы запустить Docker Swarm, нужно выполнить следующую команду, которая создаст главную виртуальную машину (VM), выполняющую роль оркестратора:

```
docker swarm init
```

В выводе этой команды будет показана команда, которую нужно выполнить, чтобы присоединиться к кластеру Swarm. Она будет уникальной для конкретного экземпляра Swarm, поскольку необходимо сгенерировать уникальный токен присоединения. Команда будет выглядеть примерно так:

```
docker swarm join --token SWMTKN-1-1bj2ltc44eb9c45zq7482vt6f1ryx3ghrwe2z xqlcakht7ui5x-
csqtvblydg7h9ge5mw0qixuz0 192.168.65.3:2377
```

Если вам когда-нибудь понадобится вспомнить эту команду, выполните следующую команду на компьютере, где установлен Swarm:

```
docker swarm join-token worker
```

После выполнения команды `docker swarm join` можно начать добавлять в Swarm сервисы. Мы также можем определить, сколько реплик каждого сервиса требуется добавить. Например, приведенная далее команда добавляет три реплики контейнера на основе образа `basiccontainerapp:1.0.0` и дает этому сервису имя `basic-service`:

```
docker service create --replicas 3 --name basic-service basiccontainerapp:1.0.0
```

Если требуется узнать, какие сервисы присутствуют в нашем экземпляре Swarm, можно выполнить команду

```
docker service ls
```

результат выполнения которой должен быть похож на следующий:

```
ID  NAME  MODE  REPLICAS  IMAGE  PORTS
kgd33x72ae31 basic-servicereplicated  3/3  basiccontainerapp:1.0.0
```

Также можно удалить все экземпляры определенного сервиса, выполнив команду:

```
docker service rm basic-service
```

Если указанный сервис был единственным сервисом в Swarm, результат команды `docker service ls` должен выглядеть следующим образом:

```
ID  NAME  MODE  REPLICAS  IMAGE  PORTS
```

Если требуется удалить из Swarm определенный узел, можно выполнить команду:

```
docker swarm leave
```

Если требуется удалить весь кластер Swarm, следует выполнить эту команду на главном узле и добавить к ней флаг `--force`.

Docker Swarm не является единственным способом организации работы контейнеров Docker в эксплуатационной среде. Другой популярной платформой является Kubernetes — ее мы и рассмотрим далее.

Оркестрация приложений с помощью Kubernetes

Kubernetes — это популярная платформа для оркестрации контейнеров, изначально созданная компанией Google. Она лучше подходит для применения в крупномасштабных распределенных приложениях, чем Docker Swarm. Это связано с тем, что она имеет множество дополнительных функций, которые облегчают процесс оркестрации. Например, группы связанных контейнеров могут быть организованы в так называемые поды. Кроме того, диаграммы Helm позволяют быстро построить распределенное приложение по заранее заданному шаблону.

Kubernetes можно установить на ваш рабочий компьютер. Процесс установки будет зависеть от имеющейся ОС и архитектуры процессора. В обоих случаях нам потребуется установить серверный компонент и клиент командной строки для управления им. Последний может быть представлен инструментом под названием kubectl.

Установка Kubernetes в Linux

На компьютере с Linux можно установить инструмент `minikube`, который представляет собой версию сервера Kubernetes для разработки. Если компьютер оснащен процессором с архитектурой *x*64, команды установки будут следующими:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/ minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Для компьютеров с процессорами ARM64 команды установки будут выглядеть так:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/ minikube-linux-arm64
sudo install minikube-linux-arm64 /usr/local/bin/minikube
```

Затем следует установить инструмент `kubectl`, который обеспечит взаимодействие с кластером. На компьютере с архитектурой *x*64 команда загрузки будет выглядеть следующим образом:

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/
                                        release/stable.txt)/bin/linux/arm64/kubectl"
```

В случае процессора AMD фрагмент `arm64` следует заменить на `amd64`.

Затем следует установить этот инструмент, выполнив команду:

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Установка Kubernetes на Mac

В Mac OS также можно использовать `minikube` в качестве сервера Kubernetes в среде разработки. Для компьютера Mac с процессором Intel команды загрузки и установки будут следующими:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-amd64
sudo install minikube-darwin-amd64 /usr/local/bin/minikube
```

Для устройства с чипом Apple команды будут такими:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/ minikube-darwin-arm64
sudo install minikube-darwin-arm64 /usr/local/bin/minikube
```

Затем мы можем загрузить инструмент `kubectl`, выполнив следующую команду, где фрагмент `amd64` в пути указывают для устройств на чипе Intel, а `arm64` — для устройств на чипе Apple.

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/
release/stable.txt)/bin/darwin/amd64/kubectl"
```

Далее нам нужно выполнить очередные команды, чтобы сделать `kubectl` доступным в терминале:

```
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl sudo chown root: /usr/local/bin/kubectl Installing
Kubernetes on Windows
```

Самый простой способ установить Kubernetes в ОС Windows — с помощью Docker Desktop. Для этого щелкните правой кнопкой мыши на значке **Docker Desktop** в области панели задач, выберите пункт **Settings** (Настройки), перейдите в раздел **Kubernetes** и установите флажок **Enable Kubernetes** (Включить Kubernetes), как показано на рис. 11.4.

После сохранения настроек будут установлены все необходимые компоненты Kubernetes, включая kubectl.

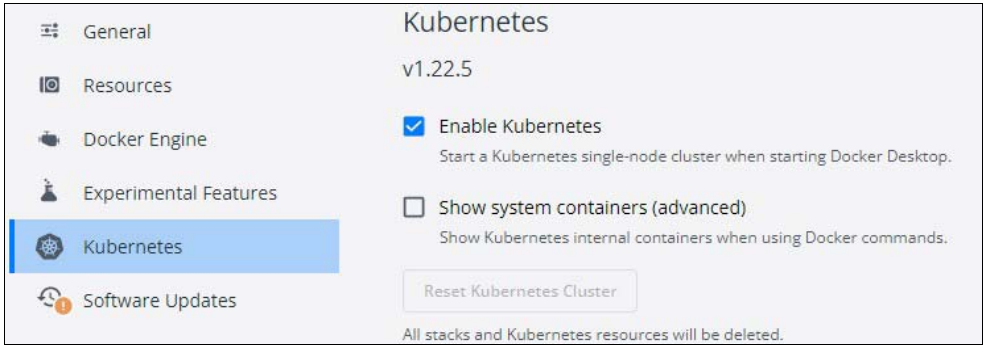


Рис. 11.4. Включение Kubernetes в Docker Desktop в ОС Windows

Добавление сервисов в кластер Kubernetes

После установки локального кластера Kubernetes мы можем проверить его, выполнив в терминале следующую команду:

```
kubectl --help
```

Если ошибок нет, можно приступить к развертыванию контейнеров в нашем кластере Kubernetes. Во-первых, мы можем развернуть определенный контейнер Docker, выполнив команду `kubectl create deployment`. В следующем примере развертывается сервис с именем `basic-service` из образа Docker с именем `basiccontainerapp:1.0.0`.

```
kubectl create deployment basic-service --image basiccontainerapp:1.0.0
```

Далее мы можем выполнить команду, чтобы открыть порт `80` сервиса:

```
kubectl expose deployment basic-service --type=NodePort --port=80
```

Мы можем просмотреть информацию о любом сервисе, выполнив команду:

```
kubectl get services basic-service
```

результат выполнения которой должен выглядеть примерно так:

```
NAME              TYPE          CLUSTER-IP          EXTERNAL-IP       PORT(S)
AGE
basic-service     NodePort      10.109.141.219      <none>            80:31546/TCP
50s
```

Чтобы сделать сервис доступным извне, необходимо перенаправить его внутренний порт на порт хоста. Мы можем перенаправить ранее открытый порт `80` сервиса в порт `7080` хоста, выполнив команду:

```
kubectl port-forward service/basic-service 7080:80
```

Теперь можно проверить полученную структуру, перейдя в браузере по адресу: **http://localhost:7080**.

Если требуется удалить кластер Kubernetes, то в Windows можно просто снять флажок **Enable Kubernetes** (Включить Kubernetes) в окне **Docker Desktop** и сохранить настройки. Если мы работаем с MacOS или Linux, то остановить все сервисы в кластере можно, выполнив команду:

```
minikube stop
```

А чтобы затем удалить все сервисы — следующую команду:

```
minikube delete --all
```

На этом обзор создания и оркестрации работы контейнеров Docker завершен. Подведем итоги тому, что мы узнали.

Заключение

В этой главе мы рассмотрели процесс контейнеризации приложения .NET. Мы узнали, что основной инструмент контейнеризации — Docker и что поддержка этой технологии встроена в .NET 7 SDK.

Мы также узнали, как установить Docker, написать скрипт Dockerfile и как выполнять сборку .NET-приложений в контейнеры Docker. Мы описали выполнение этого процесса с помощью различных инструментов графического интерфейса и интерфейса командной строки.

Мы также рассмотрели процесс оркестрации распределенного приложения, состоящего из контейнеризированных микросервисов. Мы познакомились с тремя стандартными инструментами для решения этой задачи: Docker Compose, обеспечивающий совместный запуск контейнеров, Docker Swarm, позволяющий масштабировать отдельные сервисы, и Kubernetes, который предлагает множество расширенных функций оркестрации. Мы рассмотрели запуск каждого из этих инструментов в среде разработки.

Эта глава завершает книгу об основных возможностях C# 11 и .NET 7.

Что следует помнить

- Docker — это технология, позволяющая развертывать приложения в контейнерах.
- Каждый контейнер включает в себя приложение и наиболее важные зависимости от ОС.
- Приложение, находящееся внутри контейнера, обычно недоступно извне, если только не было прописано сопоставление портов и не была выполнена привязка внутренних папок к определенным папкам на хосте.
- Контейнеризация особенно полезна для распределенных приложений, оркестрация которых может осуществляться с помощью Docker Swarm или Kubernetes.

Контрольные вопросы

1. **В чем разница между контейнером Docker и образом Docker?**

   а) Это взаимозаменяемые понятия;

   б) Контейнер — это хранилище образов Docker;

   в) Образ представляет собой данные контейнера в пассивном состоянии, а контейнер — это работающее приложение;

   г) Контейнер представляет собой данные образа в пассивном состоянии, а образ — это работающее приложение.

2. **Какую платформу можно использовать для оркестрации контейнеров Docker?**

а) Docker Swarm;

б) Kubernetes;

в) Любую из вышеперечисленных;

г) Cron.

3. **Каковы преимущества контейнеризации?**

а) Согласованное поведение в любых средах;

б) Изоляция приложения от любых потенциальных источников непреднамеренных побочных воздействий;

в) Легкость развертывания и оркестрации приложений;

г) Все вышеперечисленные.

4. **Каково главное преимущество использования многоступенчатого процесса создания образа в Docker?**

а) Отделение компонентов среды выполнения от SDK и зависимостей, требуемых для сборки;

б) Улучшение читабельности;

в) Соблюдение общих стандартов;

г) Это не дает ощутимых преимуществ.

Ответы

1. в.
2. в.
3. г.
4. а.

Основные термины

- **Docker** — система, позволяющая упаковывать приложения в изолированные среды, называемые контейнерами.
- **Образ контейнера Docker** — предварительно собранная часть программного обеспечения, содержащая само приложение и его зависимости от ОС, которая может быть запущена как контейнер.
- **Контейнер Docker** — исполняемый файл, включающий в себя главное приложение и основные компоненты ОС, которые позволяют приложению функционировать.
- **Реестр образов Docker** — система, хранящая образы контейнеров Docker, которые могут быть извлечены из нее.
- **Dockerfile** — файл, определяющий порядок создания образа Docker.

- **Монтирование каталогов хоста** — процесс сопоставления файлов и папок внутри контейнера Docker с файлами и папками на хосте.
- **Оркестрация** — процесс координации работы компонентов распределенного приложения для обеспечения его функционирования, как единого целого.
- **Docker Swarm** — группа физических или виртуальных машин, которые работают как единый кластер и могут координировать работу набора контейнеров Docker.
- **Kubernetes** — многофункциональная система оркестрации контейнеров, подходящая для крупномасштабного развертывания распределенных приложений промышленного уровня.

Предметный указатель

.

.NET 7 SDK
◊ установка 25

A

Ahead of Time, AOT 64
Ahead-of-Time, AoT 183

B

Blazor 97, 172
Blazor Server 173
◊ настройка 189
Blazor WebAssembly 178

C

CLI 63
Command Line Interface, CLI 25
Community edition 28
Create-Read-Update-Delete, CRUD 168

D

Docker
◊ установка 269
Docker Compose 276
Docker Swarm 278

E

Entity Framework 7 117
Entity Framework Core 111

G

Graphic User Interface, GUI 26
gRPC 217

I

IDE 104
◊ JetBrains Rider 28, 30
◊ Roslyn-совместимая 76
◊ Visual Studio 2022 28
◊ Visual Studio 2022 for Mac 29
Integrated Development Environment, IDE 26

J

JSON 68, 204, 224
◊ полиморфизм 71

K

Kubernetes 279

M

MAUI 88
MessagePack 204
Microsoft.Extensions 84
Minimal API 149
ML.NET 240
Model-View-Controller, MVC 155
Model-View-Update (MVU) 101
Model-View-ViewModel, MVVM 99
Multi-platform UI 88

N

Node.js 149
Node.js Package Manager, NPM 204

O

Object-Relational Mapper, ORM 111
Object-Relational Mapping, ORM 112

R

Razor Components 174
React 101
RegEx 75
Remote Procedure Calls, RPC 216
RPC 221

S

scratch-образ 267
Server-Sent Events, SSE 198
SignalR 197
Software Development Kit, SDK 24
SQL 113
Swagger 148, 224
Swarm 278

T

Table-Per-Hierarchy, TPH 133
Table-Per-Type, TPT 133
TAR 78
Transport Layer Security, TLS 228

V

Visual Studio 2022 118
Visual Studio Code
◊ установка 26

W

WASM 178

X

Xamarin.Forms 89
XCode 107

А

Алгоритм
◊ GZIP 79
◊ TAR 78
Атрибут 50
◊ [Inject] 180
◊ [Parameter] 182
◊ RegexGenerator 76
◊ обобщенный 51

Б

База данных
◊ контекст 122
◊ реляционная 112
◊ таблицы 113

В

Веб-сервис 144
Виртуальная машина 266

Д

Диспетчер пакетов 65
Длительный опрос 199

Е

Единицы времени в .NET 66

И

Интерфейс 37
◊ MAUI 88
◊ командной строки 63
Искусственный интеллект 239

К

Класс
◊ GreeterService 220
◊ Hub 199
◊ NavigationManager 188
◊ Stream 73
◊ System.Diagnostics.Activity 80
◊ UpDownCounter 83
Классификация
◊ двоичная 246

◊ изображений 253
◊ многоклассовая 248, 253
Кластеризация 254
Ключ
◊ защищенный 132
◊ первичный 115, 131
◊ перечисления 233
Компиляция предварительная 183
Контейнер Docker 267
Контейнеризация 265

М

Машинное обучение 239
◊ модель 241
◊ показатели 256
◊ ранг 255
◊ типы 242
Метод
◊ AddEndpointFilter 154
◊ EnumerateMatches 76
◊ invoke 209
◊ MapHub 202
◊ ReadExactly 73
◊ SaveChanges 125
◊ SendAsync 212
Методы переопределяемые 191
Механизмы взаимодействия 197
Микросервис 265
Модификаторы доступа 44

Н

Необрабатываемый строковый литерал 56
Нормализация 115

О

Обнаружение отклонений 254
Обучение
◊ без учителя 250, 254
◊ с учителем 249, 253
Объект BLOB 153
Оркестрация 275

П

Паттерн
◊ MVC 119, 155
◊ MVU 101
◊ MVVM 99
◊ ReactiveUI 101
Переменная 39
Перехватчики 136
Подход
◊ сначала база данных 129
◊ сначала код 124
Полиморфизм 38, 71
Приложение
◊ MAUI 102
▫ для Mac OS 108
◊ NET 7 31
◊ Razor Pages 167
◊ без пользовательского интерфейса 144
◊ консольное 32
◊ мобильное 105
◊ нативное 65
◊ реального времени 196
◊ с поддержкой Docker 271
Принцип тонкого контроллера 145
Прогнозирование 251
Проект 31
Протокол
◊ gRPC, типы вызовов 221
◊ HTTPS 228
◊ Protobuf 218
◊ REST 224
◊ WebSocket 198

Р

Регрессия 249
Редактор кода
◊ установка 26
Решение 31

Т

Технология
◊ Blazor 97
◊ gRPC 217
Тип данных
◊ ByteString 222
◊ gRPC 232
◊ span 54
◊ struct 49
◊ встроенный 42
▫ Protobuf 222
◊ десятичный 120
◊ общеизвестный 235
Типизированные результаты 151

Ф

Файл
◊ *.proto 219
◊ App.razor 190
◊ appsettings.json 142
◊ AppShell.xaml 92
◊ Counter.razor 174
◊ Customer.cs 35
◊ Dockerfile 273
◊ EmployeesController.cs 158
◊ http.proto 224
◊ Index.cshtml 161
◊ MessageHub.cs 199
◊ nullable_greet.proto 235
◊ Program.cs 32, 122, 143
◊ RAZOR 97
◊ TarApi.cs 78
◊ WeatherForecastController.cs 145
◊ проекта 32
◊ формата XAML 92
◊ формата YAML 276
Функция
◊ TryParse 147
◊ сопоставления последовательности с образцом 53

Х

Хаб 199

Ш

Шаблоны T4 131
Шифрование 76

Э

Элементы
◊ XAML 94
◊ потока управления 43

Я

Язык
◊ GraphQL 216
◊ Python 239
◊ структурированных запросов 113