

# .NET 8 инструменты и навыки

Лучшие практики и паттерны проектирования,  
отладки и тестирования



Марк Дж. Прайс

**packt**

# Tools and Skills for .NET 8

Get the career you want with good practices and patterns  
to design, debug, and test your solutions

**Mark J. Price**



# .NET 8: инструменты и навыки

Лучшие практики и паттерны проектирования, отладки и тестирования

**Марк Дж. Прайс**

*Марк Прайс*

**.NET 8: инструменты и навыки.**  
**Лучшие практики и паттерны проектирования,**  
**отладки и тестирования**

*Перевел с английского А. Ларин*

ББК 32.973.2-018

УДК 004.42

**Прайс Марк**

П68 .NET 8: инструменты и навыки. Лучшие практики и паттерны проектирования, отладки и тестирования. — Астана: «Сprint Бук», 2025. — 800 с.: ил.

ISBN 978-601-08-5251-8

Раскройте богатство возможностей разработки .NET-приложений с новой книгой Марка Прайса! Здесь вы узнаете, как управлять исходным кодом и проектами с помощью Git, обеспечивая при этом контроль версий. Откроете для себя передовые методы отладки и стратегии выявления и устранения проблем, а также увидите, как правильно документировать код, API и сервисы, чтобы повысить удобство обслуживания проекта. Погрузитесь в мир криптографии и освойте приемы обеспечения конфиденциальности и целостности на протяжении всего цикла разработки.

Вы сможете расширить свои навыки, познакомившись с такими продвинутыми темами, как создание интеллектуальных приложений с поддержкой пользовательских чат-сервисов на основе LLM, внедрение зависимостей, оптимизация производительности с помощью тестирования и запуск сервисов в контейнерах Docker. В книге описаны возможности разработки облачных сервисов с применением .NET Aspire и преимущества современных облачных платформ. А знакомство с особенностями передовых архитектур программного обеспечения позволит вам создавать надежные и масштабируемые приложения, удобные в обслуживании.

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

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

© Packt Publishing 2024.

First published in the English language under the title 'Tools and Skills for .NET 8 – (9781837635207)'

ISBN 978-601-08-5251-8

© Перевод на русский язык Сprint Бук, 2025

© Издание на русском языке, оформление Сprint Бук, 2025

Изготовлено в России. Изготовитель: ТОО «Сprint Бук». Место нахождения и фактический адрес: 010000 Казахстан, город Астана, район Алматы, проспект Рахымжан Кошкараев, дом 10/1, н.п. 18

Дата изготовления: 09.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 12.08.25. Формат 70×100/16. Бумага офсетная. Усл. п. л. 64,500. Тираж 500. Заказ 0000.

# Краткое содержание

Об авторе.....	20
О научных редакторах.....	21
Предисловие .....	22
От издательства.....	27
<b>Глава 1.</b> Обзор инструментов и навыков для работы с .NET .....	28
<b>Глава 2.</b> Эффективная работа с инструментами редактора кода .....	69
<b>Глава 3.</b> Управление исходным кодом с помощью Git .....	123
<b>Глава 4.</b> Отладка и устранение проблем с памятью.....	170
<b>Глава 5.</b> Ведение журналов, трассировка и сбор метрик для обеспечения наблюдаемости.....	210
<b>Глава 6.</b> Документирование кода, API и сервисов .....	239
<b>Глава 7.</b> Наблюдение и динамическое изменение выполнения кода .....	282
<b>Глава 8.</b> Защита данных и приложений с помощью криптографии .....	309
<b>Глава 9.</b> Создание чат-сервиса на основе LLM.....	342
<b>Глава 10.</b> Внедрение зависимостей, контейнеры и время жизни сервисов.....	379
<b>Глава 11.</b> Модульное и mock-тестирование.....	405
<b>Глава 12.</b> Интеграционное тестирование и проверка безопасности .....	454
<b>Глава 13.</b> Оценка производительности, нагрузочное и стресс-тестирование .....	482
<b>Глава 14.</b> Функциональное и комплексное тестирование сайтов и сервисов .....	520
<b>Глава 15.</b> Контейнеризация с помощью Docker.....	552
<b>Глава 16.</b> Облачная разработка с помощью .NET Aspire .....	591
<b>Глава 17.</b> Паттерны и принципы проектирования.....	637
<b>Глава 18.</b> Основы архитектуры ПО и решений.....	679
<b>Глава 19.</b> Ваша карьера, командная работа и собеседования .....	708
<b>Глава 20.</b> Эпилог .....	762
<b>Приложение.</b> Ответы к разделам «Проверьте свои знания» .....	765

# Оглавление

Об авторе.....	20
О научных редакторах.....	21
Предисловие .....	22
Примеры кода.....	22
Структура книги.....	22
Введение .....	23
Инструменты .....	23
Навыки.....	23
Тестирование .....	24
Построение карьеры в сфере разработки.....	25
Что вам понадобится для работы с этой книгой .....	25
Полноцветные иллюстрации из этой книги .....	25
Условные обозначения.....	26
От издательства .....	27
<b>Глава 1.</b> Обзор инструментов и навыков для работы с .NET .....	28
Знакомство с этой книгой и ее содержимым.....	28
Сопутствующие книги для продолжения обучения.....	29
Целевая аудитория .....	29
Инструменты .....	31
Навыки.....	32
Тестирование .....	32
Построение карьеры в сфере разработки.....	33
Настройка среды разработки .....	34
Выбор подходящего ПО .....	34
Кросс-платформенное развертывание.....	37
Загрузка и установка Visual Studio 2022 .....	37
Visual Studio Code.....	39
Загрузка и установка Rider .....	43
Другие инструменты JetBrains .....	44
ИИ-инструменты в браузере Chrome .....	45

GitHub-репозиторий для этой книги .....	45
При возникновении вопросов по книге .....	45
Связь с автором.....	46
Загрузка кода проектов из репозитория GitHub .....	46
Источники справочной информации.....	46
Техническая документация на сайте Microsoft Learn .....	47
Получение справки с помощью команды dotnet.....	47
ИИ-инструменты, такие как ChatGPT и GitHub Copilot.....	47
Получение помощи в Discord и других чат-форумах.....	51
Настройка базы данных и проектов в рамках книги .....	53
Использование примера реляционной базы данных.....	53
Настройка SQL Server и базы данных Northwind .....	55
Создание библиотеки классов для моделей сущностей с помощью SQL Server .....	57
Создание библиотеки классов для контекста данных с помощью SQL Server .....	60
Создание тестового проекта для проверки интеграции библиотек классов .....	62
Тестирование .....	64
Применение .NET 9 в контексте данной книги .....	64
Практика и исследование.....	66
Упражнение 1.1. Дополнительные онлайн-материалы .....	67
Упражнение 1.2. Практические задания.....	67
Упражнение 1.3. Проверьте свои знания .....	67
Упражнение 1.4. Дополнительная информация .....	67
Резюме.....	68
<b>Глава 2. Эффективная работа с инструментами редактора кода .....</b>	<b>69</b>
Общие инструменты и функции редакторов .....	69
Средства рефакторинга .....	69
Фрагменты кода.....	70
Настройка редактора кода .....	71
Интеллектуальные помощники в редакторах кода.....	73
Инструменты в Visual Studio 2022.....	74
Функционал рефакторинга .....	74
Фрагменты кода.....	81
Конфигурация редактора .....	88
Интеллектуальные помощники на базе ИИ: GitHub Copilot .....	90
Навигация в Visual Studio.....	93
Визуальные настройки редактора кода .....	94
Инструменты в Visual Studio Code .....	97
Функции рефакторинга .....	98
Фрагменты кода.....	98
Декомпиляция сборок .NET .....	100
Создание консольного приложения для декомпиляции.....	100
Декомпиляция с помощью расширения ILSpy для Visual Studio.....	102

Просмотр ссылок на исходный код в Visual Studio .....	105
Нет, предотвратить декомпиляцию техническими средствами невозможно .....	107
Снижение уровня C#-конструкций .....	108
Пользовательские шаблоны проектов и элементов .....	113
Создание проекта для шаблона.....	115
Тестирование шаблона проекта .....	120
Практика и исследование.....	120
Упражнение 2.1. Дополнительные онлайн-материалы .....	120
Упражнение 2.2. Практические задания.....	121
Упражнение 2.3. Проверьте свои знания .....	121
Упражнение 2.4. Дополнительная информация .....	122
Резюме.....	122
<b>Глава 3. Управление исходным кодом с помощью Git.....</b>	<b>123</b>
Внедрение систем управления исходным кодом .....	123
Функциональные возможности систем управления исходным кодом.....	123
Типы систем управления исходным кодом .....	124
Популярные системы управления исходным кодом .....	125
Знакомство с Git.....	125
Основные возможности Git.....	126
Почему Git сложно изучать.....	127
Роли в команде при работе с Git .....	128
Загрузка последней версии Git .....	128
Интеграция Git в Visual Studio .....	128
Настройка идентификационных данных Git .....	129
Настройка SSH-подписей .....	130
Настройка ветки по умолчанию.....	131
Как получить справку по Git .....	132
Основы работы с Git .....	132
Начало работы с репозиторием Git .....	133
Создание и добавление файлов в репозиторий Git (теоретически) .....	134
Отслеживание изменений в Git .....	135
Создание репозитория Git на практике .....	136
Создание проекта .....	137
Коммиты файлов.....	141
Отмена коммита .....	143
Очистка коммита.....	144
Сохранение .....	144
Игнорирование файлов.....	146
Просмотр репозиториев Git .....	149
Просмотр различий в файлах .....	149
Просмотр истории коммитов .....	152
Фильтрация вывода журнала .....	155

Управление удаленными репозиториями .....	156
Ветвление и слияние.....	159
Пример ветвления и слияния .....	160
Удаление и просмотр веток .....	166
Краткое описание распространенных команд Git .....	166
Практика и исследование.....	167
Упражнение 3.1. Дополнительные онлайн-материалы .....	167
Упражнение 3.2. Практические задания.....	168
Упражнение 3.3. Проверьте свои знания .....	168
Упражнение 3.4. Дополнительная информация .....	169
Резюме.....	169
<b>Глава 4. Отладка и устранение проблем с памятью.....</b>	<b>170</b>
Стратегии отладки.....	170
Обзор стратегий отладки.....	171
Анализ проблемы .....	172
Как начать отладку .....	174
Когда стоит остановиться.....	174
Интерактивная отладка в Visual Studio .....	175
Создание кода для изучения объектов во время выполнения .....	175
Установка точки останова и начало отладки.....	178
Навигация с помощью панели инструментов отладки .....	181
Панели отладки.....	182
Настройка отображения пользовательских типов в отладчике .....	183
Отладка тестовых проектов.....	186
Отладка с помощью GitHub Copilot Chat .....	187
Анализ стека и кучи памяти .....	190
Как ссылочные и типы значений хранятся в памяти .....	190
Работа с небезопасным кодом в C# .....	193
Что нужно знать об указателях в C#.....	194
Упаковывание значений .....	197
Сборка мусора .....	198
Управление ресурсами с помощью IDisposable.....	200
Инструменты и приемы для устранения проблем с памятью.....	201
Распространенные инструменты и навыки работы с памятью .....	202
Инструменты Visual Studio .....	203
Анализ использования памяти в Visual Studio.....	204
Практика и исследование.....	208
Упражнение 4.1. Дополнительные онлайн-материалы .....	208
Упражнение 4.2. Практические задания .....	208
Упражнение 4.3. Проверьте свои знания .....	208
Упражнение 4.4. Дополнительная информация .....	209
Резюме.....	209

---

## 10 Оглавление

<b>Глава 5.</b> Ведение журналов, трассировка и сбор метрик для обеспечения наблюдаемости.....	210
Журналирование и трассировка в .NET .....	210
Интерфейс ILogger в .NET.....	212
Как вести журнал с помощью ILogger .....	214
Создание веб-сервиса для ведения журнала .....	216
Тестирование базовой функциональности веб-сервиса.....	221
Мониторинг с использованием метрик в .NET .....	221
Метрики и оповещения: основные понятия .....	221
Реализация метрик .....	223
Просмотр метрик.....	228
Знакомство с OpenTelemetry .....	229
Поддерживаемые пакеты инструментов.....	230
Сбор телеметрии в проекте ASP.NET Core.....	231
Просмотр телеметрии .....	233
Практика и исследование.....	237
Упражнение 5.1. Дополнительные онлайн-материалы .....	237
Упражнение 5.2. Практические задания.....	237
Упражнение 5.3. Проверьте свои знания.....	237
Упражнение 5.4. Дополнительная информация .....	238
Резюме.....	238
<b>Глава 6.</b> Документирование кода, API и сервисов .....	239
Знакомство с документацией.....	239
Преимущества документации.....	239
Когда не стоит документировать .....	241
Документирование исходного кода.....	242
Когда следует документировать исходный код.....	242
Эффективные приемы комментирования кода.....	244
Документирование API в библиотеках классов.....	244
Документирование с помощью XML-комментариев .....	246
Создание документации с помощью DocFX .....	253
Добавление собственных страниц в документацию .....	261
Язык разметки Markdown .....	261
Документирование сервисов.....	265
Важные моменты на счет документирования сервисов .....	265
Инструменты для документирования сервисов.....	266
Основы спецификации OpenAPI (OAS) .....	267
Документирование сервисов с минимальными API с помощью OpenAPI .....	268
Визуальное документирование с помощью диаграмм Mermaid.....	272
Рендеринг диаграмм Mermaid .....	273
Создание блок-схемы с помощью Mermaid .....	275
Создание диаграммы классов с помощью Mermaid .....	276
Преобразование диаграмм Mermaid в формат SVG .....	278

Практика и исследование.....	279
Упражнение 6.1. Материалы, доступные только онлайн.....	279
Упражнение 6.2. Практические задания.....	280
Упражнение 6.3. Проверьте свои знания.....	280
Упражнение 6.4. Дополнительная информация .....	281
Резюме.....	281
<b>Глава 7. Наблюдение и динамическое изменение выполнения кода .....</b>	<b>282</b>
Работа с рефлексией и атрибутами.....	283
Метаданные в сборках .NET .....	283
Динамическая загрузка сборок и выполнение методов.....	291
Предупреждение о рефлексии и Native AOT .....	297
Изменения рефлексии в .NET 9 .....	298
Дополнительные возможности рефлексии.....	298
Работа с деревьями выражений .....	299
Обзор компонентов деревьев выражений .....	301
Выполнение простейшего дерева выражений.....	301
Создание генераторов исходного кода .....	302
Реализация простейшего генератора исходного кода .....	303
Практика и исследование.....	307
Упражнение 7.1. Материалы, доступные только онлайн.....	307
Упражнение 7.2. Практические задания.....	307
Упражнение 7.3. Проверьте свои знания.....	307
Упражнение 7.4. Дополнительная информация .....	308
Резюме.....	308
<b>Глава 8. Защита данных и приложений с помощью криптографии .....</b>	<b>309</b>
Основные понятия в сфере безопасности .....	309
Методы защиты данных.....	310
Ключи и их размеры.....	311
Векторы инициализации и размеры блоков.....	312
Соль.....	313
Генерация ключей и векторов инициализации .....	314
Шифрование и расшифровка данных .....	314
Симметричное шифрование с помощью AES.....	316
Хеширование данных .....	321
Хеширование с помощью популярного алгоритма SHA-256 .....	322
Создание цифровой подписи .....	327
Создание подписи с помощью SHA-256 и RSA.....	327
Генерация случайных чисел для криптографии .....	329
Проверка подлинности и авторизация пользователей .....	331
Механизмы проверки подлинности и авторизации.....	332
Реализация проверки подлинности и авторизации.....	334
Защита функциональности приложения.....	337
Проверка подлинности и авторизация в реальных приложениях.....	338

Новые возможности .NET 9 .....	339
Метод CryptographicOperations.HashData.....	339
Алгоритм КМАС .....	339
Практика и исследование.....	340
Упражнение 8.1. Дополнительные онлайн-материалы .....	340
Упражнение 8.2. Практические задания .....	340
Упражнение 8.3. Проверьте свои знания .....	340
Упражнение 8.4. Дополнительная информация .....	341
Резюме.....	341
<b>Глава 9. Создание чат-сервиса на основе LLM.....</b>	<b>342</b>
Обзор технологии LLM .....	342
Принципы работы LLM .....	343
Получение доступа к LLM .....	345
Интеграция семантического ядра с моделью OpenAI.....	349
Знакомство с семантическим ядром.....	349
Функции моделей OpenAI.....	355
Добавление функций .....	356
Сохранение сеанса и поддержка нескольких функций .....	361
Потоковая передача ответов.....	363
Добавление журналирования и обеспечение отказоустойчивости .....	364
Запуск локальной LLM-модели.....	366
Платформа Hugging Face .....	366
Платформа Ollama.....	368
Приложение LM Studio .....	374
Практика и исследование.....	376
Упражнение 9.1. Дополнительные онлайн-материалы .....	376
Упражнение 9.2. Практические задания .....	377
Упражнение 9.3. Проверьте свои знания .....	377
Упражнение 9.4. Дополнительная информация .....	378
Резюме.....	378
<b>Глава 10. Внедрение зависимостей, контейнеры и время жизни сервисов.....</b>	<b>379</b>
Знакомство с внедрением зависимостей .....	379
Для чего внедряются зависимости.....	380
Механизмы внедрения зависимостей в .NET.....	381
Примеры в современном .NET .....	382
Регистрация времени жизни сервиса зависимостей.....	385
Регистрация нескольких реализаций.....	385
Когда возникают исключения? .....	386
Лучшие практики внедрения зависимостей (DI).....	387
Реализация универсального узла .NET .....	388
Ключевые особенности универсального узла .NET .....	388
Создание универсального узла .NET .....	388

Обзор сервисов и событий узла .....	393
Методы регистрации сервисов .....	397
Графы зависимостей и разрешение сервисов.....	398
Удаление сервисов.....	399
Внедрение зависимостей с помощью ASP.NET Core .....	399
Регистрация сервисов для функций с помощью методов расширения .....	399
Когда конструктор не подходит для получения зависимостей .....	400
Разрешение сервисов при запуске .....	402
Внедрение зависимостей в представлениях .....	402
Методы действий и минимальные API .....	402
Практика и исследование.....	403
Упражнение 10.1. Дополнительные онлайн-материалы .....	403
Упражнение 10.2. Практические задания.....	403
Упражнение 10.3. Проверьте свои знания.....	403
Упражнение 10.4. Дополнительная информация.....	404
Резюме.....	404
<b>Глава 11. Модульное и mock-тестирование.....</b>	<b>405</b>
Обзор всех видов тестирования .....	406
Модульное тестирование.....	406
Интеграционное и сквозное тестирование, проверка безопасности .....	407
Тестирование производительности, нагрузки и стресс-тестирование.....	408
Функциональное и юзабилити-тестирование .....	408
Терминология тестирования.....	408
Характеристики качественных тестов .....	409
Результаты тестирования.....	410
Тестовые дублеры, имитации и заглушки.....	411
Формирование подхода к тестированию .....	412
Преимущества и недостатки разработки через тестирование.....	412
Основные принципы разработки через тестирование .....	412
Преимущества разработки через тестирование.....	413
Недостатки разработки через тестирование .....	413
Рекомендуемые приемы разработки через тестирование .....	414
Модульное тестирование с помощью xUnit .....	415
Насколько изолированными должны быть модульные тесты .....	415
Схема именования методов модульного тестирования.....	415
Модульное тестирование с помощью xUnit .....	416
Распространенные атрибуты xUnit .....	418
Создание testируемой системы (SUT) .....	419
Создание простых модульных тестов .....	420
Методы тестирования с параметрами.....	423
Положительные и отрицательные результаты теста.....	427
Предупреждающие признаки в модульных тестах .....	427

Просмотр результатов во время выполнения теста.....	428
Настройка и завершение работы .....	429
Управление фикстурами тестов.....	432
Имитация сущностей в тестах.....	435
Библиотеки для имитаций.....	437
Создание тестовых дублеров с помощью NSubstitute .....	438
Создание имитаций с помощью NSubstitute на практике .....	439
Произвольные утверждения в модульном тестировании.....	441
Утверждения для строковых значений .....	442
Создание утверждений для коллекций и массивов .....	443
Создание утверждений для значений даты и времени .....	444
Генерация тестовых данных с библиотекой Bogus .....	445
Тестовый проект с фальшивыми данными .....	448
Создание теста с фальшивыми данными .....	449
Практика и исследование.....	451
Упражнение 11.1. Дополнительные онлайн-материалы .....	451
Упражнение 11.2. Практические задания.....	451
Упражнение 11.3. Проверьте свои знания.....	452
Упражнение 11.4. Дополнительная информация .....	453
Резюме.....	453
<b>Глава 12. Интеграционное тестирование и проверка безопасности .....</b>	<b>454</b>
Основы интеграционного тестирования.....	454
Какие внешние системы включать в интеграционное тестирование.....	456
Совместное использование фикстур в интеграционных тестах .....	456
Обзор примера интеграционного теста .....	457
Интеграционное тестирование с хранилищами данных .....	458
Разработка экземпляров базы данных и миграции.....	459
Жизненный цикл данных.....	461
Тестирование сервисов с помощью туннелей разработки.....	463
Установка CLI для туннелей разработки .....	464
Создание туннеля разработки с помощью CLI и эхо-сервиса.....	465
Исследование туннеля разработки с проектом ASP.NET Core .....	466
Основы тестирования безопасности .....	471
Открытый проект обеспечения безопасности приложений.....	472
Отчет OWASP Top 10 .....	473
Моделирование угроз .....	478
Практика и исследование.....	480
Упражнение 12.1. Дополнительные онлайн-материалы .....	480
Упражнение 12.2. Практические задания.....	480
Упражнение 12.3. Проверьте свои знания.....	480
Упражнение 12.4. Дополнительная информация .....	481
Резюме.....	481

<b>Глава 13.</b> Оценка производительности, нагрузочное и стресс-тестирование .....	482
Оценка производительности.....	482
Важность базовых показателей.....	483
Нотация «О большое» .....	484
Статистические метрики .....	486
Оценка производительности с помощью BenchmarkDotNet .....	487
Предотвращение ошибок при оценке производительности .....	493
Выявление неточных публикаций о производительности.....	496
Нагрузочное и стресс-тестирование.....	498
Apache JMeter .....	500
Bombardier – быстрый кросс-платформенный инструмент для оценки производительности HTTP .....	501
Применение инструмента Bombardier .....	502
Загрузка и установка инструмента Bombardier.....	502
Сравнение веб-сервисов с компиляцией АОТ и без нее .....	504
Тестирование созданных веб-сервисов с помощью Bombardier .....	507
Интерпретация результатов Bombardier .....	510
NBomber – фреймворк для нагрузочного тестирования .....	512
Сценарии NBomber.....	512
Моделирование нагрузки .....	512
Типы NBomber .....	513
Пример проекта NBomber .....	514
Практика и исследование.....	517
Упражнение 13.1. Дополнительные онлайн-материалы .....	517
Упражнение 13.2. Практические задания.....	518
Упражнение 13.3. Проверьте свои знания.....	518
Упражнение 13.4. Дополнительная информация .....	518
Резюме.....	518
<b>Глава 14.</b> Функциональное и комплексное тестирование сайтов и сервисов .....	520
Основы функционального и сквозного тестирования.....	520
Пример 1. Тестирование сервиса веб-API.....	521
Пример 2. Тестирование сайта ASP.NET Core .....	521
Пример 3. Тестирование чат-приложения на базе SignalR .....	522
Тестирование пользовательских веб-интерфейсов с помощью Playwright .....	523
Преимущества Playwright для разработчиков .NET.....	524
Альтернативы Playwright .....	525
Распространенные типы объектов в Playwright .....	526
Распространенные методы работы со страницами в Playwright .....	527
Распространенные методы поиска элементов в Playwright .....	527
Распространенные методы работы с локаторами в Playwright.....	528
Тестирование распространенных сценариев с помощью приложения eShopOnWeb .....	529

Взаимодействие с элементами веб-интерфейса .....	537
Выбор элементов списка и элементов управления .....	537
Отправка форм, проверка подлинности и валидация .....	539
Проверка адаптивности дизайна .....	539
Одностраничные приложения (SPA) и динамический контент .....	541
Генерация тестов с помощью инструмента Playwright Inspector.....	542
Тестирование веб-сервисов с помощью xUnit.....	546
Создание веб-сервиса, готового к тестированию .....	547
Создание тестового проекта .....	548
Практика и исследование.....	549
Упражнение 14.1. Дополнительные онлайн-материалы .....	550
Упражнение 14.2. Практические задания.....	550
Упражнение 14.3. Проверьте свои знания.....	550
Упражнение 14.4. Дополнительная информация .....	550
Резюме.....	551
<b>Глава 15. Контейнеризация с помощью Docker.....</b>	<b>552</b>
Знакомство с контейнеризацией .....	552
Устройство и преимущества контейнеров .....	554
Docker, Kubernetes и .NET Aspire.....	556
Реестры контейнеров .....	557
Основные концепции Docker.....	559
Инструменты и технологии Docker .....	560
Команды интерфейса командной строки (CLI) Docker .....	561
Создание образов с помощью файлов Dockerfile.....	562
Настройка портов и запуск контейнера .....	565
Интерактивный режим.....	566
Настройка переменных среды .....	567
Распространенные образы контейнеров Docker .....	568
Образы контейнеров .NET .....	568
CVE и Chiseled Ubuntu .....	569
Управление контейнерами с помощью Docker .....	570
Установка Docker и использование готовых образов .....	570
Иерархия и слои Docker .....	574
Контейнеризация проектов .NET .....	577
Контейнеризация проекта консольного приложения .....	577
Публикация проекта в контейнере Docker .....	580
Контейнеризация проекта ASP.NET Core .....	584
Работа с тестовыми контейнерами .....	587
Принцип работы Testcontainers для .NET .....	587
Пример использования .....	588
Практика и исследование.....	589
Упражнение 15.1. Дополнительные онлайн-материалы .....	589
Упражнение 15.2. Тренировочные упражнения.....	589

Упражнение 15.3. Проверьте свои знания.....	589
Упражнение 15.4. Дополнительная информация .....	590
Резюме.....	590
<b>Глава 16. Облачная разработка с помощью .NET Aspire .....</b>	<b>591</b>
Знакомство с Aspire .....	591
Что говорит команда Aspire.....	593
Редактор кода и поддержка CLI для Aspire .....	594
Запуск решения Aspire .....	594
Типы проектов Aspire .....	595
Типы ресурсов Aspire.....	596
Модель и оркестрация приложений Aspire.....	596
Шаблоны проектов Aspire.....	599
Изучение начального шаблона Aspire.....	599
Создание начального приложения Aspire .....	600
Структура начального решения Aspire.....	603
Подробный разбор возможностей Aspire.....	606
Информационная панель разработчика для мониторинга.....	606
Проект AppHost для оркестрации ресурсов .....	607
Проект ServiceDefaults для централизованной настройки .....	609
Как подключаются функциональные проекты в Aspire .....	611
Настройка Redis.....	612
Компоненты Aspire .....	613
Журналирование, трассировка и сбор метрик для обеспечения наблюдаемости...	615
Docker и Podman для контейнеров .....	616
Ожидание готовности контейнеров.....	616
А как же инструменты Dapr, Orleans и Project Tye?.....	617
Применение Aspire в новых и существующих решениях .....	620
Создание решения Aspire .....	620
Aspire и PostgreSQL .....	625
Использование томов данных и настройка постоянного пароля.....	626
Добавление Aspire к существующему решению.....	626
Обзор эталонного приложения eShop .....	628
Развертывание с помощью Aspire .....	633
Практика и исследование .....	634
Упражнение 16.1. Дополнительные онлайн-материалы .....	635
Упражнение 16.2. Практические задания.....	635
Упражнение 16.3. Проверьте свои знания.....	636
Упражнение 16.4. Дополнительная информация .....	636
Резюме.....	636
<b>Глава 17. Паттерны и принципы проектирования.....</b>	<b>637</b>
Принципы SOLID.....	638
Принцип единой ответственности (SRP).....	638
Принцип открытости/закрытости (OCP).....	641

Принцип подстановки Лисков (LSP) .....	644
Принцип разделения интерфейса (ISP) .....	648
Принцип инверсии зависимостей (DIP) .....	652
Паттерны проектирования .....	655
Порождающие паттерны.....	658
Структурные паттерны проектирования.....	661
Поведенческие паттерны проектирования .....	664
Принципы проектирования.....	667
DRY .....	667
KISS .....	668
YAGNI .....	668
Закон Деметры .....	669
Агрегирование вместо наследования .....	670
Принцип наименьшего удивления.....	672
Алгоритмы и структуры данных.....	674
Алгоритмы сортировки .....	675
Алгоритмы поиска .....	675
Алгоритмы работы со структурами данных .....	675
Алгоритмы хеширования .....	676
Рекурсивные алгоритмы.....	676
Практика и исследование.....	677
Упражнение 17.1. Дополнительные онлайн-материалы .....	677
Упражнение 17.2. Практические задания.....	677
Упражнение 17.3. Проверьте свои знания.....	677
Упражнение 17.4. Дополнительная информация.....	678
Резюме.....	678
<b>Глава 18. Основы архитектуры ПО и решений.....</b>	<b>679</b>
Основные понятия архитектуры ПО и решений.....	679
Архитектура ПО .....	680
Архитектура решений.....	680
Концепции архитектуры ПО .....	681
Архитектурные стили ПО .....	684
Концепции архитектуры решений .....	689
В заключение .....	691
Чистая архитектура Роберта Мартина .....	691
Концепции чистой архитектуры.....	692
Передовой опыт в области чистой архитектуры .NET .....	695
Построение архитектурных диаграмм с помощью Mermaid .....	696
Визуализация архитектуры и процессов с помощью Mermaid.....	696
Типы диаграмм Mermaid .....	697
Блок-схемы Mermaid .....	698
Диаграммы последовательностей Mermaid .....	702

---

Практика и исследование.....	706
Упражнение 18.1. Дополнительные онлайн-материалы.....	706
Упражнение 18.2. Практические задания.....	706
Упражнение 18.3. Проверьте свои знания.....	707
Упражнение 18.4. Дополнительная информация.....	707
Резюме.....	707
<b>Глава 19. Ваша карьера, командная работа и собеседования .....</b>	<b>708</b>
Работа в команде разработчиков .....	708
Чем занимается .NET-разработчик.....	708
Роли в команде разработчиков, с которыми вам предстоит взаимодействовать.....	711
Процесс введения в должность .....	714
Как поднять тему профессионального развития .....	716
Парное программирование .....	718
Подготовка к собеседованию .....	720
До отклика на вакансию .....	720
Комплексная подготовка к собеседованию.....	724
Примеры вопросов, задаваемых на собеседованиях.....	745
Практика и исследование.....	760
Упражнение 19.1. Дополнительные онлайн-материалы.....	760
Упражнение 19.2. Практические задания.....	760
Упражнение 19.3. Проверьте свои знания.....	760
Упражнение 19.4. Дополнительная информация.....	761
Резюме.....	761
<b>Глава 20. Эпилог .....</b>	<b>762</b>
Следующие шаги на пути к изучению .NET .....	762
Сопутствующие книги для продолжения обучения.....	762
Обновленное девятое издание книги C# и .NET – основы кросс-платформенной разработки .....	763
Будущее трилогии книг о .NET 10 .....	764
<b>Приложение. Ответы к разделам «Проверьте свои знания» .....</b>	<b>765</b>

# Об авторе

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

*Спасибо всем моим читателям. Именно ваша поддержка вдохновляет меня писать эти книги и радоваться вашим успехам.*

*Особая благодарность тем, кто делится отзывами — в моем репозитории на GitHub, по электронной почте и в тематических сообществах Discord. Благодаря вам с каждым изданием эти книги становятся лучше.*

# О научных редакторах

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

**Милан Йованович** — опытный специалист, удостоенный звания Microsoft MVP, с более чем семилетним стажем работы в индустрии. Ранее занимал должность архитектора ПО, а сейчас сосредоточен на создании обучающего контента, помогая .NET-разработчикам развиваться в архитектуре и проектировании. Специализируется на разработке веб-приложений и особенно интересуется предметно-ориентированным проектированием (Domain-Driven Design, DDD).

*Хочу поблагодарить мою замечательную жену Милицу за неизменную поддержку. Я люблю тебя, Мико!*

*Отдельное спасибо моим удивительным родителям — за все, что вы для меня сделали. Я никогда не смогу отплатить вам сполна, но буду стараться.*

*Особая благодарность Марку Прайсу — автору этой книги. Его работы невероятно информативны, и я получаю искреннее удовольствие от их прочтения.*

# Предисловие

Существуют книги по программированию объемом тысячи страниц, авторы которых стремятся охватить все: безопасность, внедрение зависимостей, модульное и веб-тестирование, контейнеризацию, облачные хранилища, паттерны проектирования, а также так называемые гибкие навыки — вплоть до подготовки к техническому собеседованию.

Эта книга — другая. У вас в руках пошаговое руководство по освоению различных технологий разработки приложений и сервисов на платформе .NET. Она лаконична и подает материал в виде живого, насыщенного практикой контента с примерами и пояснениями. Да, из-за широты охвата темы где-то придется пожертвовать глубиной. Но вы найдете множество ссылок для дальнейшего изучения — я сопровождаю ими почти каждый раздел.

По моему опыту, самое сложное при изучении новой технологии — это начать. Но стоит разобраться с ключевыми понятиями и взглянуть на реальные примеры кода — как появляется уверенность, а за ней и желание копнуть глубже, изучить документацию, поэкспериментировать самостоятельно.

Эта книга предназначена для тех, кто уже знаком с основами языка C# и платформы .NET. Для тех, кто написал хотя бы пару собственных приложений или сервисов и теперь хочет освоить практики, востребованные в профессиональной корпоративной среде. Мы пройдемся по наиболее важным инструментам и навыкам .NET-разработчика, чтобы вы могли обсуждать с командой архитектурные решения и технологии — и с уверенностью браться за их реализацию.

## Примеры кода

Решения всех пошаговых заданий и упражнений вы можете скачать или клонировать из репозитория на GitHub: [github.com/markjprice/tools-skills-net8](https://github.com/markjprice/tools-skills-net8). В конце главы 1 есть подробная инструкция, как это сделать.

## Структура книги

Здесь я привожу краткое содержание каждой главы. Более развернутое описание структуры, а также объяснение выбора тем приведены в первой главе.

## Введение

В главе 1 вы подготовите рабочее окружение. Вы можете выбрать любую удобную программу: Visual Studio 2022, Visual Studio Code, JetBrains Rider или другой редактор кода. Также подробно объясняется, какие темы охватывает книга и почему именно они были выбраны. Вы создадите несколько библиотек классов — для описания моделей сущностей и работы с базой данных. Эти библиотеки будут использоваться в проектах, которые вы начнете разрабатывать в следующих главах.

В конце главы рассказывается, как адаптировать материалы книги для работы на платформе .NET 9, выпущенной в ноябре 2024 года.

## Инструменты

Ключевые инструменты профессионального .NET-разработчика — это редактор кода, система контроля версий Git, отладчики и средства анализа памяти.

В главе 2 мы сосредоточимся на менее известных возможностях, встроенных в Visual Studio, Visual Studio Code и JetBrains Rider. Вы научитесь настраивать редактор под себя, писать код эффективно и использовать встроенные функции рефакторинга.

В главе 3 рассматриваются наиболее распространенные задачи, связанные с использованием Git для управления исходным кодом — особенно в командной разработке. Описываются приемы, которые помогут .NET-разработчику организовать чистую, понятную историю изменений и эффективно взаимодействовать с коллегами.

В главе 4 вы познакомитесь с инструментами отладки и анализа памяти, доступными прямо в редакторе кода. Вы узнаете, как находить ошибки, отслеживать поведение приложения во время выполнения и выявлять утечки памяти или неэффективные места в коде.

В главе 5 речь пойдет о внедрении инструментов наблюдаемости на этапах тестирования и эксплуатации, а также о сборе телеметрии.

## Навыки

Документирование, создание динамического кода, защита и интеграция *искусственного интеллекта (ИИ)* — важнейшие навыки профессионального .NET-разработчика.

В главе 6 рассматриваются подходы к документированию кода с помощью комментариев, чтобы его могли сопровождать другие разработчики, а также способы описания сервисов и API — для их корректного использования.

Глава 7 посвящена распространенным типам .NET, применяемым для рефлексии, чтения и использования атрибутов, работы с деревьями выражений, а также для генерации кода во время выполнения.

В главе 8 рассматриваются средства шифрования для защиты данных от несанкционированного доступа, а также хеширования и цифровой подписи — для предотвращения изменений и повреждений. Кроме того, описываются механизмы проверки подлинности и авторизации, защищающие приложения от неавторизованного доступа.

В главе 9 вы создадите пользовательский чат-сервис, интегрированный с искусственным интеллектом, построенным на *большой языковой модели* (*Large Language Model, LLM*).

## Тестирование

Важнейшие инструменты .NET-разработчика — это управление зависимостями между сервисами и всестороннее тестирование, от модульного до пользовательского интерфейса. Владение этими инструментами необходимо каждому профессиональному.

В главе 10 рассматриваются приемы уменьшения жесткой связности между компонентами — особенно важные для эффективного тестирования. Такой подход упрощает сопровождение, снижает сложность и повышает гибкость системы.

Глава 11 посвящена методам тестирования, повышающим качество кода. Модульные тесты полезны, но при неправильной реализации могут ввести в заблуждение команду. Если же применять их грамотно — это экономит время, снижает издержки и ускоряет разработку.

В главе 12 рассматриваются два уровня тестирования — интеграционное и проверка безопасности. Такие тесты охватывают все решение целиком, проверяя взаимодействие компонентов и устойчивость к уязвимостям.

В главе 13 вы познакомитесь с библиотекой BenchmarkDotNet и научитесь измерять производительность кода. Также рассматриваются приемы нагрузочного и стресс-тестирования, которые позволяют спрогнозировать потребление ресурсов и рассчитать затраты на развертывание.

Глава 14 посвящена функциональному и сквозному тестированию API сервисов и веб-интерфейсов. Оно охватывает все уровни взаимодействия и позволяет проверить корректность работы системы в целом.

В главе 15 вы познакомитесь с концепцией контейнеризации на основе платформы Docker — для изоляции сервисов и виртуализации хостов в сложных архитектурах решений.

В главе 16 рассматривается фреймворк .NET Aspire, с помощью которого можно воссоздать облачную среду разработки на локальном компьютере. Благодаря этому можно разрабатывать и тестировать облачные приложения без необходимости постоянного подключения к облачной инфраструктуре.

## Построение карьеры в сфере разработки

К числу ключевых навыков .NET-разработчика относятся знание общих паттернов проектирования, реализация алгоритмов и архитектурных решений. Также важно уметь строить карьеру, к которой вы стремитесь, — начиная с резюме и заканчивая успешным прохождением собеседования.

В главе 17 рассматриваются паттерны, основанные на принципах SOLID, а также распространенные паттерны, такие как Одиночка (Singleton) и Фабрика (Factory), которые широко применяются в .NET.

В главе 18 вы познакомитесь с подходами к архитектуре ПО и архитектурными решениями, необходимыми для построения масштабируемых и поддерживаемых систем.

Глава 19 рассказывает о том, как эффективно работать в команде, подавать резюме и проходить собеседования на должность .NET-разработчика и смежные роли. Приведены 60 часто задаваемых вопросов, встречающихся на собеседованиях. Первые три вопроса сопровождаются ответами прямо в книге. Полный список с ответами доступен в формате PDF на сайте GitHub по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/interview-qa/readme.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/interview-qa/readme.md).

В послесловии приведены рекомендации по дальнейшему обучению и развитию, чтобы вы смогли овладеть навыками, необходимыми высококвалифицированному .NET-разработчику.

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

## Что вам понадобится для работы с этой книгой

Вы можете разрабатывать и развертывать проекты на C# и .NET с помощью Visual Studio 2022, Visual Studio Code или сторонних инструментов, таких как JetBrains Rider и консольные приложения, работающие в большинстве операционных систем — включая Windows, macOS и многие версии Linux. Программа Visual Studio предназначена только для Windows. Версия для macOS официально не поддерживает .NET 8 с августа 2024 года.

## Полноцветные иллюстрации из этой книги

К книге прилагается PDF-файл с полноцветными оригиналами снимками экрана и диаграммами, приведенными в тексте. На этих иллюстрациях лучше видны изменения в коде и результат его выполнения. Вы можете скачать файл по ссылке [packt.link/gbp/9781837635207](https://packt.link/gbp/9781837635207).

## Условные обозначения

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

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

Листинги оформлены следующим образом:

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

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

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

Ввод/вывод командной строки/терминала оформлен следующим образом:

```
dotnet new console
```

*Курсивным* начертанием выделяется новый термин или важное слово.

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



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



**Дельный совет:** так оформлены экспертные рекомендации по программированию.

# От издательства

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

Мы будем рады узнать ваше мнение!

# 1

# Обзор инструментов и навыков для работы с .NET

В первой главе мы познакомимся с инструментами и навыками, которые вы изучите в этой книге, настроим среду разработки, установив программы Visual Studio 2022, Visual Studio Code и JetBrains Rider, а также разберемся с созданием базы данных и проектов для использования в остальных главах. В книге я буду употреблять названия *Visual Studio*, *Code* и *Rider* для обозначения этих трех редакторов кода.

Упоминая в этой книге слово «современная» в контексте платформы .NET, я имею в виду версию 8 и ее предшественницу, такую как .NET 6, произошедшую из .NET Core. Слово «устаревшая» касается платформ .NET Framework, Mono, Xamarin и .NET Standard. Современная версия .NET консолидирует перечисленные унаследованные платформы и стандарты.

Каждая глава книги охватывает определенные инструменты и навыки, при этом некоторые из глав более детально посвящены конкретным инструментам или аспектам.



В репозитории GitHub, который я создал для этой книги, размещены полные решения для всех описываемых задач. Он доступен по ссылке [github.com/markjprice/tools-skills-net8/](https://github.com/markjprice/tools-skills-net8/).

Перейдя на страницу репозитория GitHub, нажмите клавишу . (точка) или в адресной строке измените домен .com на .dev, чтобы открыть облачный веб-редактор GitHub Codespaces, основанный на ПО Visual Studio Code.

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

## Знакомство с этой книгой и ее содержимым

Прежде чем перейти к обзору содержания, давайте обозначим контекст. Эта книга — одна из трех, которые я написал о платформе .NET 8, и она охватывает почти все ключевые аспекты, необходимые начинающему разработчику для уверенного старта в работе с .NET.

## Сопутствующие книги для продолжения обучения

Эта книга — третья и завершающая трилогию, посвященную платформе .NET 8.

1. В первой книге, *C# 12 and .NET 8 — Modern Cross-Platform Development Fundamentals*, рассматриваются основы языка C# и библиотеки .NET, Blazor и ASP.NET Core для разработки веб-приложений. Она предназначена для последовательного чтения, поскольку навыки и знания, полученные в предыдущих главах, дополняются и необходимы для понимания материала последующих глав.
2. Во второй книге<sup>1</sup> рассматриваются более узкие темы, такие как интернационализация, а также популярные пакеты сторонних разработчиков, в том числе Serilog и NodaTime. Вы научитесь разрабатывать нативные сервисы с *AOT-компиляцией* (ahead-of-time) на минимальных API ASP.NET Core и узнаете, как повысить производительность, масштабируемость и надежность посредством кэширования данных, очередей и фоновых сервисов. С технологиями GraphQL, gRPC, SignalR и Azure Functions вы реализуете дополнительные сервисы. Наконец, вы узнаете, как с помощью Blazor и .NET MAUI создавать графические пользовательские интерфейсы для сайтов, классических и мобильных приложений.
3. В третьей книге, которую вы держите в руках, рассматриваются важные инструменты и навыки, которыми должен обладать каждый профессиональный разработчик .NET. Сюда входят паттерны проектирования и архитектуры решений, отладка, анализ памяти, все важные виды тестирования — модульное, интеграционное, производительности, веб- и мобильное, а также темы, важные для тестирования облачных решений на локальном компьютере, такие как контейнеризация, Docker и .NET Aspire. Наконец, мы рассмотрим, как подготовиться к собеседованию на замещение вакансии разработчика .NET.

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

## Целевая аудитория

Эта книга рассчитана на следующих читателей.

- Читателей моей книги *C# 12 and .NET 8 — Modern Cross-Platform Development Fundamentals*, желающих продолжить обучение.
- Разработчиков с базовыми навыками и знаниями языка C# и платформы .NET, которые стремятся развить практические навыки, освоить распространенные инструменты и повысить уровень профессионализма в разработке на .NET, а также присоединиться к команде разработчиков .NET.

<sup>1</sup> Прайс М. .NET 8: приложения и сервисы. Практика создания проектов с использованием Blazor, .NET MAUI, gRPC, GraphQL. 2-е изд. — СПб.: Питер, 2025.



**Рис. 1.1.** Трилогия для изучения C# 12 и .NET 8 для начинающих и более опытных читателей

Рассмотрим аналогию.

- Во-первых, начинающий повар может купить книгу, чтобы освоить базовые навыки, ключевые понятия и терминологию, необходимые для приготовления самых популярных блюд.
- Во-вторых, он может взять сборник рецептов, чтобы научиться применять эти знания и навыки на практике — для создания полноценных блюд.
- В-третьих, чтобы стать профессиональным поваром, ему нужно понять, как работает кухня как коллектив, освоить специализированные инструменты и подходы, необходимые для приготовления пищи на большое количество людей, и научиться взаимодействовать с другими поварами в команде.

Именно поэтому я написал три книги о .NET — каждая из них соответствует одному из этих этапов.

Предисловие дает краткий обзор глав. Но так как многие читатели имеют привычку его пропускать, давайте чуть подробнее поговорим о том, почему каждая тема в книге раскрыта так глубоко и последовательно.

## Инструменты

Профессиональные разработчики .NET должны быть знакомы со множеством инструментов. Некоторые из них встроены в редакторы кода, например функции отладки и интеграция с системой управления исходным кодом. Другим инструментам требуется отдельные приложения и сервисы, такие как анализ памяти и телеметрия.



Даже начинающие разработчики обычно знают основы работы с инструментами, встроенными в редактор кода: как использовать главное окно редактирования, управлять файлами проекта, устанавливать точки останова, запускать отладку, просматривать код пошагово и запускать проект. Эти темы уже знакомы многим, поэтому в данной книге они не будут детально рассматриваться.

Глава 2 посвящена менее известным инструментам, встроенным в Visual Studio, Visual Studio Code и JetBrains Rider. Основные компоненты — такие как отладчик и средства анализа памяти — будут рассмотрены в отдельных главах. Здесь основное внимание уделяется рефакторингу и настройке редактора кода с учетом стандартов, например `.editorconfig`. Таким образом улучшается совместная работа и обеспечивается единообразие стиля кодирования в команде.

В главе 3 рассматриваются наиболее распространенные задачи, которые .NET-разработчики выполняют с помощью Git при управлении исходным кодом — особенно в командной работе. Git — это распределенная система контроля версий, позволяющая вести разработку в автономном режиме, создавать локальные репозитории, быстро работать с ветками и объединять изменения. Git остается основным инструментом управления версиями в экосистеме .NET. Для него существует множество расширений, интегрируемых с любыми редакторами кода и интерфейсами командной строки. Платформа GitHub, принадлежащая Microsoft, широко используется для хранения репозиториев и совместной работы над проектами.

В главе 4 вы познакомитесь с инструментами отладки, встроенными в редакторы кода. Вы узнаете, как пользоваться отладчиком, размечать код атрибутами для отслеживания событий, а также анализировать потребление памяти. Так вы повысите эффективность и производительность приложений и сервисов.

Глава 5 посвящена внедрению в код инструментов трассировки, сбора метрик и журналирования. Вы узнаете, как применять стандарт OpenTelemetry для сбора телеметрии и мониторинга в реальном времени, что позволяет достичь высокой степени наблюдаемости за состоянием системы.

## Навыки

Профессиональному .NET-разработчику важно не только владеть инструментами, но и развивать ключевые навыки — такие как документирование кода, работа с динамическим кодом, а также применение криптографических методов для защиты данных и приложений. В этой книге вы также создадите собственный чат-сервис, использующий модель с поддержкой пользовательских функций.

Глава 6 посвящена эффективным приемам документирования кода, чтобы его было легко поддерживать другим разработчикам. Вы узнаете, как правильно писать комментарии и оформлять документацию для сервисов и API, чтобы ими можно было пользоваться по назначению. Процесс документирования часто помогает выявить участки кода, которые стоит улучшить, — а значит, вы улучшите и архитектуру, и читаемость проекта.

В главе 7 рассматриваются распространенные типы .NET, позволяющие выполнять рефлексию, управлять атрибутами, работать с деревьями выражений и создавать генераторы кода. Эти инструменты особенно полезны при работе в динамически меняющейся среде, когда требуется дополнительная гибкость.

Глава 8 посвящена методам шифрования для защиты данных от несанкционированного доступа, а также хешированию и цифровым подписям — для предотвращения изменений и повреждений данных. Вы также научитесь внедрять проверку подлинности и авторизацию — для защиты приложений от неавторизованного доступа.

В главе 9 вы создадите чат-сервис, интегрированный с большой языковой моделью (LLM), дополненной пользовательскими функциями для обработки и анализа бизнес-данных.

В главе 10 рассматриваются принципы внедрения зависимостей, позволяющие ослабить связность между компонентами, облегчить тестирование, упростить сопровождение и повысить гибкость архитектуры.

## Тестирование

Одним из ключевых инструментов .NET-разработчика является тестирование — важнейший компонент обеспечения качества продукта. Тесты охватывают весь жизненный цикл разработки: от проверки отдельных участков кода до сквозного тестирования пользовательского опыта и системной интеграции. Чем раньше в процессе начинается тестирование, тем дешевле и быстрее обходится исправление ошибок.

Глава 11 посвящена методам, помогающим повысить общее качество кода. Модульные тесты легко реализовать неправильно — и тогда они теряют ценность и подры-

вают доверие команды. Но при грамотной реализации они позволяют сэкономить ресурсы и ускорить разработку.

В главе 12 рассматриваются более сложные уровни тестирования, охватывающие все решение целиком, а также методы, направленные на обеспечение безопасности проекта.

Из главы 13 вы узнаете, как использовать библиотеку BenchmarkDotNet для мониторинга и оценки производительности кода. Также рассматриваются нагрузочное и стресс-тестирование с помощью инструментов Bombardier и NBomber, которые помогают прогнозировать потребление ресурсов и оценивать затраты на продакшен-развертывание.

Глава 14 посвящена автоматизированному тестированию API и пользовательских интерфейсов с использованием фреймворка Playwright.

В главе 15 вы познакомитесь с концепцией контейнеризации и узнаете, как применять платформу Docker для виртуализации хостов в сложных архитектурах.

В главе 16 рассматривается .NET Aspire — облачная нативная среда разработки, автоматизирующая локальное развертывание и интеграцию с инструментами, такими как OpenTelemetry для наблюдаемости, Docker для контейнеризации микросервисов, и другие вспомогательные средства.

## Построение карьеры в сфере разработки

Заключительная часть книги посвящена теоретическим основам и профессиональному росту. Вы познакомитесь с паттернами проектирования, архитектурными принципами и подходами, а также с рекомендациями по подготовке к собеседованиям на позиции в командах разработчиков и смежных профессиональных ролях.

Глава 17 знакомит с ключевыми паттернами и принципами разработки, включая *SOLID*, а также с другими важными концепциями: *DRY (Don't Repeat Yourself)*, *KISS (Keep It Simple, Stupid)*, *YAGNI (You Ain't Gonna Need It)* и *PoLA (Principle of Least Astonishment)*.

В главе 18 рассматриваются архитектурные подходы к разработке ПО. Вы также изучите инструмент Mermaid, позволяющий создавать архитектурные диаграммы и визуализировать структуру решений.

В главе 19 даны практические рекомендации по поиску работы и подготовке к собеседованиям. Вы узнаете, как составить резюме, произвести впечатление на работодателя и представить свой опыт. Также приведены часто задаваемые вопросы и примеры ответов, которые помогут уверенно пройти собеседование и получить желаемую должность.

## Настройка среды разработки

Прежде чем приступить к программированию, вам нужно установить редактор кода для работы с C#. Microsoft предлагает несколько редакторов и *интегрированных сред разработки (IDE)*, включая следующие.

- **Visual Studio:** Visual Studio 2022 для Windows. Версия для macOS не рекомендуется, так как ее поддержка прекращена 31 августа 2024 года.
- **Code:** Visual Studio Code для Windows, macOS и Linux, а также в виде веб-версии или через GitHub Codespaces.
- **Rider:** программа JetBrains Rider доступна для компьютеров под управлением операционной системы Windows, Mac и Linux, требует приобретения лицензии. Эта среда особенно популярна среди более опытных разработчиков .NET.

## Выбор подходящего ПО

Какой редактор лучше всего подходит для создания приложений и сервисов на C# и .NET?

Для выполнения упражнений из книги вы можете использовать любой редактор или IDE с поддержкой языка C#, будь то Visual Studio, VS Code, Rider или даже что-то менее известное.

Все инструкции в книге универсальны и подходят для любого редактора. Выберите тот, с которым вам удобнее работать.

### Универсальная среда: Visual Studio 2022

Visual Studio 2022 поддерживает разработку большинства типов проектов — от консольных до мобильных приложений, включая сайты и веб-сервисы. Программа работает в операционной системе Windows 10 версии 1909 и выше или Windows Server 2016 и выше. Полноценная поддержка архитектуры Arm64 появилась, начиная с версии Visual Studio 2022 17.4.



**Внимание!** Программа Visual Studio 2022 для Mac официально не поддерживается .NET 8 с августа 2024 года. Если вы раньше использовали Visual Studio 2022 для Mac, вам следует перейти на Visual Studio Code для Mac, JetBrains Rider для Mac или запускать Visual Studio 2022 для Windows в виртуальной машине на локальном компьютере или в облаке с помощью технологии Microsoft Dev Box. Объявление о прекращении поддержки можно прочитать по адресу [devblogs.microsoft.com/visualstudio/visual-studio-for-mac-retirement-announcement/](https://devblogs.microsoft.com/visualstudio/visual-studio-for-mac-retirement-announcement/).

### Кросс-платформенная разработка в Visual Studio Code

Visual Studio Code — самый современный и легкий редактор кода от Microsoft, а также единственный кросс-платформенный редактор, официально поддерживаемый этой компанией. Он доступен для Windows, macOS и множества дистрибутивов Linux, включая Ubuntu и Red Hat Enterprise Linux (RHEL).

VS Code — оптимальный выбор для современной кросс-платформенной разработки. Он поддерживает широкий и постоянно расширяющийся каталог расширений, обеспечивающих работу не только с C#, но и со множеством других языков программирования. Этот легкий и удобный редактор можно установить на всех целевых платформах, что особенно удобно при отладке и быстром внесении изменений. Visual Studio Code предоставляет кросс-платформенный инструмент для разработки кросс-платформенных приложений.

Согласно опросу Stack Overflow за 2023 год, более 74 % профессиональных разработчиков выбрали именно Visual Studio Code ([survey.stackoverflow.co/2023/#most-popular-technologies-new-collab-tools-prof](https://survey.stackoverflow.co/2023/#most-popular-technologies-new-collab-tools-prof)), сделав ее самой популярной интегрированной средой разработки в мире.

## Облачная разработка в GitHub Codespaces

Github Codespaces — это полностью настраиваемая облачная среда разработки, построенная на основе Visual Studio Code. Она запускается на удаленном сервере и доступна прямо из любого браузера. В среде поддерживаются работа с Git-репозиториями, расширения VS Code и встроенный интерфейс командной строки.

Проекты, созданные в Codespaces, можно редактировать, запускать и тестировать с любого устройства. Поскольку вся среда работает в облаке, не требуется установка или настройка ПО на локальном компьютере. А благодаря тому, что основой служит Visual Studio Code, интерфейс и поведение полностью совпадают с настольной версией редактора.



**В копилку знаний:** подробнее о платформе GitHub Codespaces можно узнать по адресу [github.com/features/codespaces](https://github.com/features/codespaces). Microsoft Dev Box — еще один вариант организации среды разработки: [azure.microsoft.com/ru-ru/products/dev-box](https://azure.microsoft.com/ru-ru/products/dev-box).

Обзоры платформы GitHub Codespaces доступны по следующим ссылкам:

- [dev.to/github/github-codespaces-ga-any-good-reviewed-and-tested-3e62](https://dev.to/github/github-codespaces-ga-any-good-reviewed-and-tested-3e62);
- [medium.com/@pooyan\\_razian/github-codespaces-a-different-way-to-code-da455777f9ab](https://medium.com/@pooyan_razian/github-codespaces-a-different-way-to-code-da455777f9ab).

## Кросс-платформенная разработка в JetBrains Rider

Rider — это интегрированная среда разработки (IDE) компании JetBrains, известной по таким продуктам, как IntelliJ IDEA (ведущая среда для Java и Kotlin) и ReSharper — популярное расширение для Visual Studio, ориентированное на .NET-разработку. Rider поддерживает операционные системы Windows, macOS и Linux.

Программа распространяется по платной лицензии, однако ее выбирают многие опытные разработчики за такие возможности, как проверка кода в реальном времени, контекстные действия и продвинутый рефакторинг. Выбор Rider нередко указывает на высокий уровень профессиональной подготовки разработчика.

Rider обычно получает поддержку новых версий платформы .NET с небольшим отставанием по сравнению с Visual Studio. Например, фреймворк .NET 8 был официально выпущен 14 ноября 2023 года. В тот же день вышла и поддерживающая его версия Visual Studio. Поддержка в Rider появилась чуть позже — 7 декабря 2023 года (версия 2023.3).

Несмотря на высокое качество продуктов JetBrains, Rider и расширение ReSharper иногда демонстрируют нестабильное поведение. Например, при работе с Razor Pages, Razor Views и Blazor Components могут возникать ложные ошибки вроде: «Невозможно определить символ». Хотя компиляция и запуск таких проектов проходят успешно, редактор может подсвечивать несуществующие проблемы.

Также замечены ошибки при использовании плагинов Unity Support и Heap Allocation Viewer: в проектах Unity возможны ложные предупреждения при упаковке (boxing) значений, таких как `int` и `DateTime`, в позиционные строковые аргументы. Они возникают из-за того, что в Unity используется собственный сборщик мусора, отличный от стандартной среды выполнения .NET. Подробнее об этой проблеме можно прочитать по адресу [docs.unity3d.com/Manual/performance-optimizing-code-managed-memory.html#boxing](https://docs.unity3d.com/Manual/performance-optimizing-code-managed-memory.html#boxing).

Для всех остальных проектов, включая примеры в этой книге, эти ошибки не имеют практического значения.

## Мое рабочее окружение

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

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

Благодаря использованию различных аппаратных и программных платформ вы сможете отслеживать различия в поведении проектов и разбираться с проблемами, возникающими в процессе разработки. В то же время любая из перечисленных выше комбинаций вполне подойдет для создания приложений и веб-сервисов, описанных в книге.

## Кросс-платформенное развертывание

Выбор редактора кода и операционной системы для разработки не ограничивает платформы, на которых можно развернуть ваши приложения.

.NET 8 поддерживает следующие платформы для развертывания.

- **Windows:** Windows 10 версии 1607 или более поздней. Windows 11 версии 22000 или более поздней. Windows Server 2012 R2 SP1 или более поздней версии. Nano Server версии 1809 или более поздней.
- **Mac:** macOS Catalina версии 10.15 или более поздней, а также в эмуляторе Rosetta 2 x64. Mac Catalyst 11.0 или более поздней версии.
- **Linux:** Alpine Linux 3.17 или более поздняя версия. Debian 11 или более поздняя версия. Fedora 37 или более поздняя версия. openSUSE 15 или более поздняя версия. Oracle Linux 8 или более поздняя версия. Red Hat Enterprise Linux (RHEL) 8 или более поздняя версия. SUSE Enterprise Linux 12 SP2 или более поздняя версия. Ubuntu 20.04 или более поздняя версия.
- **Android:** API 21 или более поздняя версия.
- **iOS и tvOS:** 11.0 или более поздняя версия.



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

## Загрузка и установка Visual Studio 2022

Многие профессиональные разработчики Microsoft используют Visual Studio 2022 в повседневной работе. Хотя для выполнения упражнений из этой книги вы можете пользоваться Visual Studio Code, я рекомендую также познакомиться с возможностями Visual Studio 2022.

Если вы работаете в среде, отличной от Windows, этот раздел можно пропустить и перейти к инструкциям по установке Visual Studio Code для macOS или Linux.

С октября 2014 года Microsoft предоставляет студентам, разработчикам проектов с открытым исходным кодом и частным пользователям бесплатную редакцию Visual Studio Community Edition.

Если вы еще не установили Visual Studio, выполните следующие шаги.

1. Загрузите дистрибутив Visual Studio 2022 версии 17.8 или более поздней по адресу [visualstudio.microsoft.com/ru/downloads/](https://visualstudio.microsoft.com/ru/downloads/).



Чтобы получить доступ к новым функциям Copilot, установите Visual Studio 2022 версии 17.10 или более поздней.

2. Запустите программу установки.
3. На вкладке **Workloads** (Рабочие нагрузки) выберите следующие компоненты:
  - **ASP.NET and web development** (ASP.NET и разработка веб-приложений);
  - **.NET desktop development** (Разработка классических приложений .NET) (данная нагрузка добавляет разработку консольных приложений);
  - **Desktop development with C++** (Разработка классических приложений на C++) со всеми компонентами по умолчанию (чтобы публиковать оптимизированные по скорости запуска и затратам памяти консольные приложения и веб-сервисы).
4. На вкладке **Individual components** (Отдельные компоненты) в разделе **Code tools** (Средства для работы с кодом) установите следующие флагки:
  - **GitHub Copilot;**
  - **Git for Windows** (Git для Windows).
5. Нажмите кнопку **Install** (Установить) и дождитесь, пока программа установки загрузит выбранное ПО и установит его.
6. Когда установка будет завершена, нажмите кнопку **Launch** (Запустить).
7. При первом запуске Visual Studio вам будет предложено авторизоваться в системе. Если у вас есть учетная запись Microsoft, укажите свой логин и пароль. Если нет, то зарегистрируйте новую учетную запись по адресу [signup.live.com](http://signup.live.com).
8. При первом запуске Visual Studio вам будет предложено настроить среду. В разделе **Development Settings** (Параметры разработки) выберите пункт **Visual C#**. В качестве цветовой темы я задал синий цвет, но вы можете выбрать любой, который предпочитаете.
9. Если вы хотите настроить сочетания клавиш, перейдите в меню **Tools ▶ Options** (Средства ▶ Параметры), а затем выберите пункт **Environment ▶ Keyboard** (Окружение ▶ Клавиатура).

## Сочетания клавиш Visual Studio 2022

В книге я, как правило, не указываю сочетания клавиш, поскольку во многих редакторах их можно настроить индивидуально. Я буду упоминать их только в тех случаях, когда комбинации совпадают в разных редакторах кода и используются достаточно часто.

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

## Издание Visual Studio Enterprise

Visual Studio Enterprise — это интегрированная среда разработки, ориентированная на крупные команды и организации с особыми требованиями к разработке, тестированию и развертыванию. Она включает в себя расширенные функции

для работы над проектами корпоративного уровня и сложных архитектур. Изданье Enterprise стоит тысячи долларов за лицензию, поэтому чаще всего используется в крупных компаниях, а не индивидуальными разработчиками. В связи с этим инструменты, доступные только в этой редакции, в книге не рассматриваются.

Перечислю некоторые функции, доступные только в Visual Studio Enterprise.

- *IntelliTrace* — запись и воспроизведение выполнения приложения, позволяющая отслеживать, что происходило до возникновения ошибки.
- *Code Map* — визуализация структуры кода и зависимостей, упрощающая навигацию в больших проектах.
- *Live Unit Testing* — автоматический запуск затронутых модульных тестов при изменении кода с мгновенным отображением результатов и покрытия.
- *Microsoft Fakes* — создание заглушек (*stubs*) и подменных объектов (*shims*) для изолированного тестирования.
- *Web Load and Performance Testing* — нагружочное тестирование производительности веб-приложений.
- *Coded UI Testing* — автоматизированная проверка пользовательского интерфейса на корректность работы.
- *CodeLens* — отображение в редакторе информации о коде: история изменений, ссылки, рабочие элементы.
- *Architecture and Dependency Validation* — контроль архитектурных зависимостей и соблюдения проектных стандартов.
- *Advanced Static Code Analysis* — расширенный статический анализ для выявления потенциальных ошибок и уязвимостей.
- *Release Management* — инструменты для автоматизации развертывания и управления релизами, помогающие разработчикам быстрее выпускать ПО.
- *Snapshot Debugger* — создание снимков состояния приложения во время его работы, без остановки среды, что позволяет выявлять и исправлять ошибки.
- *App Center Test* — автоматизированное облачное тестирование интерфейсов на тысячах реальных устройств.

## Visual Studio Code

Visual Studio Code за последние пару лет значительно улучшилась и приятно удивила даже сотрудников Microsoft своей популярностью. Если вы относитесь к числу тех, кто любит пробовать все новое и использовать возможности по максимуму, обратите внимание на редакцию *Insiders* — это ежедневная сборка, содержащая нововведения, готовящиеся к релизу.

Даже если вы планируете использовать для разработки исключительно Visual Studio 2022, я рекомендую установить Visual Studio Code и попробовать выполнить

в ней код из этой главы. А затем принять решение, в каком редакторе продолжить работу с остальной частью книги: в Visual Studio Code или Visual Studio 2022.

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

1. Загрузите и установите сборку Stable или Insiders программы Visual Studio Code по адресу [code.visualstudio.com](http://code.visualstudio.com).



**В копилку знаний:** если требуется дополнительная помощь в установке Visual Studio Code в вашей операционной системе, см. официальное руководство по адресу [code.visualstudio.com/docs/setup/setup-overview](http://code.visualstudio.com/docs/setup/setup-overview).

2. По ссылке [dotnet.microsoft.com/ru-ru/download](http://dotnet.microsoft.com/ru-ru/download) скачайте и установите пакет .NET SDK для версии .NET 8.0.



**Дельный совет:** на момент написания книги (июнь 2024 года) последняя версия SDK — 8.0.301. Всегда устанавливайте или обновляйте самую актуальную исправленную версию.

3. Чтобы установить расширение C# Dev Kit в графическом интерфейсе, необходимо сначала запустить приложение Visual Studio Code.
4. В Visual Studio Code нажмите кнопку Extensions (Расширения) или выберите команду меню View ▶ Extensions (Вид ▶ Расширения).
5. C# Dev Kit — одно из самых популярных расширений, поэтому оно появится в верхней части списка. В противном случае введите запрос C# Dev Kit в поле поиска.



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

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



**Дельный совет:** обязательно прочтайте лицензионное соглашение к расширению C# Dev Kit. В нем перечислены более строгие правила, чем в соглашении к расширению C#: [aka.ms/vs/csdevkit](http://aka.ms/vs/csdevkit).

## Установка расширений

Выполняя примеры из книги, вы будете использовать несколько разных расширений Visual Studio Code. Чтобы вы могли установить их прямо сейчас, я перечислил их в табл. 1.1.

**Таблица 1.1.** Расширения Visual Studio Code, используемые в этой книге

Название и идентификатор расширения	Описание
<b>C# Dev Kit</b> <code>ms-dotnettools.csdevkit</code>	Официальное расширение компании Microsoft, добавляющее поддержку инструментов C#-разработчиков. Управляйте проектами и решениями на панели обозревателя решений и проверяйте их с помощью интегрированной среды модульного тестирования. Зависит от расширений C# и IntelliCode for C# Dev Kit
<b>C#</b> <code>ms-dotnettools.csharp</code>	Добавляет поддержку C#-кода с дополнительным функционалом, в том числе подсветкой синтаксиса, IntelliSense, переходом к определениям, поиском ссылок, отладкой для .NET и поддержкой <code>csproj</code> -файлов в операционной системе Windows, macOS и Linux
<b>IntelliCode for C# Dev Kit</b> <code>ms-dotnettools.vscodeintellicode-csharp</code>	Реализует ИИ-функции разработки для разработчиков на Python, TypeScript/JavaScript, C# и Java
<b>MSBuild project tools</b> <code>tintoy.msbuild-project-tools</code>	Добавляет поддержку технологии IntelliSense в файлах проектов MSBuild, в том числе автодополнение элементов <code>&lt;PackageReference&gt;</code>
<b>SQL Server (mssql) for Visual Studio Code</b> <code>ms-mssql.mssql</code>	Расширение с богатым набором функциональных возможностей для разработки проектов SQL Server, Azure SQL Database и SQL-хранилищ данных
<b>REST Client</b> <code>humao.rest-client</code>	Позволяет отправлять и просматривать результаты HTTP-запросов прямо в программе Visual Studio Code
<b>ilspy-vscode</b> <code>icsharpcode.ilspy-vscode</code>	Расширение для декомпиляции сборок MSIL, поддерживающее современные версии .NET, .NET Framework, .NET Core и .NET Standard

## Управление расширениями в терминале

Вы можете установить расширение Visual Studio Code с помощью терминала (табл. 1.2).

**Таблица 1.2.** Управление расширениями в терминале

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

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

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



Я подготовил PowerShell-сценарии для установки и удаления перечисленных в таблице расширений Visual Studio Code. Вы найдете их по адресу [github.com/markjprice/tools-skills-net8/tree/main/scripts/extension-scripts](https://github.com/markjprice/tools-skills-net8/tree/main/scripts/extension-scripts).

## Версии Microsoft Visual Studio Code

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

- версия 1.90.0, релиз в мае 2024 года;
- версия 1.90.1, перевыпуск майской версии с исправленными ошибками.

Обратите внимание, что версия Microsoft Visual Studio Code не так важна, как актуальность установленного расширения C#. Хотя без расширения C# можно обойтись, оно внедряет удобные IntelliSense-инструменты автодополнения кода по мере ввода, функции навигации по коду и устранения ошибок. В силу этого я рекомендую его установить и обновлять для поддержки актуальных возможностей языка C#.

## Сочетания клавиш Visual Studio Code

В книге я не указываю сочетания клавиш для выполнения таких операций, как создание файла, поскольку в разных операционных системах они часто отличаются. Я буду приводить многократно нажимаемые сочетания клавиш, например, во время отладки. С высокой долей вероятности они одинаковы в разных операционных системах.

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

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

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

## Загрузка и установка Rider

Согласно опросу, проведенному на сайте Stack Overflow Survey в 2023 году, примерно 20 % профессиональных разработчиков на C# и .NET регулярно используют Rider в своей работе. Эта среда разработки кросс-платформенная, поэтому ее можно установить на устройствах под управлением операционной системы Windows, macOS и различных версий Linux. Также доступна бесплатная 30-дневная лицензионная пробная версия.

Дополнительно компания JetBrains выпускает удобное и бесплатное приложение *Toolbox App*, позволяющее управлять IDE. Оно особенно полезно для пользователей Linux, поскольку Rider не обновляется с помощью инструмента *apt* (Advanced Package Tool), как большинство Linux-программ. Toolbox App также предоставляет доступ к таким программам, как JetBrains Fleet — редактор кода нового поколения.

Если вы еще не установили Rider, но планируете, сделаем это прямо сейчас.

1. Загрузите последнюю версию Rider по адресу [www.jetbrains.com/rider/download/](https://www.jetbrains.com/rider/download/).



Если вы работаете на компьютере под управлением операционной системы Ubuntu 16.04 или более поздней версии, Rider можно установить через терминал с помощью команды:

```
sudo snap install rider --classic
```

2. Запустите установочный файл.
3. Загрузите последнюю версию приложения JetBrains Toolbox по адресу [www.jetbrains.com/toolbox-app/](https://www.jetbrains.com/toolbox-app/).
4. Запустите программу установки.
5. Перейдите по адресу [visualstudio.microsoft.com/ru/downloads/#build-tools-for-visual-studio-2022](https://visualstudio.microsoft.com/ru/downloads/#build-tools-for-visual-studio-2022) и установите набор инструментов *Build Tools for Visual Studio*. Эти инструменты необходимы для работы с компилятором Native AOT.



Функционал пакета JetBrains ReSharper в Visual Studio идентичен инструментам Rider.

## Другие инструменты JetBrains

Компания JetBrains выпускает набор инструментов, которые помогают разработчикам .NET повышать производительность, качество кода и быстродействие приложений. Рассмотрим инструменты dotPeek, dotTrace, dotMemory и dotCover, а также их возможности и преимущества.

- **dotPeek** — позволяет декомпилировать сборки .NET в читаемый код на C#, упрощая изучение и анализ внешнего кода без исходных файлов. Он предоставляет функции для навигации и поиска в декомпилированном коде, включая быстрый переход к определениям и реализациям. Это особенно полезно для понимания работы сторонних библиотек, фреймворков или любого скомпилированного .NET-кода без исходных файлов. dotPeek выполняет роль системы, которая предоставляет информацию о символах программы, их местоположении и связях, необходимую для работы с внешними библиотеками во время отладки. Этот инструмент распространяется бесплатно, поэтому доступен для всех разработчиков .NET.
- **dotTrace** — позволяет детально профилировать производительность приложений .NET, выявляя узкие места путем замеров времени выполнения и частоты вызовов. Он поддерживает режимы выборки, трассировки и временной шкалы, что позволяет адаптировать анализ под разные типы задач. dotTrace предоставляет разнообразные представления для анализа данных профилирования, включая деревья вызовов, «горячие точки» и временные шкалы, помогая оптимизировать производительность, выявляя медленные участки кода и неэффективные алгоритмы.
- **dotMemory** — анализирует использование памяти в приложениях .NET, помогая выявлять утечки и проблемы со *сборщиком мусора* (*Garbage Collector, GC*). dotMemory позволяет отслеживать, как выделяется и освобождается память, и выявлять чрезмерные выделения объемов памяти, которые могут привести к снижению производительности. Он также помогает в оптимизации памяти, предоставляя возможности для сравнения моментальных снимков использования памяти и обнаружения объектов, вызывающих утечки.
- **dotCover** — измеряет, насколько плотно кодовая база покрыта тестами dotCover, поддерживает различные фреймворки тестирования и работает со всеми приложениями .NET. Доступен режим непрерывного тестирования, который автоматически запускает тесты при изменении кода, предоставляя обратную связь о покрытии тестами и успешности их выполнения в реальном времени. Благодаря этому можно снизить количество ошибок, улучшить качество кода и сделать его стабильнее. dotCover повышает эффективность тестирования, выявляя непроверенный код и акцентируя внимание разработчиков на участках с недостаточным покрытием тестами.

Вместе эти инструменты составляют мощный набор, который может значительно повысить эффективность и производительность команд разработчиков .NET.

## ИИ-инструменты в браузере Chrome

Встроенные инструменты искусственного интеллекта появились и в браузере Chrome. Ознакомиться с ними можно по адресу [developer.chrome.com/docs/devtools/console/understand-messages](https://developer.chrome.com/docs/devtools/console/understand-messages).

## GitHub-репозиторий для этой книги

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

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

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

## При возникновении вопросов по книге

Если при выполнении шагов из книги вы столкнулись с затруднением или заметили ошибку в тексте или коде, задайте вопрос в репозитории на GitHub.

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

Следующая информация менее актуальна, но тоже может оказаться полезной:

- используемая операционная система, скажем 64-битная версия Windows 11 или macOS Big Sur версии 11.2.3;

- используемое аппаратное обеспечение, например процессор Intel, Apple Silicon или ARM.

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

## Связь с автором

Если вы хотите оставить общий отзыв о книге, то можете написать мне по адресу [markjprice@gmail.com](mailto:markjprice@gmail.com). Издательство Packt создало каналы Discord для читателей, чтобы они могли общаться с авторами и другими читателями. Вы можете присоединиться к нам по адресу [packt.link/TS1e](https://packt.link/TS1e).

Я люблю получать от своих читателей информацию о том, что им нравится в моих книгах, а также предложения по улучшению и работе с C# и .NET, так что не стесняйтесь. Пожалуйста, свяжитесь со мной!

Заранее благодарю вас за вдумчивый и конструктивный отзыв.

## Загрузка кода проектов из репозитория GitHub

В репозитории GitHub размещен код всех практических пошаговых примеров и упражнений из этой книги. Репозиторий доступен по адресу [github.com/markjprice/tools-skills-net8](https://github.com/markjprice/tools-skills-net8).

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

Рекомендую добавить показанную выше ссылку в избранное браузера.



**Дельный совет:** рекомендую клонировать/скачивать файлы примеров в папку с коротким именем, например, C:\toolskills-net8\ или C:\book\, чтобы предотвратить превышение максимальной длины пути у генерируемых компилятором файлов. Следует также избегать специальных символов, таких как #. Например, не используйте имя папки C:\C# projects\. Такое имя хотя и подойдет для записи проекта консольного приложения, но при автоматической генерации кода компилятором у вас возникнут проблемы. Имена папок должны быть короткими и простыми.

## Источники справочной информации

Этот раздел посвящен поиску качественной информации о программировании в интернете.

Мои книги написаны в учебном стиле, чтобы помочь вам быстро освоить основы, а затем обратиться к официальной документации для более детального изучения или работы с узкоспециализированными проектами.

Этот подход популярен среди разработчиков. По данным исследования Stack Overflow Survey 2023 ([survey.stackoverflow.co/2023/#section-learning-to-code-learning-how-to-code](https://survey.stackoverflow.co/2023/#section-learning-to-code-learning-how-to-code)), более 50 % изучают код с помощью книг и других физических источников, а свыше 80 % используют дополнительные онлайн-ресурсы.

## Техническая документация на сайте Microsoft Learn

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

## Получение справки с помощью команды dotnet

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

```
dotnet help <команда>
```

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



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

Документацию можно вывести и в самом терминале. Синтаксис следующий:

```
dotnet <команда> -? | -h | --help
```

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

## ИИ-инструменты, такие как ChatGPT и GitHub Copilot

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

В настоящее время используются две модели ChatGPT: 3.5 и 4.0 (обе бесплатные) и индивидуальная подписка Plus (20 долларов в месяц). Доступны также подписки Pro, Team и Enterprise. Сравнить их можно по адресу [openai.com/chatgpt/pricing/](https://openai.com/chatgpt/pricing/).

Допустим, вам нужно больше узнать об инструментах и навыках .NET. Вы можете ввести следующий запрос: «Какие инструменты и навыки наиболее важны для профессионального разработчика .NET?»

Ответ будет примерно следующим: «Профессиональному разработчику .NET важно владеть широким спектром инструментов и навыков, охватывающих весь цикл разработки. Вот основные.

## Инструменты

- **Visual Studio.** Основная среда разработки для .NET. Убедитесь, что вам удобно работать с ее функционалом и расширениями.
- **Visual Studio Code.** Легкий, но мощный и универсальный редактор кода, который набирает популярность среди разработчиков .NET, несмотря на мощь Visual Studio.
- **.NET Core / .NET 5 и далее.** Следите за последними версиями фреймворка .NET. .NET 5 и последующие версии обеспечивают кроссплатформенную поддержку и улучшенную производительность.
- **Git и GitHub.** Контроль версий играет ключевую роль. Git — популярная распределенная система контроля версий, а GitHub — платформа для совместной разработки и обмена кодом.
- **Azure DevOps.** Изучите Azure DevOps для управления проектами и организации CI/CD (непрерывная интеграция и развертывание).
- **NuGet.** Понимание управления пакетами с помощью NuGet важно для добавления библиотек и пакетов в проекты .NET.
- **Docker.** Контейнеризация становится важной для развертывания приложений .NET. Docker поможет эффективно создавать и управлять контейнерами.

## Навыки

- **C#.** C# — основной язык для разработчика .NET. Освойте основы объектно-ориентированного программирования, асинхронного программирования и расширений LINQ.
- **ASP.NET Core.** Для веб-разработки ASP.NET Core — один из самых популярных фреймворков. Изучите архитектуру MVC, Razor Pages и разработку API.
- **Entity Framework Core.** Освойте работу с базами данных, используя Entity Framework Core для эффективного доступа к данным.
- **Внедрение зависимостей.** Понимание принципов внедрения зависимостей важно для управления зависимостями объектов в .NET.
- **Модульное тестирование.** Написание модульных тестов с помощью инструментов, таких как xUnit или NUnit, необходимо для обеспечения надежности кода.

- **Проектирование веб-интерфейсов.** При создании API изучите принципы проектирования RESTful, проверки подлинности и версионирования.
- **Технологии фронтенда.** Для веб-разработки важно освоить HTML, CSS, JavaScript и такие фреймворки, как React или Angular.
- **Сервисы Azure.** Если вы работаете в облаке, изучите сервисы Azure, такие как Azure Functions, Azure SQL Database и Azure App Service.
- **Безопасность.** Знание методов обеспечения безопасности, включая проверку подлинности, авторизацию и защиту данных, имеет решающее значение.
- **Оптимизация производительности.** Изучите методы оптимизации, такие как профилирование, кэширование и другие способы улучшения производительности приложений.
- **Agile-методологии.** Понимание методик Agile, таких как Scrum и Канбан, улучшит вашу работу в команде и повысит эффективность разработки.
- **Гибкие навыки.** Коммуникация, решение проблем и командная работа — важные навыки для любого разработчика.

Я рассказал о многих ключевых аспектах, но не забывайте, что технологии продолжают развиваться. Быть любознательным и готовым учиться — важный навык для успешного развития. Если у вас возникли вопросы или нужно больше информации по любой из тем, смело спрашивайте!»

ChatGPT упоминает несколько тем, которые в моих книгах не рассматриваются. Вот они.

- **Azure DevOps и сервисы Azure.** В книге я решил исключить платформу Azure и прочие облачные сервисы. Я хочу, чтобы все читатели могли выполнять задания на своем локальном компьютере, не тратя деньги на облачные аккаунты. Моя цель — не подталкивать читателей к конкретному облачному провайдеру, поэтому в будущем мои книги не будут зависеть от облачных сервисов.
- **Технологии фронтенда.** HTML, CSS, JavaScript, а также фреймворки для клиентских приложений, такие как React и Angular.
- **Agile-методологии.** Эти методологии требуют отдельной книги для подробного освещения. Если получится представить их кратко и понятно, я обязательно добавлю эту информацию в следующие издания.

## Промт-инженерия для более эффективной работы с LLM

Возможно, вы уже сталкивались с термином «*промт-инженерия*». Это важный навык для профессионалов, поскольку современные задачи часто решаются гораздо эффективнее в сотрудничестве с искусственным интеллектом (ИИ).



Инструменты искусственного интеллекта, такие как GitHub Copilot, интегрируемые в редакторы кода, будут рассмотрены в главе 2.

Чтобы прояснить тему или глубже понять информацию, вы можете использовать следующие шаблоны запросов.

- Объясни [название темы] простым языком.
- Объясни мне, как новичку в [название сферы].
- Объясни мне это естественным, человеческим языком.
- Объясни мне, как будто мне десять лет.
- Напиши подробное [эссе/текст/доклад] на [название темы], добавив всю необходимую информацию.

Используйте утвердительные формулировки и избегайте отрицательных, например «не надо».

Когда ответ сложный или ИИ дает поверхностное объяснение, добавьте фразу: «Обоснуй пошагово».

Можно назначить роль для большой языковой модели, например: «Ты — эксперт-архитектор решений с многолетним опытом в области внедрения .NET».

Если вы не уверены, как сформулировать вопрос, попросите ИИ помочь. Например:

«Я хочу, чтобы ты стала моим промт-инженером. Твоя задача — помочь мне создать оптимальный запрос для моих нужд, который будет использован тобой, ChatGPT. Следуй этим шагам.

Сначала ты спросишь меня, о чем должен быть запрос. Я дам ответ, и мы будем работать вместе, чтобы улучшить его в несколько итераций.

На основе моего ответа ты создашь три раздела.

1. Пересмотренный запрос. Переформулируй запрос, чтобы он был четким, лаконичным и понятным.
2. Предложения. Поделись идеями, какие детали стоит добавить в запрос для его улучшения.
3. Вопросы. Задавай вопросы, чтобы получить от меня дополнительные детали и улучшить запрос.

Мы продолжим этот процесс: я буду предоставлять тебе дополнительные сведения, а ты будешь обновлять запрос, пока он не станет окончательным. Спасибо!»



Если ваш компьютер оборудован видеокартой Nvidia GTX, вы можете скачать бесплатное чат-приложение, которое можно настроить для общения с локальной LLM. Это приложение также поддерживает загрузку файлов, в том числе и в формате PDF. Попробуйте «скормить» ИИ PDF-файл моей книги и задать по ней вопросы. Пожалуйста, сообщите мне, что у вас получилось. Более подробную информацию вы найдете по ссылке [blogs.nvidia.com/blog/chat-with-rtx-available-now/](https://blogs.nvidia.com/blog/chat-with-rtx-available-now/).

## Использование ИИ разработчиками

Согласно исследованию Stack Overflow Survey 2023 ([survey.stackoverflow.co/2023/#section-sentiment-and-usage-ai-tools-in-the-development-process](https://survey.stackoverflow.co/2023/#section-sentiment-and-usage-ai-tools-in-the-development-process)), «70 % разработчиков уже пользуются или планируют начать использовать инструменты ИИ в текущем году. Особенно активно ИИ задействуют новички в программировании — 82 %, тогда как среди профессионалов этот показатель составляет 70 %».

## Получение помощи в Discord и других чат-форумах

Задавать вопросы на форумах по программированию и каналах Discord — это искусство, требующее не только знаний, но и навыков общения. Чтобы получить полезный ответ, важно сочетать ясность, конкретность и осведомленность о сообществе. Приведу несколько советов, которые повысят ваши шансы на информативный ответ.

- **Спрашивайте в публичном канале, а не в личном.** Не отправляйте личные сообщения и запросы на добавление в друзья с просьбами о помощи. Каждый вопрос и ответ способствуют развитию коллективного знания и могут быть полезны другим участникам. Задавая вопрос публично, вы даете возможность помочь не только автору, но и всему сообществу. Сотрудники издательства сплотили дружелюбное и умное сообщество вокруг моих книг, и оно готово поддержать вас.
- **Иследуйте вопрос перед тем, как спрашивать.** Поиск ответа самостоятельно — это важный шаг. Используйте поисковые системы, официальную документацию, а также функцию поиска на форуме или сервере Discord. Такой подход не только демонстрирует уважение к чужому времени, но и способствует вашему обучению. Не забывайте проверять раздел с обнаруженными в книге ошибками, доступный по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/errata/README.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/errata/README.md).
- **Будьте конкретны и лаконичны.** Четко формулируйте, чего именно вы хотите добиться, что уже пробовали и где возникли сложности. Конкретный и лаконичный вопрос с большей вероятностью получит быстрый ответ.
- **Укажите местонахождение ошибки в книге.** Если ваш вопрос связан с конкретной частью книги, указывайте номер страницы и название раздела, чтобы другие пользователи могли быстро найти контекст.
- **Демонстрируйте свои усилия.** Покажите, что вы сами пытались решить проблему. Так вы помогаете другим лучше понять ход ваших мыслей и выявить ошибку. Одновременно вы формируете контекст для более точного и полезного ответа.
- **Проработайте свой вопрос.** Избегайте слишком широких или расплывчатых вопросов. Пригодятся снимки экрана с ошибками или фрагменты кода (с правильным форматированием).



В последнее время я все чаще замечаю, что некоторые пользователи публикуют фотографии экрана компьютера на смартфон, что затрудняет чтение. Лучше всего копировать и вставлять текст кода или сообщения об ошибке, чтобы другие могли легко скопировать его. Ну или хотя бы сделайте скриншот с высоким разрешением вместо фотографии!

- **Правильно форматируйте код.** Используйте синтаксис Markdown для правильного форматирования кода в постах на форумах и в Discord. Так ваш код будет более читабельным. Например, оберните ключевые слова кода знаками грависа: `public void`, а блоки кода оформляйте с тремя знаками грависа, указав язык программирования:

```
```cs
using static System.Console;
WriteLine("This is C# formatted code.");
```

```



**Дельный совет:** после трех символов грависа, с которых начинается блок кода в синтаксисе Markdown, укажите код языка, например cs, csharp, js, javascript, json, html, css, cpp, xml, mermaid, python, java, ruby, go, sql, bash или shell.



**В копилку знаний:** о том, как правильно форматировать текст в сообщениях Discord, читайте в статье по адресу [support.discord.com/hc/ru/articles/210298617-Разметка-текста-101-Форматирование-в-чате-Жирный-Курсив-Подчеркнутый](https://support.discord.com/hc/ru/articles/210298617-Разметка-текста-101-Форматирование-в-чате-Жирный-Курсив-Подчеркнутый).

- **Будьте вежливы и терпеливы.** Помните, что вы обращаетесь за помощью к людям, которые добровольно жертвуют своим временем. Вежливость и терпение при ожидании ответа помогут наладить конструктивное общение. Участники канала могут находиться в разных часовых поясах, поэтому ваш вопрос может остаться незамеченным до следующего дня.
- **Будьте готовы к активному участию.** После того как вы задали вопрос, продолжайте следить за обсуждением. Возможно, вам нужно будет предоставить дополнительные уточнения. Быстрые и точные ответы на уточняющие вопросы увеличивают шансы получить полезный ответ. Когда я задаю вопрос, я ставлю будильник через три часа, чтобы проверить, не ответил ли кто-нибудь. Если ответа нет, я захожу на форум через 24 часа.

Применение этих подходов не только увеличивает шансы на получение полезного ответа, но и способствует положительному развитию сообщества, демонстрируя уважение ко времени и труду других участников.



**Дельный совет:** никогда не начинайте сообщение с простого «Привет!». Почему так, можно узнать на сайте [nohello.net/ru](http://nohello.net/ru). Также избегайте фразы «Можно спросить?» — об этом читайте на сайте [dontasktoask.com](http://dontasktoask.com).

## Настройка базы данных и проектов в рамках книги

Для работы с книгой нам понадобятся база данных и несколько проектов, которые будут использоваться в разных примерах. Чтобы все выглядело достаточно реалистично, мы создадим несколько проектов, использующих общие компоненты, такие как база данных SQL Server, библиотеки классов, модульные тесты и другие элементы.

Мы определим модель данных сущностей в виде двух библиотек классов, предназначенных для многократного использования:

- первая будет содержать описания сущностей, например `Product` и `Customer`;
- вторая — таблицы базы данных, настройки подключения по умолчанию и конфигурацию с помощью Fluent API дополнительных параметров модели.

В рамках этой структуры мы создадим три проекта:

- `Northwind.EntityModels` — библиотеку классов для сущностей, таких как `Category` и `Product`;
- `Northwind.DataContext` — библиотеку классов с контекстом данных EF Core;
- `Northwind.Tests` — проект на базе xUnit для модульного и интеграционного тестирования.

## Использование примера реляционной базы данных

Для работы с примерами в книге полезно иметь под рукой реалистичную базу данных — с умеренной сложностью структуры и достаточным объемом данных. Microsoft предлагает ряд демонстрационных баз, но большинство из них слишком громоздкие для наших целей. Вместо них мы будем использовать базу данных, созданную еще в начале 1990-х годов, — Northwind.

Давайте взглянем на диаграмму базы данных Northwind, где отображены восемь ее ключевых таблиц. Вы можете использовать диаграмму на рис. 1.2 в качестве справочного материала при написании кода и составлении SQL-запросов в ходе чтения этой книги.

Обратите внимание на следующие особенности модели данных.

- Каждая категория имеет уникальный идентификатор, название, описание и изображение. Изображение хранится в виде массива байтов в формате JPEG.
- Каждый товар содержит уникальный идентификатор, название, цену за единицу, количество единиц на складе и другие поля.

- Каждый товар связан с категорией — он хранит идентификатор соответствующей записи в таблице категорий.
- Связь между **Categories** и **Products** — один ко многим: каждая категория может включать ноль, один или несколько товаров.
- Каждый товар также связан с поставщиком и содержит идентификатор компании-поставщика.
- Для каждой детали заказа фиксируются количество и цена единицы товара.
- Каждый заказ оформляется клиентом, обрабатывается сотрудником и доставляется транспортной компанией.
- Каждый сотрудник имеет имя, адрес, контактные данные, дату рождения и дату приема на работу, а также фотографию, хранящуюся в виде массива байтов в формате JPEG.
- Кроме того, указывается идентификатор руководителя (в поле **ReportsTo**), за исключением главного руководителя, у которого это поле равно **null**. Таблица сотрудников имеет отношение один ко многим: один сотрудник может руководить несколькими другими.

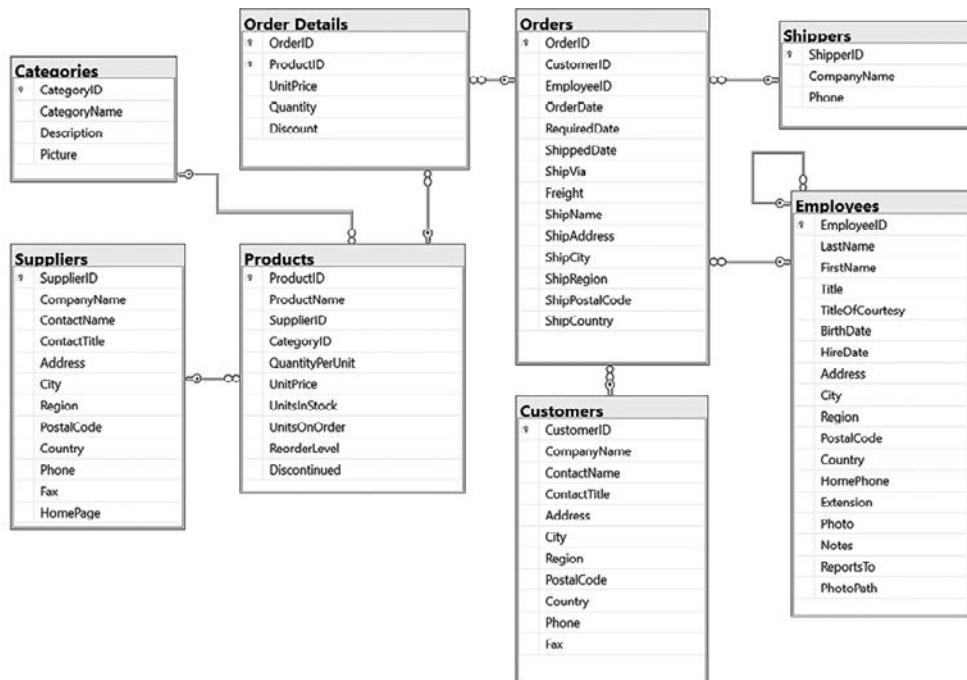


Рис. 1.2. Таблицы и взаимосвязи базы данных Northwind

## Настройка SQL Server и базы данных Northwind

Microsoft предлагает различные редакции своего популярного и функционального продукта SQL Server для систем Windows, Linux и контейнеров Docker.



Если вы работаете в операционной системе Windows, можно использовать бесплатную автономную версию SQL Server Developer Edition. Также подойдут SQL Server Express или SQL Server LocalDB, которые можно установить вместе с Visual Studio. Для инсталляции SQL Server в Windows воспользуйтесь инструкцией по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/sql-server/README.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/sql-server/README.md). Если вы хотите установить SQL Server на Linux, следуйте официальной документации на сайте [learn.microsoft.com/ru-ru/sql/linux/sql-server-linux-setup](https://learn.microsoft.com/ru-ru/sql/linux/sql-server-linux-setup).

Если у вас нет компьютера под Windows или вы хотите использовать кросс-платформенное решение, обратитесь к онлайн-разделу «Установка Azure SQL Edge в Docker» на сайте [github.com/markjprice/tools-skills-net8/blob/main/docs/sql-server/edge.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/sql-server/edge.md).

Прежде чем продолжить работу над проектом, необходимо:

- установить SQL Server (локально или через Docker);
- запустить SQL-сценарий для создания базы данных Northwind;
- убедиться, что вы можете подключиться к базе и просматривать строки, например, в таблицах `Products` и `Categories`.

В следующих двух подразделах приведены пошаговые инструкции для обоих сценариев — с использованием локального SQL Server и SQL Edge в Docker. Если SQL Server у вас уже установлен и настроен — этот раздел можно пропустить.

### Создание базы данных Northwind для локального сервера SQL Server

Чтобы создать образец базы данных Northwind для локального SQL Server, выполните следующие шаги.

1. Скачайте или клонируйте репозиторий GitHub для этой книги: [github.com/markjprice/tools-skills-net8](https://github.com/markjprice/tools-skills-net8).
2. Скопируйте файл сценария для создания базы данных из локального Git-репозитория `scripts/sql-scripts/Northwind4SQLServer.sql` в свою рабочую папку.
3. Откройте программу *SQL Server Management Studio*.
4. В появившемся окне *Connect to Server* (Подключение к серверу) введите в поле *Server name* (Имя сервера) символ `.` — он означает локальный сервер.
5. Затем нажмите кнопку *Connect* (Подключить).

6. Если вы используете именованный экземпляр, например `tools-skills-net8`, введите `.\tools-skills-net8`.
7. Выберите команду меню **File ▶ Open ▶ File** (Файл ▶ Открыть ▶ Файл).
8. Выберите файл `Northwind4SQLServer.sql` и нажмите кнопку **Open** (Открыть).
9. На панели инструментов нажмите кнопку **Execute** (Выполнить).
10. Убедитесь, что в нижней части окна появилось сообщение: `Command(s) completed successfully`.
11. На панели **Object Explorer** (Обозреватель объектов) раскройте созданную базу данных `Northwind`, затем — раздел **Tables** (Таблицы).
12. Щелкните правой кнопкой мыши на таблице **Products** и выберите пункт **Select Top 1000 Rows** (Выбрать 1000 верхних строк).
13. Убедитесь, что таблица содержит данные.
14. Закройте SQL Server Management Studio.

## Создание базы данных Northwind для SQL Edge в Docker

Чтобы развернуть образец базы данных Northwind в SQL Edge, запущенном в Docker-контейнере, выполните следующие шаги.

1. В редакторе кода откройте файл `Northwind4AzureSQLEdge.sql`.
2. Подключитесь к SQL Edge в Docker, указав следующую информацию о подключении:
  - **Data Source** (Источник данных), он же сервер: `tcp:127.0.0.1,1433`;
  - Метод проверки подлинности (**SQL Server Authentication**) (также известен как **SQL Login**) — укажите имя пользователя и пароль. В образе Azure SQL Edge уже создан пользователь `sa`, при запуске контейнера вы задавали пароль. В примере используется `s3cret-Ninja`;
  - **Database** (База данных): `master` или оставьте поле пустым — база данных `Northwind` будет создана с помощью SQL-сценария.
3. Выполните SQL-сценарий:
  - в Visual Studio щелкните на сценарии правой кнопкой мыши, выберите команду **Execute** (Выполнить) и дождитесь сообщения `Command completed successfully`;
  - в Visual Studio Code щелкните на сценарии правой кнопкой мыши, выберите команду **Execute Query** (Выполнить запрос), выберите профиль подключения `Azure SQL Edge in Docker` и дождитесь сообщения `Commands completed successfully`.
4. Обновите подключение к данным:
  - в Visual Studio на панели **Server Explorer** (Обозреватель серверов) щелкните на узле **Tables** (Таблицы) правой кнопкой мыши и выберите в контекстном меню команду **Refresh** (Обновить);
  - в Visual Studio Code щелкните на профиле подключения `Azure SQL Edge in Docker` и выберите команду **Refresh** (Обновить).

5. Разверните узел **Databases** (Базы данных), затем **Northwind**, затем **Tables** (Таблицы).
6. Убедитесь, что было создано 13 таблиц, включая **Categories**, **Customers** и **Products**. Также будут созданы десятки представлений и хранимых процедур.
7. На следующем этапе мы определим модель данных сущностей для базы данных **Northwind** в виде двух библиотек классов многократного использования.



**Дельный совет:** для моделей данных сущностей рекомендуется выделить отдельный проект библиотеки классов. Такой подход упрощает обмен данными между внутренними веб-сервисами и внешними клиентами — будь то классические, мобильные или Blazor-приложения.

## Создание библиотеки классов для моделей сущностей с помощью SQL Server

В этом разделе вы создадите модели сущностей с помощью инструмента командной строки `dotnet-ef`.

1. В редакторе кода создайте проект со следующими настройками:
  - выберите шаблон проекта **Class Library** (Библиотека классов) / `classlib`;
  - в поле **Project name** (Имя проекта) укажите значение `Northwind.EntityModels`;
  - в поле **Solution name** (Имя решения) укажите значение `Chapter01`.
2. В проекте `Northwind.EntityModels` включите опцию обработки предупреждений как ошибок и добавьте ссылки на пакеты для поставщика базы данных SQL Server и поддержки EF Core в режиме проектирования:

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

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

    <ItemGroup>
        <PackageReference>
            <Include>Microsoft.EntityFrameworkCore.SqlServer</Include>
            <Version>8.0.6</Version>
        </PackageReference>
        <PackageReference>
            <Include>Microsoft.EntityFrameworkCore.Design</Include>
            <Version>8.0.6</Version>
        </PackageReference>
        <PrivateAssets>all</PrivateAssets>
        <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
    </ItemGroup>
</Project>
```



Последние версии пакетов вы найдете на сайтах [www.nuget.org/packages/Microsoft.EntityFrameworkCore.SqlServer](http://www.nuget.org/packages/Microsoft.EntityFrameworkCore.SqlServer) и [www.nuget.org/packages/Microsoft.EntityFrameworkCore.Design](http://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Design).



Если вы не знакомы с тем, как пакеты типа Microsoft.EntityFrameworkCore.Design управляют своими активами, почитайте руководство по адресу [learn.microsoft.com/ru-ru/nuget/consume-packages/package-references-in-project-files#controlling-dependency-assets](https://learn.microsoft.com/ru-ru/nuget/consume-packages/package-references-in-project-files#controlling-dependency-assets).



**Дельный совет:** по умолчанию компилятор может выдавать предупреждения при первой сборке, но они не мешают компиляции и исчезают при повторной. Такие предупреждения обычно указывают на потенциальные проблемы в коде, и игнорировать их — плохая практика. Рекомендуется включить обработку предупреждений как ошибок, чтобы мотивировать себя их исправлять.

3. Удалите файл Class1.cs.
4. Создайте проект Northwind.EntityModels.
5. Откройте в командной строке/терминале (далее в книге используется термин «терминал») папку Northwind.EntityModels.
6. Установите инструмент dotnet-ef, если он еще не установлен:

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

7. Если инструмент уже установлен, выполните обновление:

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



На следующем этапе предполагается, что строка подключения к базе данных для локального SQL Server включает метод Windows Integrated Security. Если вы работаете с Azure SQL Edge, отредактируйте строку подключения и укажите имя пользователя и пароль.

8. В терминале выполните команду для генерации классов сущностей для всех таблиц:

```
dotnet ef dbcontext scaffold "Data Source=.;Initial  
Catalog=Northwind;Integrated Security=true;TrustServerCertificate=True;"  
Microsoft.EntityFrameworkCore.SqlServer --namespace Northwind.  
EntityModels --data-annotations
```

Обратите внимание на следующее:

- команда для запуска: `dbcontext scaffold`;
- строка подключения: `"Data Source=.;Initial Catalog=Northwind;Integrated Security=true;TrustServerCertificate=True;"`;

- поставщик базы данных: `Microsoft.EntityFrameworkCore.SqlServer`;
  - пространство имен для сгенерированных классов: `--namespace Northwind.EntityModels`;
  - использование аннотаций данных и Fluent API: `--data-annotations`.
9. В проекте появятся 28 сгенерированных классов — начиная с `AlphabeticalListOfProduct.cs` и заканчивая `Territory.cs`.
10. В верхней части файла `NorthwindContext.cs` добавьте пространство имен для работы с типами ADO.NET:

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

11. Измените метод `OnConfiguring`, чтобы он динамически устанавливал строку подключения и задавал конфиденциальные параметры через переменные среды:

```
protected override void OnConfiguring(  
    DbContextOptionsBuilder optionsBuilder)  
{  
    // Если метод еще не переопределен в клиентском проекте.  
    // Например, клиент может использовать метод AddNorthwindContext  
    // для задания строки подключения.  
  
    if (!optionsBuilder.IsConfigured)  
    {  
        SqlConnectionStringBuilder builder = new();  
  
        builder.DataSource = ".";  
        builder.InitialCatalog = "Northwind";  
        builder.TrustServerCertificate = true;  
        builder.MultipleActiveResultSets = true;  
  
        // Если используется Azure SQL Edge.  
        // builder.DataSource = "tcp:127.0.0.1,1433";  
  
        // Чтобы отказы происходили быстрее (по умолчанию 15 секунд)  
        builder.ConnectTimeout = 3;  
  
        // Если используется интегрированная проверка подлинности Windows  
        builder.IntegratedSecurity = true;  
  
        // Если используется проверка подлинности SQL Server.  
        // builder.UserID = Environment.GetEnvironmentVariable("MY_SQL_USR");  
        // builder.Password = Environment.GetEnvironmentVariable("MY_SQL_PWD");  
  
        optionsBuilder.UseSqlServer(builder.ConnectionString);  
    }  
}
```

12. Инструмент `dotnet-ef` корректно определил, что столбец `CustomerId` является первичным ключом и ограничен по длине до пяти символов. Однако, если вы хотите, чтобы значения всегда состояли только из прописных букв

западноевропейского алфавита, добавьте в определение свойства атрибут с регулярным выражением:

```
[Key]
[StringLength(5)]
[RegularExpression("[A-Z]{5}")]
public string CustomerId { get; set; } = null!;
```

## Создание библиотеки классов для контекста данных с помощью SQL Server

Далее вы перенесете контекстную модель, представляющую базу данных, в отдельную библиотеку классов.

1. В редакторе кода создайте проект со следующими настройками:
  - выберите шаблон проекта **Class Library** (Библиотека классов) / **classlib**;
  - в поле **Project name** (Имя проекта) укажите значение **Northwind.DataContext**;
  - в поле **Solution name** (Имя решения) укажите значение **Chapter01**.
2. В проекте **Northwind.DataContext** добавьте ссылку на проект **Northwind.EntityModels**, а также подключите пакет поставщика данных EF Core для SQL Server:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.6" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include=
    "..\Northwind.EntityModels\Northwind.EntityModels.csproj" />
</ItemGroup>
```



Вы можете опробовать предварительные версии EF Core 9, указав версию пакета **9.0-\***. При этом целевой фреймворк проекта должен по-прежнему оставаться **net8.0**. Используя подстановочный знак (\*), вы автоматически получите последнюю ежемесячную предварительную сборку при восстановлении пакетов. После выхода финальной версии EF Core 9 (GA) в ноябре 2024 года рекомендуется изменить версию пакета на **9.0.0** или новее. Начиная с февраля 2025 года вы сможете поступить так же с EF Core 10, указав версию **10.0-\***. Однако для этого, скорее всего, потребуется проект, ориентированный уже на **net10.0**, а значит, придется установить предварительную версию .NET 10 SDK.

3. В проекте **Northwind.DataContext** удалите файл **Class1.cs**.
4. Создайте проект **Northwind.DataContext**.
5. Переместите файл **NorthwindContext.cs** из проекта/папки **Northwind.EntityModels** в проект/папку **Northwind.DataContext**.

6. В проекте Northwind.DataContext добавьте класс NorthwindContextExtensions.cs и определите в нем метод расширения, который добавляет контекст базы данных Northwind в коллекцию зависимых сервисов:

```
using Microsoft.Data.SqlClient; // SqlConnectionStringBuilder
using Microsoft.EntityFrameworkCore; // UseSqlServer
using Microsoft.Extensions.DependencyInjection; // IServiceCollection

namespace Northwind.EntityModels;

public static class NorthwindContextExtensions
{
    /// <summary>
    /// Добавляет NorthwindContext в указанную коллекцию IServiceCollection.
    /// Используется поставщик базы данных SqlServer.
    /// </summary>
    /// <param name="services">Коллекция сервисов.</param>
    /// <param name="connectionString">Устанавливается для переопределения
    /// значения по умолчанию.</param>
    /// <returns> Коллекция IServiceCollection, которая может быть
    /// использована для добавления других сервисов.</returns>
    public static IServiceCollection AddNorthwindContext(
        this IServiceCollection services,
        string? connectionString = null)
    {
        if (connectionString == null)
        {
            SqlConnectionStringBuilder builder = new();

            builder.DataSource = ".";
            builder.InitialCatalog = "Northwind";
            builder.TrustServerCertificate = true;
            builder.MultipleActiveResultSets = true;

            // Если используется Azure SQL Edge.
            // builder.DataSource = "tcp:127.0.0.1,1433";
            // Потому что мы хотим, чтобы отказы происходили быстро.

            // По умолчанию это значение равно 15 секунд
            builder.ConnectTimeout = 3;

            // Если используется интегрированная проверка подлинности Windows
            builder.IntegratedSecurity = true;

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

            connectionString = builder.ConnectionString;
        }

        services.AddDbContext<NorthwindContext>(options =>
        {
            options.UseSqlServer(connectionString);
        });
    }
}
```

```

        // Ведение журнала в консoli при выполнении команд EF Core
        options.LogTo(Console.WriteLine,
            new[] { Microsoft.EntityFrameworkCore
                .Diagnostics.RelationalEventId.CommandExecuting });
    },
    // Регистрация с переменным временем жизни, чтобы избежать проблем
    // конкурентности с проектами Blazor Server
    contextLifetime: ServiceLifetime.Transient,
    optionsLifetime: ServiceLifetime.Transient);
}

return services;
}
}

```

- Соберите обе библиотеки классов и исправьте все ошибки компилятора.



**Дельный совет:** мы добавили дополнительный аргумент в метод AddNorthwindContext, чтобы при необходимости можно было переопределить строку подключения к базе данных SQL Server. Конфигурация становится более гибкой — например, можно загружать параметры подключения из конфигурационного файла.

## Создание тестового проекта для проверки интеграции библиотек классов

Поскольку в этой главе нет клиентского проекта с моделью EF Core, следует создать тестовый проект, чтобы корректно интегрировать контекст базы данных и модели сущностей.

- В редакторе кода создайте C#-проект xUnit Test Project (Тестовый проект xUnit) / xunit, присвойте ему имя Northwind.Tests и добавьте в решение Chapter01.
- В файле Northwind.Tests.csproj настройте обработку предупреждений как ошибок и добавьте группу элементов со ссылкой на проект Northwind.DataContext:

```

<ItemGroup>
  <ProjectReference Include=
    "..\Northwind.DataContext\Northwind.DataContext.csproj" />
</ItemGroup>

```



**Внимание!** Путь к ссылке на проект в файле проекта должен быть записан в одной строке, без символов разрыва строки.

- Соберите проект Northwind.Tests, чтобы восстановить зависимости и выполнить сборку проекта.

4. Переименуйте файл `UnitTest1.cs` в `NorthwindEntityModelsTests.cs`. При переименовании файла Visual Studio предложит также переименовать класс — согласитесь с этим предложением.
5. Если вы работаете в редакторе кода без такой поддержки (например, в Visual Studio Code), откройте файл `NorthwindEntityModelsTests.cs` и вручную переименуйте класс в `NorthwindEntityModelsTests`.
6. Затем измените содержимое класса: подключите пространство имен `Northwind.EntityModels` и добавьте несколько тестовых методов, которые будут проверять следующее:

- контекст данных успешно подключается к базе;
- в качестве поставщика БД используется SQL Server;
- первый товар в таблице назван `Chai`:

```
using Northwind.EntityModels; // Подключение классов NorthwindContext и Product

namespace Northwind.Tests;

public class NorthwindEntityModelsTests
{
    [Fact]
    public void CanConnectIsTrue()
    {
        using (NorthwindContext db = new()) // подготовка
        {
            bool canConnect = db.Database.CanConnect(); // действие
            Assert.True(canConnect); // проверка
        }
    }

    [Fact]
    public void ProviderIsSqlServer()
    {
        using (NorthwindContext db = new())
        {
            string? provider = db.Database.ProviderName;
            Assert.Equal("Microsoft.EntityFrameworkCore.SqlServer", provider);
        }
    }

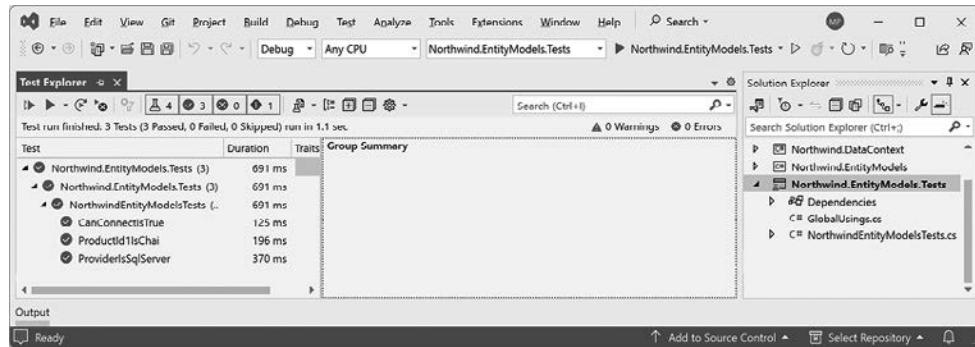
    [Fact]
    public void ProductId1IsChai()
    {
        using (NorthwindContext db = new())
        {
            Product? product1 = db?.Products?.Single(p => p.ProductId == 1);
            Assert.Equal("Chai", product1?.ProductName);
        }
    }
}
```

## Тестирование

Теперь можно запустить тесты и посмотреть результаты — в программе Visual Studio или в редакторе Visual Studio Code.

Если вы используете Visual Studio, сделайте следующее.

- На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши по проекту `Northwind.Tests` и выберите команду `Run Tests` (Запуск тестов).
- На панели Test Explorer (Обозреватель тестов) отобразятся результаты: выполнены три теста, все они прошли успешно (рис. 1.3).



**Рис. 1.3.** Все тесты пройдены успешно

Если вы используете Visual Studio Code, шаги будут такими.

- На панели TERMINAL (Терминал) проекта `Northwind.Tests` выполните команду:

```
dotnet test
```



Если вы используете C# Dev Kit, можно выполнить сборку тестового проекта, а затем запустить тесты в разделе Testing (Тестирование) панели Primary Side Bar (Боковая панель).

- В выводе терминала вы увидите, что выполнены три теста и все они завершились успешно.

## Применение .NET 9 в контексте данной книги

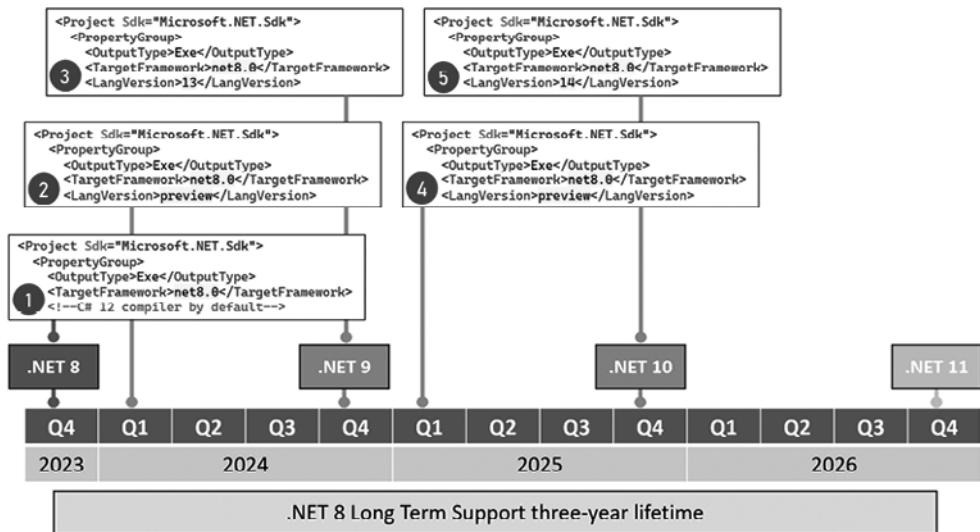
Компания Microsoft выпустила .NET 9 на конференции .NET Conf 2024, которая прошла 12 ноября 2024 года — через неделю после президентских выборов в США (5 ноября). Поскольку многие читатели будут использовать эту книгу уже после релиза .NET 9, в этом разделе объясняется, как работать с книгой в контексте новой версии платформы.

Если на вашем компьютере установлен набор инструментов SDK для .NET 9, он будет использоваться по умолчанию для всех проектов .NET — за исключением случаев, когда вы переопределяете версию в файле `global.json`. Подробнее об этом можно узнать на сайте [learn.microsoft.com/ru-ru/dotnet/core/tools/global-json](https://learn.microsoft.com/ru-ru/dotnet/core/tools/global-json).

Как показано на рис. 1.4 и описано в списке ниже, даже с установленной версией .NET 9 вы можете продолжать работать в .NET 8 — при этом можно использовать возможности компиляторов C# из новой версии платформы.

- С ноября 2023 года и далее.** Установите .NET SDK версии 8.0.100 или более новой и используйте его для сборки проектов, нацеленных на .NET 8. По умолчанию в нем используется компилятор C# 12. Регулярно обновляйте исправления SDK на компьютерах разработчиков и исправления среды выполнения .NET 8 на всех целевых системах.
- С февраля по октябрь 2024 года.** При желании устанавливайте ежемесячные предварительные версии .NET SDK 9, чтобы познакомиться с новыми возможностями языка C# и библиотеки .NET. Обратите внимание: новые возможности библиотек не будут доступны в проектах, ориентированных на .NET 8. Следите за ежемесячными обновлениями и анонсами на сайте [github.com/dotnet/core/discussions/9234](https://github.com/dotnet/core/discussions/9234).
- С ноября 2024 года и далее.** Установите .NET SDK версии 9.0.100 или новее. Вы можете использовать его для проектов, по-прежнему ориентированных на .NET 8, но уже с компилятором C# 13 и его новыми возможностями. Такой сценарий полностью поддерживается и на уровне SDK, и на уровне среды выполнения. Также вы получите доступ к возможностям EF Core 9, которая по-прежнему работает с .NET 8.
- С февраля по октябрь 2025 года.** Устанавливайте предварительные версии .NET 10, если хотите заранее изучить возможности языка C# и новых библиотек. Начинайте планировать, какие из новых функций ASP.NET Core и библиотек .NET 9/10 можно будет использовать в ваших текущих проектах на .NET 8 при подготовке к миграции.
- С ноября 2025 года и далее.** Установите .NET SDK версии 10.0.100 или более новой. Вы можете использовать его для сборки проектов, ориентированных на .NET 8 и использующих компилятор C# 14. Также вы можете начать перенос приложений на .NET 10 — это релиз с долгосрочной поддержкой (LTS). У вас будет время до ноября 2026 года, чтобы завершить миграцию до окончания срока поддержки .NET 8.

При выборе версии .NET SDK для установки помните: по умолчанию для сборки всех проектов будет использоваться последняя установленная версия SDK. Если вы установите предварительную или финальную версию .NET 9 SDK, она автоматически станет активной по умолчанию для всех новых проектов — если только вы явно не укажете использовать более раннюю, полностью поддерживаемую версию, например **8.0.100** или более новый патч.



**Рис. 1.4.** Ориентация на долгосрочную поддержку .NET 8 при использовании новейших компиляторов C#

Чтобы использовать все новые возможности C# 13 и при этом оставаться на платформе с долгосрочной поддержкой, такой как .NET 8, измените конфигурацию в своем проекте:

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

<PropertyGroup>
<OutputType>Exe</OutputType>
<TargetFramework>net8.0</TargetFramework>
<LangVersion>13</LangVersion> <!--Требуется стабильная версия SDK .NET 9-->
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
</PropertyGroup>

</Project>
```



**Дельный совет:** используйте общедоступный (GA) релиз SDK, например .NET 9, чтобы получить доступ к возможностям новых версий компилятора, сохраняя при этом ориентацию на стабильную версию платформы, такую как .NET 8 с долгосрочной поддержкой.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

## Упражнение 1.1. Дополнительные онлайн-материалы

Освойте работу с разными редакторами кода. В главе 1 книги *C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals* представлены ссылки на онлайн-разделы, демонстрирующие, как начать работу с несколькими проектами, используя разные редакторы: Visual Studio, Visual Studio Code и Rider.

Ознакомиться с этими материалами можно на сайте [github.com/markjprice/cs12dotnet8/blob/main/docs/code-editors/README.md](https://github.com/markjprice/cs12dotnet8/blob/main/docs/code-editors/README.md).

## Упражнение 1.2. Практические задания

Если у вас еще нет учетной записи в OpenAI ChatGPT, зарегистрируйтесь и попробуйте задать нейросети несколько вопросов: [chat.openai.com/](https://chat.openai.com/).

## Упражнение 1.3. Проверьте свои знания

Некоторые из этих вопросов потребуют от вас чуть больше, чем просто вспомнить материал, — нужно будет самому немного покопаться и разобраться. Я специально не даю все ответы в книге, потому что важно уметь выходить за рамки и смотреть шире.

И вот что действительно стоит запомнить: учитесь спрашивать, искать помощь и разбираться сами. Как и с рыбой — куда полезнее научиться ловить ее самому, чем каждый раз ждать, что кто-то принесет готовую.

1. Как относятся к .NET-разработчикам, которые выбрали Rider, по сравнению с теми, кто работает только в Visual Studio?
2. Какую формулировку стоит использовать, чтобы ChatGPT дал наиболее точный и полезный ответ?
3. О чём важно помнить, задавая вопрос на форуме или в чате Discord?
4. Вы установили .NET 9 SDK. Как настроить проект так, чтобы он продолжал использовать .NET 8 для надежной поддержки, но при этом позволял пользоваться новыми возможностями компилятора C# 13?
5. Зачем нужен GitHub-репозиторий, связанный с этой книгой?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 1.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-1---introducing-tools-and-skills-for-net](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-1---introducing-tools-and-skills-for-net).

## Резюме

В этой главе вы:

- познакомились с инструментами и навыками, которые будут использоваться в книге;
- настроили среду разработки;
- узнали, где искать помощь и как эффективно обращаться за ней — в том числе с помощью инструментов искусственного интеллекта, таких как ChatGPT;
- создали пример базы данных Northwind для SQL Server и несколько проектов .NET, чтобы задать модель EF Core и подготовить тесты, которые будут использоваться в следующих главах.

В следующей главе вы узнаете, как максимально эффективно использовать инструменты вашего редактора кода или IDE.

# 2

## Эффективная работа с инструментами редактора кода

В этой главе речь пойдет о том, как выйти за рамки базового использования редактора кода — управления файлами и простого редактирования — и освоить более продвинутые инструменты. Мы рассмотрим три популярных редактора кода — *интерактивные среды разработки (IDE)*: Visual Studio 2022, Visual Studio Code и JetBrains Rider. После первого упоминания я буду использовать для них сокращенные названия: Visual Studio, Code и Rider.

Читать можно только те разделы, которые относятся к вашему редактору, но я все же рекомендую заглянуть и в другие — вдруг вы захотите попробовать что-то новое или сравнить возможности.

### Общие инструменты и функции редакторов

Большинство современных редакторов кода и IDE поддерживают набор типовых инструментов. Среди них:

- средства рефакторинга;
- фрагменты кода;
- настройки и конфигурация редактора;
- интеллектуальные помощники на базе ИИ.

Давайте коротко разберемся, что представляет собой каждый из этих инструментов. А в следующих разделах вы увидите, как конкретные редакторы реализуют эти возможности на практике.

### Средства рефакторинга

*Рефакторинг* — это процесс изменения внутренней структуры существующего кода без изменения его внешнего поведения. Большинство редакторов кода и IDE предоставляют функции рефакторинга через меню *Edit* (Редактирование) и контекстные меню (вызываемые по щелчку правой кнопкой мыши).

Но зачем тратить время на переписывание кода, если он и так работает? Ответ в том, что рефакторинг — это инвестиция в будущее.

Цель — сделать код более читаемым и менее сложным для понимания. Такой подход снижает вероятность ошибок при сопровождении и упрощает внесение изменений. Даже если вы работаете в одиночку, через несколько месяцев вы скажете себе спасибо за эти усилия. Рефакторинг помогает избежать появления технического долга — и это особенно важно в долгосрочных проектах.

Вот наиболее распространенные типы рефакторинга.

- **Переименование метода/переменной.** Новое, более выразительное имя помогает лучше понять, как устроен метод или что представляет собой переменная. В результате улучшается читаемость, а код становится почти самодокументируемым.
- **Извлечение метода.** При первоначальной реализации метод может получиться слишком длинным. Часть такого кода можно вынести в отдельный метод с понятным именем, описывающим его назначение. Код становится чище, удобнее для чтения и лучше подходит для повторного использования.
- **Перемещение членов.** Иногда имеет смысл перенести метод, свойство или поле из одного класса в другой — туда, где им логически самое место.
- **Инкапсуляция поля.** Закрытие поля (присвоение ему модификатора `private`) и создание для него свойства с геттером и сеттером позволяет контролировать доступ к нему и управлять моментом изменения.
- **Извлечение интерфейса.** Когда несколько классов реализуют одинаковое поведение, но используются по-разному, имеет смысл выделить общее подмножество членов в интерфейс. Такой подход повышает уровень абстракции и расширяет возможности повторного использования.
- **Упорядочение параметров в методе.** Методы могут вызываться в разных частях кода. Если вы добавляете, удаляете или меняете порядок параметров, соответствующий рефакторинг поможет автоматически обновить все вызовы, чтобы они остались корректными.

Рефакторинг — один из ключевых навыков в разработке ПО, и он становится особенно важным по мере роста и усложнения кодовой базы. Регулярный рефакторинг помогает поддерживать код в чистом, понятном и устойчивом состоянии на протяжении всего жизненного цикла проекта.

Теперь давайте посмотрим, как можно ускорить работу, повторно используя код, который вы часто набираете.

## Фрагменты кода

*Фрагменты* — это небольшие, часто многократно используемые части кода, которые выполняют типовые задачи. Они могут представлять собой одну функцию, несколько строк или даже шаблон целого класса. Такие фрагменты можно создавать

для любого языка программирования. Часто фрагменты сохраняются в виде кросс-платформенных файлов, например в формате XML или JSON, со встроенным кодом на нужном языке. Благодаря этому можно быстро вставлять и повторно использовать готовые решения, ускоряя разработку и снижая объем рутинного ввода.

Среди преимуществ использования фрагментов кода можно назвать следующие.

- **Экономия времени.** Вместо того чтобы начинать с чистого листа, можно вставить готовый фрагмент, выполняющий нужную задачу. Фрагменты особенно полезны при обучении или демонстрации новых возможностей языка — их часто используют преподаватели и авторы курсов, чтобы наглядно показать, как работает та или иная концепция.
- **Обучение и обмен знаниями.** Фрагменты — отличный инструмент для обучения. Они помогают понять, как реализовать определенную функцию или использовать конкретную технологию. Некоторые сторонние библиотеки поставляются с набором фрагментов, содержащих типовой стартовый код для работы с пакетом. Для начинающих разработчиков это практические примеры, помогающие разобраться в абстрактных концепциях, а для опытных — способ изучить лучшие практики и новые подходы.
- **Последовательность и совместная работа.** Стандартизованные фрагменты помогают команде реализовывать однотипные задачи единообразно — как в рамках одного проекта, так и между разными проектами. Такой подход позволяет поддерживать качество кода и снижать риск ошибок. Участники команды могут делиться своими фрагментами, формируя общую библиотеку, основанную на коллективном опыте. Благодаря этому сохраняется единый стиль программирования, ускоряется работа и повышается согласованность решений.
- **Документация и ссылки.** Правильно оформленные фрагменты могут выполнять роль технической документации. Если фрагмент кода снабжен комментариями и описанием, он не только выполняет нужную функцию, но и объясняет, как она работает. Такой подход облегчает другим разработчикам понимание и повторное использование кода.

Фрагменты кода — это практичный и удобный инструмент, который дает множество преимуществ: экономию времени, повышение единообразия в коде и поддержку обучения. Они ускоряют разработку и способствуют формированию культуры совместной работы в команде.

Теперь посмотрим, как можно обеспечить единый стиль программирования в команде .NET-разработчиков, работающих над одним проектом в разных редакторах кода.

## Настройка редактора кода

Существует стандартный формат файла для настройки стиля программирования в редакторе кода — файл `.editorconfig`. Большинство популярных редакторов поддерживают этот формат и автоматически применяют заданные в нем правила оформления кода.

Файлы `.editorconfig` легко читаются, их можно редактировать вручную или через интерфейс самого редактора. Они отлично подходят для совместной работы, так как легко отслеживаются и хранятся в системах контроля версий, таких как Git.

При открытии проекта редактор кода будет искать файл `.editorconfig` в текущем каталоге и выше по иерархии, пока не достигнет корня или не встретит файл, в котором явно указано `root=true`. Вот пример такой конфигурации:

```
# Файл .editorconfig на верхнем уровне
root = true
```

Комментарии в файле `.editorconfig` начинаются с символа `#`.

Свойства задаются в формате `ключ=значение` — каждое на отдельной строке.

Правила можно группировать по секциям. Каждая секция начинается с шаблона имени файла в квадратных скобках, например `[*.cs]`. Благодаря этому можно применять разные настройки к разным типам файлов.

Например, чтобы задать правила только для файлов C#, можно использовать следующую конфигурацию:

```
# Правило отступа из двух пробелов, применяемое только к файлам на C#
[*.cs]
indent_style = space
indent_size = 2
```

Файл `.editorconfig` читается сверху вниз, и правила применяются в порядке их появления. Это значит, что более поздние правила могут переопределять ранее заданные. Кроме того, если в иерархии каталогов присутствует несколько файлов `.editorconfig`, приоритет будет у того, который находится ближе к файлу с исходным кодом.

Наиболее часто используемые свойства, которые можно указать в файле `.editorconfig`, приведены в табл. 2.1.

**Таблица 2.1.** Часто используемые параметры в файле `.editorconfig`

Свойство	Допустимые значения	Описание
<code>root</code>	<code>true, false</code>	Если указано <code>true</code> , поиск других файлов <code>.editorconfig</code> прекращается
<code>charset</code>	<code>latin1, utf-8</code> и др.	Определяет набор символов, используемый в файле
<code>indent_style</code>	<code>tab, space</code>	Указывает символ отступа: табуляция или пробел. Регистр не учитывается
<code>tab_width</code>	<code>&lt;целое_число&gt;</code>	Количество пробелов, соответствующее одному символу табуляции
<code>indent_size</code>	<code>tab, &lt;целое_число&gt;</code>	Размер отступа в символах

<b>Свойство</b>	<b>Допустимые значения</b>	<b>Описание</b>
<code>end_of_line</code>	<code>lf, cr, crlf</code>	Определяет стиль переноса строк
<code>trim_trailing_whitespace</code>	<code>true, false</code>	Удаляет пробелы в конце строк при сохранении
<code>insert_final_newline</code>	<code>true, false</code>	Гарантирует наличие пустой строки в конце файла при сохранении



**В копилку знаний:** подробнее о стандарте файла `.editorconfig` можно узнать на сайте [editorconfig.org](http://editorconfig.org).

Файл `.editorconfig` можно расширять — большинство редакторов кода поддерживают дополнительные параметры, которые они умеют интерпретировать. Например, в Visual Studio можно включить или отключить предупреждение об удалении неиспользуемых директив `using` (импортов пространств имен). Это предупреждение имеет код `IDE0005`, и его можно отключить с помощью следующей настройки:

```
[*.cs]
dotnet_diagnostic.IDE0005.severity = none
```

Подробнее о том, как файлы `.editorconfig` используются в проектах .NET для настройки анализа кода, можно прочитать на сайте [learn.microsoft.com/ru-ru/dotnet/fundamentals/code-analysis/configuration-files](https://learn.microsoft.com/ru-ru/dotnet/fundamentals/code-analysis/configuration-files).

## Интеллектуальные помощники в редакторах кода

Современные редакторы кода включают в себя помощников с искусственным интеллектом. Такие инструменты могут взять на себя часть рутинной или специализированной работы.

- **Повышение качества и безопасности кода.** ИИ может выявлять потенциально небезопасные шаблоны программирования или некорректное использование API.
- **Ответы на вопросы о проекте.** ИИ может отвечать как на общие вопросы по программированию, так и на специфические — по вашей кодовой базе. Когда вы присоединяетесь к новому проекту, попросите ИИ провести обзор: он может объяснить структуру проекта, назначение компонентов и даже расшифровать сложное регулярное выражение.
- **Завершение кода и генерация инструкций.** Во время ввода кода или комментариев ИИ может предложить варианты автодополнения. Он умеет преобразовывать естественный язык в фрагменты кода, соответствующие стилю проекта.
- **Автоматизация документации и тестирования.** ИИ может сгенерировать комментарий к Git-запросу на извлечение (pull request) на основе внесенных изменений, а также написать модульные тесты, опираясь на сигнатуру метода и его логику.

Теперь, когда вы познакомились с основными инструментами и возможностями, которые встречаются в большинстве редакторов кода, рассмотрим их на конкретных примерах. Начнем с Visual Studio — самого популярного редактора среди разработчиков .NET.

## Инструменты в Visual Studio 2022

Visual Studio содержит множество встроенных инструментов, которые помогают реализовать концепции, описанные ранее в этой главе.



### Visual Studio vNext

На момент написания книги актуальной версией Visual Studio была 17.10, обозначаемая как Visual Studio 2022. Ожидается, что следующая основная версия — 18.0 — получит название Visual Studio 2025 и выйдет в ноябре 2025 года, уже после публикации книги. Версия Visual Studio 2025, скорее всего, сохранит большинство возможностей версии 2022, хотя в пользовательском интерфейсе могут появиться изменения. В этой книге ссылки на Visual Studio по умолчанию относятся к версии 2022. Если необходимо перейти по ссылке к предыдущей версии документации, добавьте в конец URL-адреса параметр ?view=vs-2022.

## Функционал рефакторинга

В программе Visual Studio предусмотрены десятки функций рефакторинга, которые помогают модифицировать код без изменения его поведения. Благодаря этому можно упростить сопровождение, сделать код более понятным и подготовить его к развитию.

Наиболее часто используемые функции доступны через меню **Edit ▶ Refactor** (Редактирование ▶ Рефакторинг), где можно выбрать одну из следующих команд.

- **Rename** (Переименовать) (F2) — безопасно переименовывает переменную, тип или член, а также все места их использования.
- **Encapsulate Field** (Инкапсулировать поле) (Ctrl+R, E) — преобразует поле в свойство и обновляет весь код, чтобы использовать это свойство.
- **Extract Interface** (Извлечь интерфейс) (Ctrl+R, I) — создает интерфейс на основе выбранных членов класса, структуры или уже существующего интерфейса.
- **Remove Parameter** (Удалить параметр) (Ctrl+R, V) — удаляет параметр из сигнатуры метода.
- **Reorder Parameters** (Упорядочить параметры) (Ctrl+R, O) — изменяет порядок параметров в методе.

Более специализированные операции рефакторинга зависят от контекста. Для их активации установите текстовый курсор в нужную часть кода, щелкните правой кнопкой мыши и в контекстном меню выберите команду **Quick Actions and Refactorings** (Быстрые действия и рефакторинг).

Посмотрим, как некоторые из этих функций реализуются на практике.

1. В Visual Studio создайте проект Console App (Консольное приложение) / `console`, присвойте ему имя `RefactoringDemos` и добавьте в решение `Chapter02`.
2. В проекте `RefactoringDemos` выполните глобальный статический импорт класса `System.Console`:

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

3. В проекте `RefactoringDemos` создайте файл класса `Product.cs`.
4. В созданном файле удалите все содержимое и добавьте выражения для определения класса `Product` с полем `description`:

```
public class Product
{
    public string? description;
}
```

5. В файле `Program.cs` удалите все содержимое и создайте класс `Product`, задав в нем описание через поле `description`:

```
Product product = new()
{
    description = "Seafood"
};

WriteLine(product.description);
```

6. В файле `Product.cs` щелкните на выражении, определяющем поле `description`, и воспользуйтесь быстрыми действиями: щелкните на значке  Quick Actions (Быстрые действия) либо нажмите сочетание клавиш `Alt+Enter` или `Ctrl+. (точка)`. Обратите внимание на доступные варианты рефакторинга (рис. 2.1).

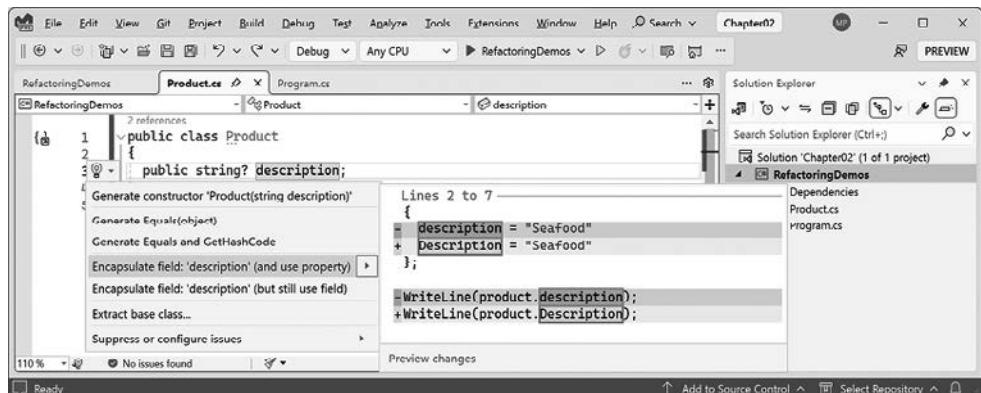


Рис. 2.1. Применение рефакторинга к полю в классе



Список доступных действий зависит от того, где именно находится текстовый курсор при активации команды Quick Actions and Refactorings (Быстрые действия и рефакторинг). Например, если курсор установлен на ключевом слове `public`, доступных вариантов будет меньше.

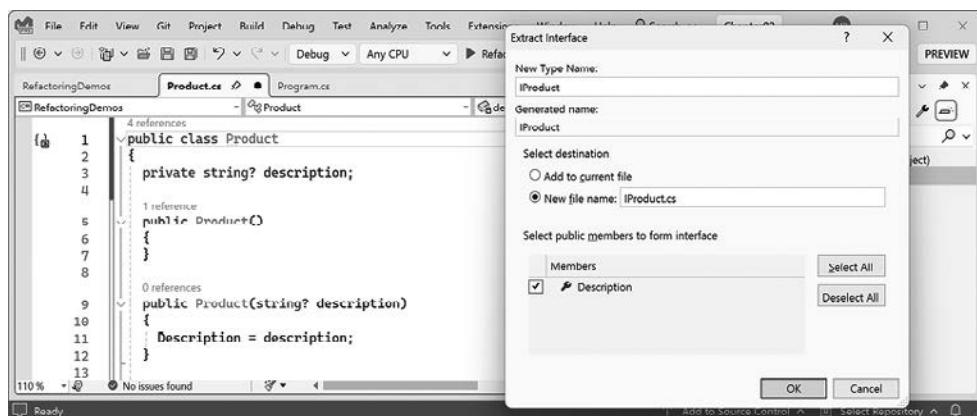
7. Выберите пункт `Encapsulate field: 'description' (and use property)` (Инкапсулировать поле `description` и использовать свойство). Обратите внимание, что поле стало `private`, а для доступа к нему было создано свойство с модификатором `public`. В файле `Program.cs` имя поля изменится на `Description`, чтобы соответствовать имени свойства.
8. В файле `Product.cs` щелкните кнопкой мыши на операторе, определяющем класс `Product`, затем щелкните правой кнопкой мыши и в контекстном меню выберите команду `Quick Actions and Refactorings` (Быстрые действия и рефакторинг) или нажмите сочетание клавиш `Alt+Enter` либо `Ctrl+.` и выберите пункт `Generate constructor` (Создать конструктор).
9. В диалоговом окне `Pick members` (Выбор членов класса) снимите флажок у поля `description`, оставив только свойство `Description`, затем нажмите кнопку `OK`.
10. В конструктор будет добавлен параметр, инициализирующий свойство `Description`:

```
public Product(string? description)
{
    Description = description;
}
```

11. В файле `Product.cs` щелкните кнопкой мыши на выражении, определяющем класс `Product`, затем щелкните правой кнопкой мыши и в контекстном меню выберите команду `Quick Actions and Refactorings` (Быстрые действия и рефакторинг) или нажмите сочетание клавиш `Alt+Enter` либо `Ctrl+.` и выберите пункт `Generate constructor 'Product()'` (Создать конструктор `Product()`). Обратите внимание, что добавляется конструктор по умолчанию без параметров.
12. В файле `Product.cs` щелкните на выражении, определяющем класс `Product`, затем щелкните правой кнопкой мыши и в контекстном меню выберите команду `Quick Actions and Refactorings` (Быстрые действия и рефакторинг) или нажмите сочетание клавиш `Alt+Enter` либо `Ctrl+.` и выберите пункт `Extract interface` (Извлечь интерфейс). Обратите внимание на параметры в открывшемся диалоговом окне (рис. 2.2).
13. Нажмите кнопку `OK`.
14. В файле `Product.cs` будет определен интерфейс на основе членов класса `Product`:

```
public interface IProduct
{
    string? Description { get; set; }
}
```

15. А класс `Product` теперь реализует интерфейс `IProduct`.



**Рис. 2.2.** Параметры диалогового окна Extract interface

Теперь рассмотрим другие полезные возможности рефакторинга.

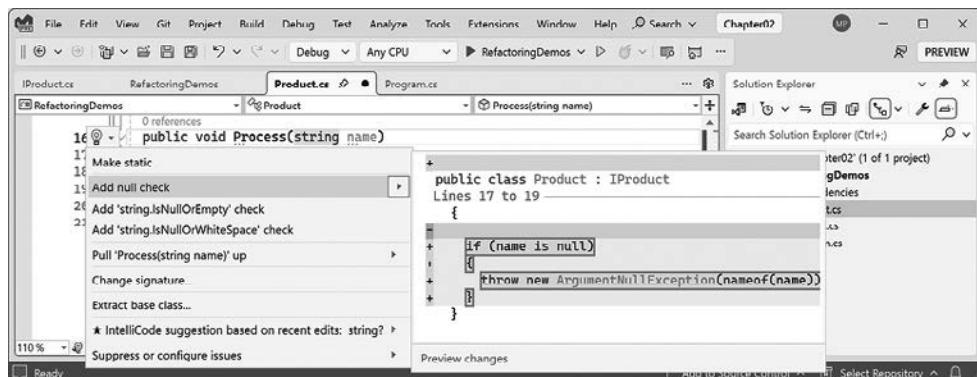
## Добавление проверки параметров метода

Приветствуется добавление инструкций `if`, проверяющих параметры метода, допускающие значение `null`. Особенно это касается строковых аргументов. Такой рефакторинг можно выполнить автоматически. Давайте посмотрим, как это работает.

1. В файле `Product.cs` добавьте следующий метод:

```
public void Process(string name)
{
}
```

2. Щелкните на параметре `name`, воспользуйтесь быстрыми действиями (`Ctrl+.` или `Alt+Enter`) и обратите внимание на вариант добавления проверки на `null`, пустую строку или пробельные символы (рис. 2.3).



**Рис. 2.3.** Добавление проверок параметров метода

3. Выберите пункт Add 'string.IsNullOrEmpty' check (Добавить проверку `string.IsNullOrEmpty`), после чего в метод будет добавлено соответствующее выражение:

```
if (string.IsNullOrEmpty(name))
{
    throw new ArgumentException($"'{nameof(name)}' cannot be null or whitespace.",
        nameof(name));
}
```

Если в методе есть несколько параметров, Visual Studio предложит вариант Add null checks for all parameters (Добавить проверку null для всех параметров). Надеюсь, что в будущих версиях среды разработки будет использовать более лаконичные защитные конструкции вроде `ArgumentException.ThrowIfNullOrEmpty(name)`, если проект ориентирован на версию .NET, в которой они доступны.

## Рефакторинг параметров метода

Теперь рассмотрим, как изменить сигнатуру метода с помощью рефакторинга.

1. В файле `Product.cs` в методе `Process` щелкните кнопкой мыши на сигнатуре метода, затем щелкните правой кнопкой мыши и в контекстном меню выберите команду Quick Actions and Refactorings (Быстрые действия и рефакторинг), а затем — пункт Change signature (Изменить сигнатуру).
2. В открывшемся диалоговом окне `Change Signature` (Изменить сигнатуру) можно добавлять, удалять и изменять порядок параметров (рис. 2.4).
3. Попробуйте добавить новый параметр, чтобы посмотреть, какие варианты доступны.
4. Нажмите кнопку `Cancel` (Отмена), чтобы закрыть окно без внесения изменений.

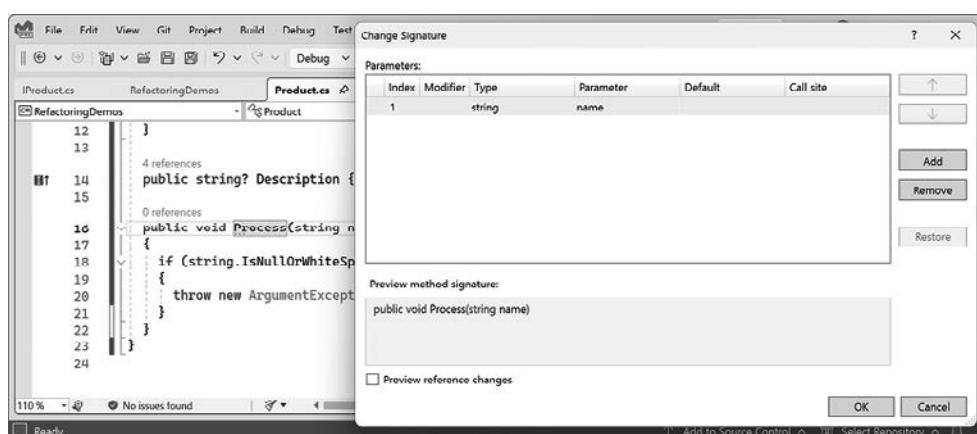
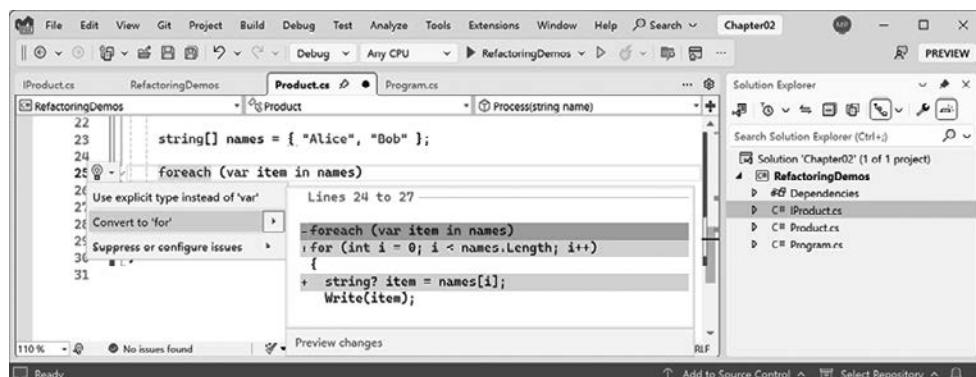


Рис. 2.4. Изменение сигнатуры метода

## Преобразование foreach в for и обратно

Инструкция `foreach` — это простой и удобный способ перебора массива или коллекции. Однако бывают случаи, когда нужно знать индекс текущего элемента. В таких ситуациях `foreach` можно заменить на `for`, в которой используется целочисленная переменная для перебора элементов по индексам. Также возможна и обратная замена — с `for` на `foreach`.

Чтобы применить этот рефакторинг, щелкните на ключевом слове `foreach` или `for`, затем щелкните правой кнопкой мыши и в контекстном меню выберите команду `Quick Actions and Refactorings` (Быстрые действия и рефакторинг). Теперь выберите пункт `Convert to 'for'` (Преобразовать в `for`) или `Convert to 'foreach'` (Преобразовать в `foreach`) (рис. 2.5).



**Рис. 2.5.** Преобразование инструкции `foreach` в инструкцию `for` с индексом

## Упрощение LINQ-операторов

Одна из распространенных ошибок при использовании LINQ — вызов метода `Where` для фильтрации, а затем отдельный вызов метода `Single`, `SingleOrDefault`, `First` или `FirstOrDefault`. Многие разработчики не знают, что эти методы принимают делегат — то есть можно передать метод или лямбда-выражение прямо в них для фильтрации. Например, следующий код...

```
sequence.Where(item => item.ProductId == 27).Single();
```

...можно упростить до следующего выражения:

```
sequence.Single(item => item.ProductId == 27);
```

Чтобы применить этот рефакторинг, выделите LINQ-выражение, затем щелкните правой кнопкой мыши и в контекстном меню выберите команду `Quick Actions and Refactorings` (Быстрые действия и рефакторинг). Теперь выберите пункт `Simplify LINQ expression` (Упростить выражение LINQ).

## Выравнивание элементов кода

Один из самых полезных рефакторингов — это набор действий, которые позволяют обернуть длинное выражение, добавить отступы и выровнять его для улучшения читаемости.

Наиболее распространенные случаи, где такие преобразования особенно полезны:

- методы с большим количеством параметров;
- выражения с множественными вызовами методов;
- двоичные операторы с несколькими operandами.

Каждое из таких выражений можно переформатировать, чтобы код стало легче читать. Это особенно важно, когда горизонтальное пространство ограничено — например, при верстке книги.

В следующем методе показаны три типичных случая, где выравнивание помогает сделать код понятнее...

```
public void Process(int number, string[] names, DateTime when, double height)
{
    if (names.Length > 10 && number < 0 && when.IsWeekend && height.IsNaN)
    {
        var query = names.Where(name.StartsWith("M")).OrderBy(name => name).
            Select(name => name.Length);
        ...
    }
}
```

Чтобы изменить формат, выделите соответствующее выражение, затем щелкните правой кнопкой мыши и в контекстном меню выберите команду **Quick Actions and Refactorings** (Быстрые действия и рефакторинг). Далее выберите нужный пункт, например:

- **Wrap expression** (Обернуть выражение);
- **Wrap every parameter** (Обернуть каждый параметр);
- **Wrap call chain** (Обернуть цепочку вызовов);
- **Wrap and align call chain** (Обернуть и выровнять цепочку вызовов).

```
public void Process(int number,
                    string[] names,
                    DateTime when,
                    double height)
{
    if (names.Length > 10
        && number < 0
        && when.IsWeekend
        && height.IsNaN)
    {
        var query = names.Where(name.StartsWith("M"))
    }
}
```

```

.OrderBy(name => name)
.Select(name => name.Length);
...
}
}

```



**В копилку знаний:** полная официальная документация по функциям рефакторинга в Visual Studio доступна на сайте [learn.microsoft.com/ru-ru/visualstudio/ide/refactoring-in-visual-studio](https://learn.microsoft.com/ru-ru/visualstudio/ide/refactoring-in-visual-studio).

## Рефакторинг с использованием первичных конструкторов

Одна из новых возможностей языка C#, представленных в версии 12 в составе .NET 8, — это первичные конструкторы для классов. Ранее такие конструкторы были доступны только для типов `record`.

Команда разработчиков C# из корпорации Microsoft стремится поощрять использование новых возможностей языка, включая первичные конструкторы. Первая статья в блоге, посвященная этой теме, рассказывает о том, как рефакторить код с использованием этой конструкции. Прочитать ее можно по адресу [devblogs.microsoft.com/dotnet/csharp-primary-constructors-refactoring/](https://devblogs.microsoft.com/dotnet/csharp-primary-constructors-refactoring/).



Хотя первичные конструкторы хорошо работают в типах `record`, я не сторонник их применения в типах `class`. Похоже, так думают большинство .NET-разработчиков — об этом можно судить по довольно критичному разделу комментариев к статье.

Тем не менее рекомендую все же прочитать эту статью: вы узнаете, как работает механизм первичных конструкторов, а заодно увидите разные точки зрения, которые помогут глубже понять язык. Ведь именно споры и обсуждения делают развитие языка таким интересным!

## Фрагменты кода

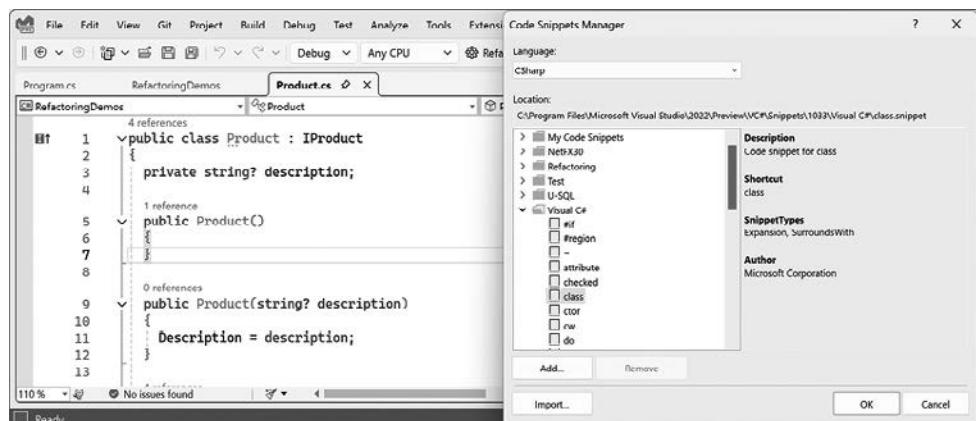
В Visual Studio можно просмотреть доступные в данный момент фрагменты кода с помощью отдельного окна. Посмотрим, как это сделать.

- Выберите команду меню `Tools > Code Snippets Manager` (Средства > Диспетчер фрагментов кода) или нажмите сочетание клавиш `Ctrl+K, Ctrl+B`.
- В поле `Language` (Язык) выберите значение `CSharp`, раскройте папку `Visual C#` в дереве и выберите пункт `class` (рис. 2.6).

Дополнительные сведения о фрагментах кода.

- Файлы фрагментов кода имеют расширение `.snippet`.
- Встроенные фрагменты для языка C# хранятся в папке `C:\Program Files\Microsoft Visual Studio\2022\Community\VC#\Snippets\1033\Visual C#` (номер культуры 1033 соответствует американскому английскому языку).

- Пользовательские фрагменты C# можно добавлять в папку C:\Users\<ваше\_имя>\OneDrive\Documents\Visual Studio 2022\Code Snippets\Visual C#\My Code Snippets.
- У фрагментов есть программные ярлыки. Их можно ввести в редакторе кода, а затем развернуть, нажав клавишу Tab.
- Фрагменты бывают типов Expansion и SurroundsWith:
  - ◆ Expansion — например, ярлык cw, который разворачивается в Console.WriteLine(...);
  - ◆ SurroundsWith — например, if, который обворачивает выделенные строки кода конструкцией условия.
- С помощью кнопок Add (Добавить) и Remove (Удалить) можно управлять папками, содержащими фрагменты.
- Кнопка Import (Импорт) позволяет импортировать файл с фрагментами в текущую папку.



**Рис. 2.6.** Просмотр фрагментов кода C# в окне Code Snippets Manager

3. В поле Location (Расположение) выделите полный путь к файлу фрагмента class и скопируйте его в буфер обмена.
4. Выберите команду меню File ▶ Open ▶ File (Файл ▶ Открыть ▶ Файл) или нажмите сочетание клавиш Ctrl+O, вставьте путь в поле File name (Имя файла), затем нажмите кнопку Open (Открыть).
5. Файлы с фрагментами имеют расширение .snippet и оформлены в формате XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<CodeSnippets xmlns=
  "http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
```

```

<CodeSnippet Format="1.0.0">
  <Header>
    <Title>class</Title>
    <Shortcut>class</Shortcut>
    <Description>Code snippet for class</Description>
    <Author>Microsoft Corporation</Author>
    <SnippetTypes>
      <SnippetType>Expansion</SnippetType>
      <SnippetType>SurroundsWith</SnippetType>
    </SnippetTypes>
  </Header>
  <Snippet>
    <Declarations>
      <Literal>
        <ID>name</ID>
        <ToolTip>Class name</ToolTip>
        <Default>MyClass</Default>
      </Literal>
    </Declarations>
    <Code Language="csharp"><![CDATA[class $name$<
{
  $selected$$end$<
}]]>
      </Code>
    </Snippet>
  </CodeSnippet>
</CodeSnippets>

```

Теперь о ключевых элементах файла.

- Код, который будет вставлен, находится внутри блока `<![CDATA[ ... ]]>`. Все, что вы видите между этими выражениями, — это исходный код фрагмента.
- Переменные во фрагменте заключаются в символы `$` с обеих сторон. Например:
  - `$selected$` — представляет текущий выделенный код при использовании фрагмента `SurroundsWith`;
  - `$end$` — позиция текстового курсора после вставки фрагмента. Например, при вставке фрагмента `cw` курсор окажется между скобками вызова `WriteLine()`.
- Пользовательские переменные описываются в разделе `<Declarations>`. Например, `$name$`. После вставки фрагмента эти переменные можно заменить на нужные значения. Если переменных несколько, можно нажимать клавишу `Tab`, чтобы переходить между ними.

## Схема фрагментов кода

Файлы фрагментов кода оформляются в виде XML-документов, структура которых должна соответствовать заранее определенной схеме. В табл. 2.2 приведены основные допустимые элементы, атрибуты и значения.

**Таблица 2.2.** XML-элементы схемы фрагментов кода

Элемент схемы	Описание	Требуется
<CodeSnippets>	Корневой элемент в XML-файле	Да
<CodeSnippet>	Родительский элемент для <Header> и <Snippet>	Да
<Header>	Содержит общую информацию, такую как автор и имя, отображаемую в окне Code Snippets Manager (Диспетчер фрагментов кода). Дочерние элементы: <Author>, <Description>, <HelpUrl>, <SnippetTypes>, <Keywords> и <Shortcut>. Обязателен только <Title>	Да
<Snippet>	Содержит код, который будет вставлен	Да
<Code>	Содержит фактический код фрагмента. \$end\$ и \$selected\$ — специальные маркеры. Остальные переменные должны быть определены как <Literal> или <Object>	Да
Атрибут Language	Устанавливает язык для кода. Возможные значения: CSharp, VB, CPP, XAML, XML, JavaScript, TypeScript, SQL, HTML	Да
Атрибут Kind	Ограничивает область использования фрагмента (например, только в теле метода ( <i>method body</i> )). Значения: <i>method body</i> , <i>method decl</i> , <i>type decl</i> , <i>file</i> , <i>any</i> и др.	Нет
Атрибут Delimiter	Символы-разделители используются для обозначения заменяемых частей кода. По умолчанию таким разделителем служит символ \$, окружающий имена переменных. Если вам нужно вставить в код сам символ \$, например при использовании буквальных строк в C#, вы можете переопределить разделитель, указав другой символ, например ~	Нет
<Declarations>	Определяет литералы и объекты, которые можно редактировать после вставки (например, имя переменной в <i>foreach</i> )	Нет
<Literal>	Описывает фиксированное значение, например <i>int max = 100;</i> в операторе <i>for</i>	Нет
<Object>	Задает имена объектов, которые используются во фрагменте	Нет
<ID>	Идентификатор литерала или объекта	Нет
<Default>	Значение по умолчанию для литерала или объекта	Нет



**В копилку знаний:** схема фрагментов кода задокументирована по адресу [learn.microsoft.com/ru-ru/visualstudio/ide/code-snippets-schema-reference](https://learn.microsoft.com/ru-ru/visualstudio/ide/code-snippets-schema-reference).

Теперь посмотрим, как создать собственный фрагмент кода и подготовить его для использования.

## Создание и импорт фрагментов кода

Создание фрагмента кода состоит из нескольких шагов.

1. Создайте XML-файл с расширением `.snippet`.
2. Заполните обязательные элементы, например `<Title>`.
3. Добавьте код, указав литералы и объекты, которые можно будет заменить при вставке.

Теперь посмотрим, как это реализуется на практике.

1. В Visual Studio создайте проект `Console App (Консольное приложение) / console`. Присвойте ему имя `CodeSnippetDemos` и добавьте в решение `Chapter02`.
2. В проекте `CodeSnippetDemos` создайте XML-файл с именем `CommonImports.snippet`.
3. В файле `CommonImports.snippet` удалите все содержимое и добавьте выражения, чтобы сформировать минимальный шаблон фрагмента кода:

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets xmlns=
    "http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
    <CodeSnippet Format="1.0.0">
        <Header>
            <Title></Title>
        </Header>
        <Snippet>
            <Code Language="">
                <![CDATA[[]]>
            </Code>
        </Snippet>
    </CodeSnippet>
</CodeSnippets>
```



**Дельный совет:** приведенный шаблон можно использовать в качестве основы для будущих фрагментов кода — он содержит все необходимое, чтобы начать.

4. В элемент `<Title>` добавьте текст `Statically import common types globally.`
5. В атрибуте `Language` присвойте значение `XML`.
6. Между символами `[` и `]` в элементе `<![CDATA[[]]>` вставьте код для импорта нескольких часто используемых типов:

```
<![CDATA[<ItemGroup>
    <Using Include="System.Console" Static="true" />
    <Using Include="System.Environment" Static="true" />
    <Using Include="System.IO.Path" Static="true" />
</ItemGroup>]]>
```

7. Сохраните файл.
  8. Выберите команду меню Tools ▶ Code Snippets Manager (Средства ▶ Диспетчер фрагментов кода).
  9. Нажмите кнопку Import (Импорт), перейдите в папку \tools-skills-net8\ Chapter02\CodeSnippetDemos\, выберите файл CommonImports.snippet, затем нажмите кнопку Open (Открыть).
  10. В диалоговом окне Import Code Snippet (Импорт фрагмента кода) выберите папку My Xml Snippets (Мои фрагменты XML) и нажмите кнопку Finish (Завершить).
  11. В окне Code Snippets Manager (Диспетчер фрагментов кода) нажмите кнопку OK.
  12. В файле CodeSnippetDemos.csproj проекта установите точку вставки на пустую строку непосредственно перед элементом </Project>.
  13. Вставьте фрагмент кода одним из следующих способов:
    - выберите команду меню Edit ▶ IntelliSense ▶ Insert Snippet (Редактирование ▶ IntelliSense ▶ Вставить фрагмент);
    - щелкните правой кнопкой мыши и в контекстном меню выберите команду Snippet ▶ Insert Snippet (Фрагмент кода ▶ Вставить фрагмент кода);
    - нажмите сочетание клавиш Ctrl+K, X.
  14. Выберите вариант My Xml Snippets (Мои фрагменты XML), затем выберите команду Static import common types globally (Статический импорт общих типов глобально). После этого фрагмент кода будет вставлен в указанное место.
  15. В файле проекта удалите все содержимое, затем добавьте выражение для вывода текущего каталога с использованием статически импортируемых типов:  
`WriteLine(CurrentDirectory);`
  16. В проекте CodeSnippetDemos создайте XML-файл с именем ConfigureConsole.snippet.
  17. В файле ConfigureConsole.snippet удалите все содержимое и добавьте выражения для определения фрагмента кода, который настраивает консоль:
- ```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets xmlns=
    "http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
    <CodeSnippet Format="1.0.0">
        <Header>
            <Title>Configure console for culture</Title>
            <Description>Define a static method for Program to configure the culture
                for the console.</Description>
            <Author>Mark J. Price</Author>
            <Shortcut>ccc</Shortcut>
        </Header>
        <Snippet>
            <Imports>
                <Import>
                    <Namespace>System.Globalization</Namespace>
                </Import>
            </Imports>
```

```

<Declarations>
  <Literal>
    <ID>defaultCulture</ID>
    <Default>en-US</Default>
    <ToolTip>The default culture when none is specified when calling
      the method.</ToolTip>
  </Literal>
</Declarations>
<Code Language="CSharp" Delimiter="~">
  <![CDATA[partial class Program
{
  static void ConfigureConsole(string culture = "~defaultCulture~",
    bool useComputerCulture = false)
  {
    // Для включения в консоли символов Юникода, например символа валюты евро
    OutputEncoding = System.Text.Encoding.UTF8;

    if (!useComputerCulture)
    {
      CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo(culture);
    }
    WriteLine($"CurrentCulture: {CultureInfo.CurrentCulture.DisplayName}");
  }
}]]>
</Code>
</Snippet>
</CodeSnippet>
</CodeSnippets>

```

18. Сохраните файл.
19. Выберите команду меню Tools ▶ Code Snippets Manager (Средства ▶ Диспетчер фрагментов кода).
20. Нажмите кнопку Import (Импорт), перейдите в папку \tools-skills-net8\Chapter02\CodeSnippetDemos\, где вы сохранили файл ConfigureConsole.snippet, выберите его и нажмите кнопку Open (Открыть).
21. В диалоговом окне Import Code Snippet (Импорт фрагмента кода) выберите папку My Code Snippets (Мои фрагменты кода) и нажмите кнопку Finish (Завершить).
22. В окне Code Snippets Manager (Диспетчер фрагментов кода):
  - в разделе Language (Язык) выберите значение CSharp;
  - в разделе Location (Расположение) разверните папку My Code Snippets (Мои фрагменты кода);
  - выберите пункт Configure console for culture (Настроить консоль на культуру);
  - проверьте отображаемую информацию о фрагменте, включая ярлык: ccc.
23. Нажмите кнопку OK.
24. В проекте CodeSnippetDemos создайте файл класса и присвойте ему имя Program.Functions.cs.
25. В этом файле удалите существующий код, введите ccc, дважды нажмите клавишу Tab и обратите внимание на вставленный фрагмент кода с выделением текста en-US, готового к замене на нужное значение.

26. Сохраните файл.
27. В верхней части файла `Program.cs` вызовите соответствующий метод:  
`ConfigureConsole();`
28. Запустите проект и посмотрите на результат выполнения, как показано в следующем выводе:

```
CurrentCulture: English (United States)
C:\tools-skills-net8\Chapter02\CodeSnippetDemos\bin\Debug\net8.0
```

## Распространение фрагментов кода

Фрагменты кода можно распространять в виде отдельных файлов с расширением `.snippet`. Разработчики могут вручную импортировать их с помощью окна **Code Snippets Manager** (Диспетчер фрагментов кода).

Если нужен более формальный и масштабируемый подход, такие файлы можно включить в расширение для Visual Studio. Благодаря этому разработчики смогут устанавливать фрагменты вместе с расширением и использовать их без дополнительных операторов импорта.



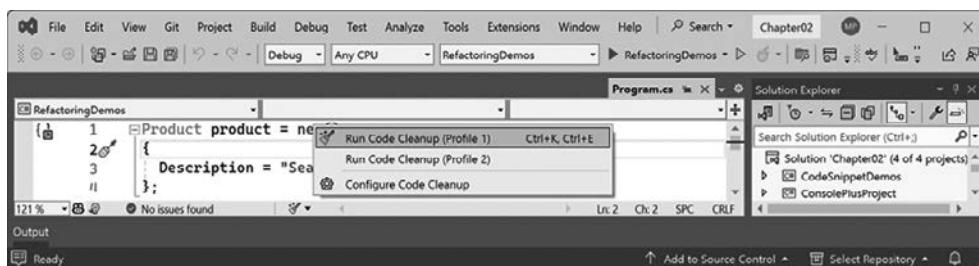
**В копилку знаний:** полную официальную документацию по функционалу фрагментов кода в Visual Studio вы найдете по следующим ссылкам: [learn.microsoft.com/ru-ru/visualstudio/ide/code-snippets](https://learn.microsoft.com/ru-ru/visualstudio/ide/code-snippets) и [learn.microsoft.com/ru-ru/visualstudio/ide/visual-csharp-code-snippets](https://learn.microsoft.com/ru-ru/visualstudio/ide/visual-csharp-code-snippets).

## Конфигурация редактора

Одно из главных преимуществ файла `.editorconfig` в проекте — он помогает поддерживать единый стиль программирования для всех участников команды. Настройки из этого файла имеют приоритет над глобальными параметрами редактора Visual Studio и применяются автоматически. Если в проекте отсутствует `.editorconfig`, вступают в силу настройки из диалогового окна **Options** (Параметры) Visual Studio. Но если файл присутствует, его параметры переопределяют глобальные.

Если вы добавляете `.editorconfig` в уже существующий проект, необходимо вручную применить его правила к текущему коду. Это можно сделать, выполнив команду **Code Cleanup** (Очистка кода) щелчком на значке в строке состояния или нажатием сочетания клавиш **Ctrl+K, E** (рис. 2.7).

Чтобы запустить функцию **Code Cleanup** (Очистка кода) для всего проекта или решения, на панели **Solution Explorer** (Обозреватель решений) щелкните правой кнопкой мыши по нужному проекту или решению и выберите команду меню **Analyze and Code Cleanup ▶ Run Code Cleanup** (Анализ и очистка кода ▶ Запустить очистку кода).



**Рис. 2.7.** Параметры очистки кода



Альтернативой встроенному в Visual Studio инструменту Code Cleanup является расширение CodeMaid. Хотя я сам не использую его, некоторые разработчики отдают предпочтение именно ему. Подробнее о нем можно узнать по адресу [marketplace.visualstudio.com/items?itemName=SteveCadwallader.CodeMaid](https://marketplace.visualstudio.com/items?itemName=SteveCadwallader.CodeMaid).

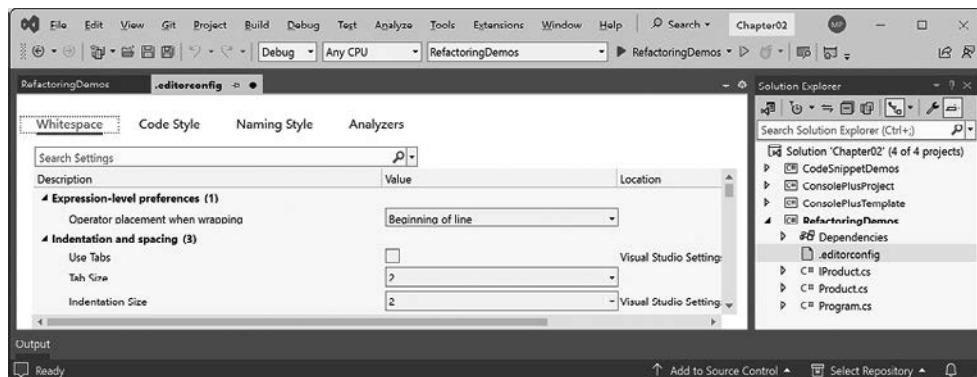
Теперь рассмотрим, как добавить файл `.editorconfig` в проект, чтобы задать стандарты оформления кода и обеспечить их соблюдение.

1. Откройте проект `RefactoringDemos` и выберите команду меню `Project > Add New Item` (Проект > Добавить новый элемент) или нажмите сочетание клавиш `Ctrl+Shift+A`.
2. В диалоговом окне `Add New Item` (Добавить новый элемент) в поле поиска введите `editor`, выберите пункт `editorconfig File (.NET)` (Файл `editorconfig (.NET)`), затем нажмите кнопку `Add` (Добавить).



`editorconfig File (.NET)` включает стиль, форматирование и соглашения об именовании, характерные для .NET. `editorconfig File (default)` отвечает только за параметры стиля и размер отступа.

3. В Visual Studio откроется графический редактор для редактирования файла `.editorconfig` с поддержкой поиска (рис. 2.8).
4. В поле `Search Settings` (Параметры поиска) введите слово `space`, и в результатах вы увидите 24 параметра, связанных с пробелами. Цель упражнения — помочь вам разобраться, как работает фильтрация при редактировании настроек. Вносить какие-либо изменения необязательно.
5. На панели `Solution Explorer` (Обозреватель решений) щелкните правой кнопкой мыши по файлу `.editorconfig` и выберите команду `Open With` (Открыть с помощью). В диалоговом окне выберите пункт `Common Language Editor Supporting TextMate Bundles` (Общий редактор языка с поддержкой пакетов TextMate), а затем нажмите кнопку `OK`. Этот редактор поддерживает функцию `IntelliSense` при редактировании файлов `.editorconfig`.



**Рис. 2.8.** Графический редактор Visual Studio с открытым файлом .editorconfig

6. Взгляните на набор свойств, фрагмент которых показан ниже:

```
[*.{cs,vb}]
##### Стили именования #####
# Правила именования

dotnet_naming_rule.interface_should_be_begins_with_i.severity = suggestion
dotnet_naming_rule.interface_should_be_begins_with_i.symbols = interface
dotnet_naming_rule.interface_should_be_begins_with_i.style = begins_with_i
...
dotnet_naming_style.pascal_case.capitalization = pascal_case
dotnet_style_operator_placement_when_wrapping = beginning_of_line
tab_width = 2
indent_size = 2
end_of_line = crlf
...
csharp_style_expression_bodied_constructors = false:silent
csharp_space_around_binary_operators = before_and_after
```

В этом разделе мы разобрались, как Visual Studio позволяет применять и изменять настройки в файлах .editorconfig. Сами свойства уже были рассмотрены ранее в этой главе.

Теперь перейдем к ИИ-компаньонам и посмотрим, как они могут помочь при работе с кодом.

## Интеллектуальные помощники на базе ИИ: GitHub Copilot

GitHub Copilot — это программный компаньон с поддержкой искусственного интеллекта, помогающий писать код. В Visual Studio версии 17.10 и новее он включен как рекомендуемый компонент. Если вы используете Visual Studio версии 17.9 или более раннюю, инструкции по установке Copilot можно найти на сайте [devblogs.microsoft.com/visualstudio/how-to-install-github-copilot-in-visual-studio/](https://devblogs.microsoft.com/visualstudio/how-to-install-github-copilot-in-visual-studio/).

Для работы с новым расширением GitHub Copilot требуется активная подписка. Сервис бесплатен для студентов с подтвержденным статусом, а также для разработчиков, поддерживающих популярные проекты с открытым исходным кодом на GitHub. Также доступна бесплатная пробная версия на 60 дней. После ее окончания подписка обойдется в 100 долларов в год. Подписаться можно на сайте [github.com/github-copilot/signup/](https://github.com/github-copilot/signup/).

GitHub Copilot Enterprise поддерживает расширенные сценарии, такие как обучение пользовательских моделей на основе содержимого ваших репозиториев.

## Эффективное использование GitHub Copilot

GitHub Copilot теперь тесно интегрирован в Visual Studio версии 17.10 и новее. Возможности, ранее разделенные между расширениями GitHub Copilot и GitHub Copilot Chat, теперь объединены и доступны по нажатию одной кнопки в правом верхнем углу интерфейса (рис. 2.9).



**Рис. 2.9.** GitHub Copilot интегрирован в Visual Studio

GitHub Copilot обеспечивает интерактивную работу с помощью чата в отдельном окне, а также прямо в редакторе кода. Вы можете начать диалог или просто набрать текст — в ответ появятся контекстно зависимые предложения, автозавершение и даже целые фрагменты кода. Представьте, что рядом с вами работает опытный напарник-программист — примерно так себя ведет Copilot: он помогает писать сообщения для коммитов, отлаживать код, генерировать модульные тесты и многое другое.

GitHub Copilot — это ценный помощник в работе с кодом: он предлагает пояснения, определения, ссылки на документацию, лучшие практики и даже объяснения ошибок. Перечислим некоторые задачи, с которыми он может помочь.

- При вводе кода в редакторе Copilot предлагает автодополнение. Чтобы принять предложение, нажмите клавишу **Tab**.
- Чтобы задать вопрос через встроенный чат, щелкните правой кнопкой мыши в файле и выберите в контекстном меню пункт **Ask Copilot** (Спросить у Copilot) либо нажмите сочетание клавиш **Alt+/. Copilot** может на основе вашего запроса сгенерировать метод, класс или модульный тест.
- Чтобы открыть боковую панель чата, нажмите кнопку **GitHub Copilot** в верхней части окна Visual Studio, затем выберите пункт **Open Chat Window** (Открыть окно чата) (рис. 2.10).

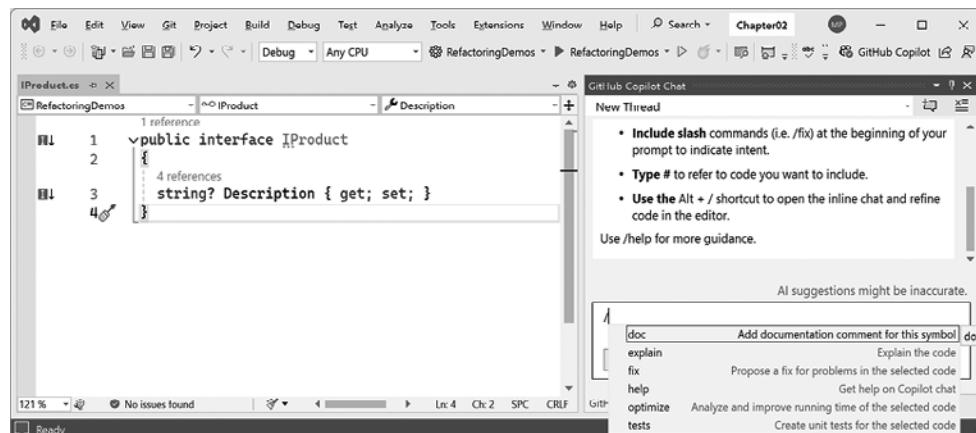


Рис. 2.10. Окно чата GitHub Copilot

- В окне чата введите символ /, чтобы указать намерение (например, /test или /doc), либо введите символ #, чтобы сослаться на файл или проект.
- Например, команда /test for # генерирует модульный тест для выбранного класса. Команда /doc # добавит комментарии к указанному файлу. Вы можете выбрать файл из выпадающего списка сразу после ввода символа # (рис. 2.11).

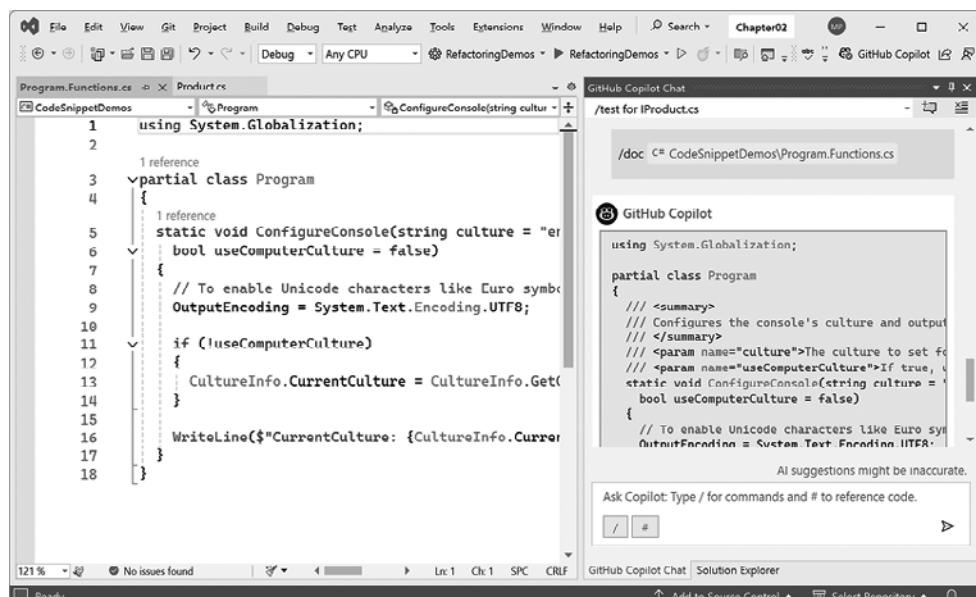


Рис. 2.11. Документирование класса с помощью GitHub Copilot

GitHub Copilot доступен не только в Visual Studio, но и в редакторах Visual Studio Code и Rider. Подробнее о Copilot в Rider можно прочитать на сайте [github.blog/changelog/2024-03-07-github-copilot-chat-general-availability-in-jetbrains-ide/](https://github.blog/changelog/2024-03-07-github-copilot-chat-general-availability-in-jetbrains-ide/).

## Навигация в Visual Studio

Может показаться, что это слишком просто, но каждый раз, когда я веду занятия, меня удивляет, как много разработчиков перемещаются по интерфейсу редактора исключительно с помощью мыши, хотя клавиатурные сочетания зачастую позволяют выполнять те же операции намного быстрее.

### Копирование и вставка выражения

Нет ничего более утомительного, чем наблюдать, как разработчик осторожно подводит указатель мыши к началу выражения, нажимает, удерживает и перетаскивает, пытаясь его выделить, — а потом случайно сбивается и начинает все заново.

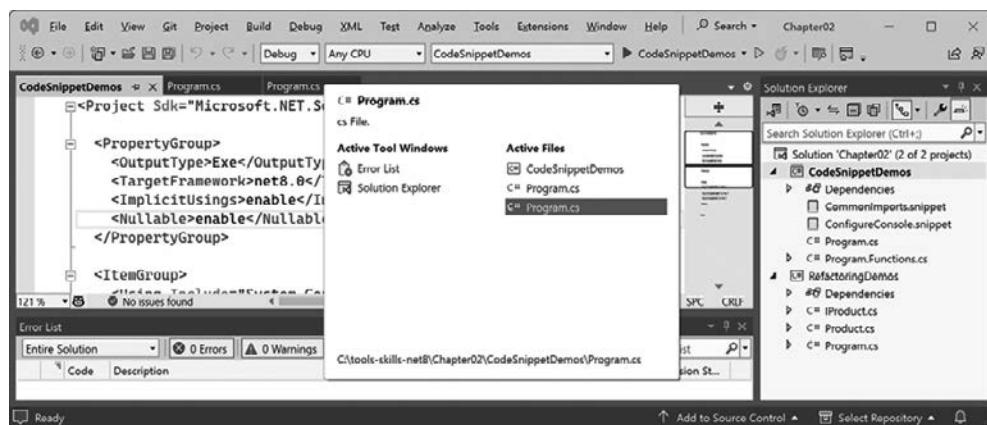
Чтобы скопировать или переместить выражение, не нужно его выделять вручную. Просто щелкните кнопкой мыши в любом месте выражения (не выделяя текст), затем нажмите сочетание клавиш **Ctrl+C**, чтобы скопировать, или **Ctrl+X**, чтобы вырезать. После этого щелкните в любом месте другого выражения и нажмите сочетание клавиш **Ctrl+V**, чтобы вставить скопированное *поверх* него.

### Переключение между вкладками файлов и окнами инструментов

Большинство пользователей знакомы с переключателем приложений Windows, который вызывается нажатием сочетания клавиш **Alt+Tab** (или **Alt+Shift+Tab** для переключения в обратном порядке). С его помощью можно быстро переключаться между открытыми приложениями.

В Visual Studio действует похожий механизм — *IDE Navigator*. Он работает аналогично и позволяет переключаться между активными окнами инструментов и файлами. Чтобы открыть его, нажмите сочетание клавиш **Alt+F7** или **Ctrl+Tab**. Для перехода в обратном направлении нажмите сочетание клавиш **Alt+Shift+F7** или **Ctrl+Shift+Tab**. Повторное нажатие клавиш переместит вас к следующему активному элементу.

В время удержания клавиш вы также можете использовать клавиши со стрелками для выбора нужного окна инструмента или вкладки файла. В нижней части окна отображается полный путь к активному файлу — это полезно, если у вас открыты файлы с одинаковыми именами (например, `Program.cs`), но из разных папок (рис. 2.12).



**Рис. 2.12.** IDE Navigator для переключения между активными окнами инструментов или активными файлами

Общие сочетания клавиш для навигации приведены в табл. 2.3.

**Таблица 2.3.** Общие сочетания клавиш для навигации

| Сочетание клавиш                | Описание                                                                    |
|---------------------------------|-----------------------------------------------------------------------------|
| Ctrl+-                          | Переключаться между открытыми файлами в порядке их последнего использования |
| Ctrl+Shift+-                    | Переключаться между открытыми файлами в обратном порядке                    |
| Alt+F7 или Ctrl+Tab             | Открыть IDE Navigator и перейти к следующему активному окну или файлу       |
| Alt+Shift+F7 или Ctrl+Shift+Tab | Перейти к предыдущему активному окну или файлу через IDE Navigator          |

## Визуальные настройки редактора кода

Редактирование кода — ключевая часть работы разработчика, и полезно знать функции редактора, которые делают этот процесс удобнее и точнее.

### Номера и перенос длинных строк

Номера строк могут отображаться слева в окне кода, но по умолчанию они не отображаются.

Чтобы включить отображение номеров строк в коде на C#, выполните следующие действия.

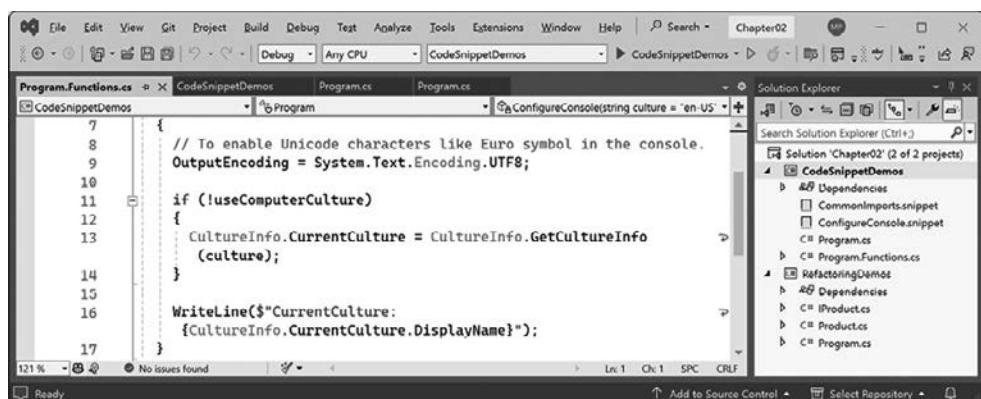
1. Выберите команду меню Tools ▶ Options (Средства ▶ Параметры) и перейдите на вкладку Text Editor ▶ C# (Текстовый редактор ▶ C#).

2. Установите флажок Line Numbers (Номера строк).
3. Нажмите кнопку OK.

Настройки переноса длинных строк включаются аналогичным образом.

1. Выберите команду меню Tools ▶ Options (Средства ▶ Параметры) и перейдите на вкладку Text Editor ▶ C# (Текстовый редактор ▶ C#).
2. Установите флажок Word wrap (Перенос строк).
3. При необходимости установите флажок Show visual glyphs for word wrap (Показывать визуальные глифы для переноса строк).
4. Нажмите кнопку OK.

Сочетание номеров строк и визуальных глифов особенно удобно — вы сразу видите, где строка кода была перенесена в редакторе (рис. 2.13).



**Рис. 2.13.** Визуальные глифы и номера строк при переносе длинных строк

## Сочетания клавиш

Ниже приведены часто используемые сочетания клавиш для редактирования кода в Visual Studio (табл. 2.4).

**Таблица 2.4.** Общие сочетания клавиш для редактирования

| Сочетание клавиш | Описание                                                                                  |
|------------------|-------------------------------------------------------------------------------------------|
| Alt+↑            | Переместить текущее выражение или выбранные строки на одну строку вверх                   |
| Alt+↓            | Переместить текущее выражение или выбранные строки на одну строку вниз                    |
| Ctrl+K, C        | Закомментировать текущие или выбранные строки. Тип комментария определяется по типу файла |
| Ctrl+K, U        | Убрать комментарии с текущих или выбранных строк                                          |

Попробуем эти сочетания клавиш в действии.

1. В файле `CodeSnippetDemos.csproj` проекта щелкните кнопкой мыши на одном из операторов импорта, например `<Using Include="System.IO.Path" Static="true" />`.
2. Нажмите `Alt+↑` — выражение переместится на одну строку вверх.
3. Еще раз нажмите `Alt+↑`, чтобы повторить перемещение.
4. Нажмите `Alt+↓` — выражение вернется на строку ниже.
5. Снова нажмите `Alt+↓`.
6. Нажмите `Ctrl+K, C` — к строке добавятся символы комментария XML.
7. Нажмите сочетание клавиш `Ctrl+K, U` — комментарии будут удалены.

## Форматирование кода

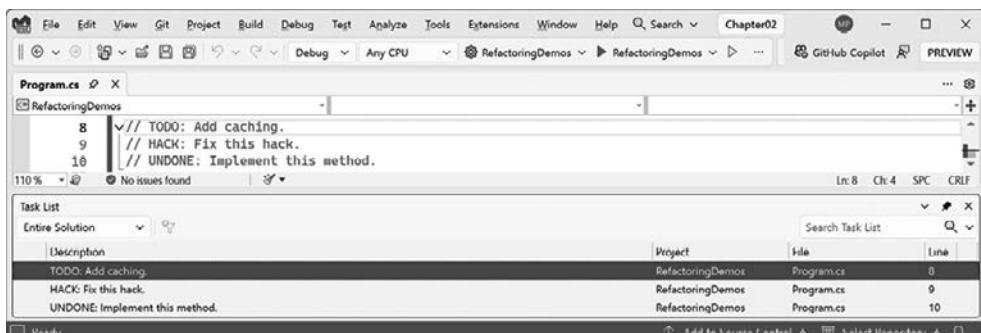
В Visual Studio можно настроить цвета для различных элементов синтаксиса. Например, ключевые слова C# отображаются синим цветом, а обычный код — черным. Начиная с версии 17.10 появилась дополнительная возможность — использовать **полужирное начертание** и *курсив*, что особенно удобно для выделения комментариев и других вспомогательных элементов в коде. Эту возможность разработчики просили добавить в течение десятилетий.

## Список задач

В Visual Studio доступен удобный список задач, в который можно добавлять элементы вручную или автоматически — на основе комментариев с маркерами, которые вы вставляете в код. По умолчанию распознаются три маркера, `TODO`, `HACK` и `UNDONE`:

```
// TODO: Реализовать кэширование
// HACK: Временное решение, нужно исправить
// UNDONE: Требуется реализация метода
```

Чтобы открыть панель **Task List** (Список задач), выберите команду меню **View ▶ Task List** (**Вид ▶ Список задач**) (рис. 2.14).



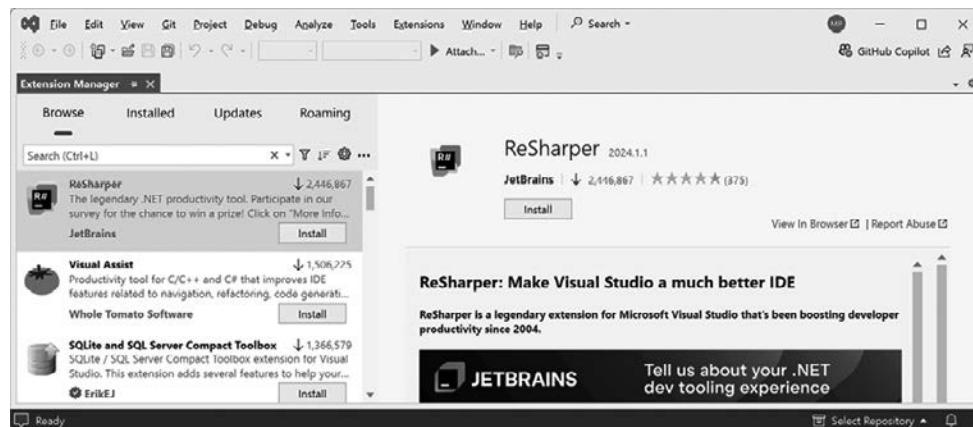
**Рис. 2.14.** Панель Task List в Visual Studio с маркерами TODO, HACK и UNDONE

Дважды щелкните по строке на панели Task List (Список задач), чтобы перейти к соответствующему комментарию в коде.

Чтобы добавить собственные маркеры, выберите команду меню Tools ▶ Options (Средства ▶ Параметры) и перейдите на вкладку Environment ▶ Task List (Окружение ▶ Список задач).

## Менеджер расширений

Начиная с версии 17.10 в Visual Studio используется обновленный менеджер расширений с улучшенным интерфейсом (рис. 2.15).



**Рис. 2.15.** Обновленный менеджер расширений в Visual Studio 17.10 и новее

Новый дизайн предоставляет больше пространства для контента, который важен при выборе расширений. В обновленном интерфейсе список результатов поиска включает расширенную панель с подробным описанием выбранного расширения.

Чтобы отфильтровать результаты по категориям, нажмите кнопку фильтра на панели инструментов. В раскрывающемся списке доступны параметры Category (Категория) и Subcategory (Подкатегория), которые помогают сузить выбор — например, отфильтровать только инструменты для программирования или шаблоны на C#.

## Инструменты в Visual Studio Code

Для начала полезно ознакомиться с основными сочетаниями клавиш в Visual Studio Code. На сайте [code.visualstudio.com/docs/getstarted/keybindings#\\_keyboard-shortcuts-reference](https://code.visualstudio.com/docs/getstarted/keybindings#_keyboard-shortcuts-reference) размещены удобные односторонние шпаргалки в формате PDF — по одной для каждой из популярных операционных систем: Windows, macOS и Linux. Их можно скачать и распечатать.

Если у вас открыто несколько вкладок, переключаться между ними можно с помощью сочетаний клавиш Alt+← / Alt+→ и Ctrl+Tab / Ctrl+Shift+Tab (в обратном порядке).

## Функции рефакторинга

В Visual Studio Code многие функции рефакторинга активируются с помощью сочетания клавиш Ctrl+Shift+R.

Ниже перечислены основные команды рефакторинга, доступные в Code.

- **Переименовать символ (F2)** — переименовывает все вхождения выбранного символа во всех файлах проекта.
- **Извлечь метод** — позволяет вынести выбранные выражения в отдельный общий метод.
- **Инкапсуляция поля** — преобразует поле в свойство и обновляет все обращения к нему.
- **Преобразование между автосвойством и полным свойством** — позволяет переключаться между автоматически реализованным и полноценным свойством.
- **Преобразование между инструкциями if и switch** — преобразует конструкцию if в оператор switch или выражение switch (начиная с C# 8).
- **Форматирование вызовов и параметров** — обертывание, отступы и выравнивание цепочек вызовов методов, параметров и аргументов.



**В копилку знаний:** подробнее о рефакторинге в Code вы можете прочитать по адресу [code.visualstudio.com/docs/csharp/refactoring](https://code.visualstudio.com/docs/csharp/refactoring).

## Фрагменты кода

Чтобы вставить фрагмент кода в Visual Studio Code, откройте палитру команд (Command Palette), затем выберите команду **Insert Snippet** (Вставить фрагмент). Также можно просто ввести ярлык фрагмента в редакторе и нажать клавишу Tab.

Создадим пользовательский фрагмент кода для Code.

1. В Visual Studio Code создайте проект консольного приложения с именем **SnippetDemos4Code** и добавьте его в решение **Chapter02**. На панели Terminal (Терминал) в каталоге **Chapter02** введите команды:

```
dotnet new console -o SnippetDemos4Code  
dotnet sln add SnippetDemos4Code
```

2. Выберите команду меню **View ▶ Command Palette** (Вид ▶ Палитра команд) или нажмите сочетание клавиш Ctrl+Shift+P.

3. Выберите Snippets: Configure User Snippets (Фрагменты: Настроить пользовательские фрагменты) и выберите язык csharp (C#).
4. В открывшемся файле `csharp.json` прочитайте комментарий, расположенный в начале файла.
5. Затем добавьте объект JSON, определяющий фрагмент для настройки консоли. Обратите внимание: для экранирования символов двойных кавычек нужно использовать обратную косую черту (\), как показано ниже:

```
{
  // Разместите здесь свои фрагменты для csharp.
  // Каждый фрагмент определяется под именем, имеет префикс, тело и описание.
  // Префикс – это то, что используется для вызова фрагмента, тело будет
  // развернуто и вставлено в код. Доступные переменные: $1, $2 –
  // для позиций табуляции, $0 – для позиции курсора после вставки,
  // ${1:метка}, ${2:другое} – для заполнителей.
  // Заполнители с одинаковыми идентификаторами связаны между собой.

  "Configure console for culture": {
    "prefix": "ccc",
    "body": [
      "using System.Globalization;",
      "using static System.Console;",
      ""
      "partial class Program",
      "{",
      "  static void ConfigureConsole(string culture = \"${1:en-US}\",",
      "    bool useComputerCulture = false)",
      "  {",
      "    // To enable Unicode characters like Euro symbol in the console.,
      "    OutputEncoding = System.Text.Encoding.UTF8;",
      "    ",
      "    if (!useComputerCulture)",
      "    {",
      "      CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo(culture);",
      "    }",
      "    ",
      "    WriteLine($"\"CurrentCulture: {CultureInfo.CurrentCulture.
        DisplayName}\");",
      "  }",
      "}"
    ],
    "description": "Define a static method for Program to configure the culture
      for the console.",
    "isFileTemplate": true
  }
}
```

6. Сохраните файл.
7. В проекте `SnippetDemos4Code` создайте файл класса `Program.Functions.cs`.
8. Удалите в нем все содержимое, введите `ccc`, нажмите клавишу `Tab` — фрагмент будет вставлен с выделенным значением `en-US`, которое можно изменить.

9. Сохраните файл.
10. В верхней части файла `Program.cs` вызовите метод:

```
ConfigureConsole();
```

11. Запустите проект и посмотрите на результат, как показано в выводе ниже:

```
CurrentCulture: English (United States)
```

## Декомпиляция сборок .NET

Один из лучших способов освоить программирование для .NET — посмотреть, как это делают профессионалы. Большинство редакторов кода поддерживают расширения для декомпиляции сборок .NET. В Visual Studio и Code можно воспользоваться расширением ILSpy, а в JetBrains Rider встроен инструмент IL Viewer.



**Дельный совет:** декомпиляция чужих сборок может использоваться не только для обучения — например, чтобы заимствовать фрагменты кода для собственной библиотеки или приложения. Но не забывайте: вы имеете дело с чьей-то интеллектуальной собственностью. Проявляйте уважение.

## Создание консольного приложения для декомпиляции

Создадим консольное приложение, которое затем декомпилируем.

1. В редакторе кода создайте проект `Console App` (Консольное приложение) / `console`, присвойте ему имя `DotNetEverywhere` и добавьте в решение `Chapter02`. Убедитесь, что выбрана целевая платформа — .NET 8.
2. В файле проекта добавьте статический глобальный импорт класса `System.Console`, чтобы он был доступен во всех файлах C#.
3. В файле `Program.cs` удалите содержимое и добавьте код, который выводит сообщение о кросс-платформенности приложения и сведения об используемой операционной системе:

```
WriteLine("I can run everywhere!");
WriteLine($"OS Version is {Environment.OSVersion}.");

if (OperatingSystem.IsMacOS())
{
    WriteLine("I am macOS.");
}
else if (OperatingSystem.WindowsVersionAtLeast(major: 10, build: 22000))
{
    WriteLine("I am Windows 11.");
}
```

```

else if (OperatingSystem.IsWindowsVersionAtLeast(major: 10))
{
    WriteLine("I am Windows 10.");
}
else
{
    WriteLine("I am some other mysterious OS.");
}
WriteLine("Press any key to stop me.");
ReadKey(intercept: true); // Нажатая клавиша не выводится на экран

```

4. Запустите проект `DotNetEverywhere` и посмотрите на результаты, которые появятся при запуске в Windows 11 (см. пример ниже):

```

I can run everywhere!
OS Version is Microsoft Windows NT 10.0.22000.0.
I am Windows 11.
Press any key to stop me.

```

5. В файле `DotNetEverywhere.csproj` добавьте идентификаторы среды выполнения (Runtime Identifiers, RID) для трех операционных систем внутри элемента `<PropertyGroup>`:

```

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

<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <RuntimeIdentifiers>
        win-x64;osx-x64;linux-x64
    </RuntimeIdentifiers>
</PropertyGroup>

</Project>

```



Существуют два элемента, с помощью которых задаются идентификаторы среды выполнения:

- `<RuntimeIdentifier>` — для одного значения;
- `<RuntimeIdentifiers>` — для нескольких значений.

Если использовать не тот элемент, компилятор сообщит об ошибке. Такая ситуация может сбить с толку, поскольку разница между элементами — всего один символ.

6. В терминале перейдите в папку `DotNetEverywhere`.
7. Введите команду для сборки и публикации автономной версии консольного приложения для Windows 10:

```
dotnet publish -c Release -r win-x64 --self-contained
```

8. Механизм сборки восстановит все необходимые пакеты, скомпилирует проект в DLL-сборку и создаст папку публикации, как показано в выводе ниже:

```
MSBuild version 17.8.0+14c24b2d3 for .NET
Determining projects to restore...
All projects are up-to-date for restore.
DotNetEverywhere -> C:\tools-skills-net8\Chapter02\DotNetEverywhere\bin\
Release\net8.0\win-x64\DotNetEverywhere.dll
DotNetEverywhere -> C:\tools-skills-net8\Chapter02\DotNetEverywhere\bin\
Release\net8.0\win-x64\publish\
```

9. Выполните команды для сборки и публикации релизов для macOS и Linux:

```
dotnet publish -c Release -r osx-x64 --self-contained
dotnet publish -c Release -r linux-x64 --self-contained
```

## Декомпиляция с помощью расширения ILSpy для Visual Studio

В учебных целях вы можете декомпилировать любую сборку .NET с помощью такого инструмента, как ILSpy.

1. В Visual Studio выберите команду меню **Extensions ▶ Manage Extensions** (Расширения ▶ Менеджер расширений).
2. В строке поиска введите **ilspy**.
3. Выберите расширение **ILSpy 2022** и нажмите кнопку **Download** (Загрузить).
4. Нажмите кнопку **Close** (Закрыть).
5. Закройте Visual Studio, чтобы завершить установку расширения.
6. Перезапустите Visual Studio и снова откройте решение **Chapter02**.
7. На панели **Solution Explorer** (Обозреватель решений) щелкните правой кнопкой мыши по проекту **DotNetEverywhere** и выберите в контекстном меню команду **Open output in ILSpy** (Открыть результат в ILSpy).
8. Убедитесь, что в раскрывающемся списке языков декомпиляции на панели ILSpy выбран вариант **C#**.
9. В дереве навигации **Assemblies** (Сборки) на панели ILSpy слева разверните узел **DotNetEverywhere (1.0.0.0, .NETCoreApp, v8.0)**.
10. Разверните строку **{ }**, затем — узел **Program** (Программа).
11. Выберите выражение **<Main>\$(string[]) : void**, чтобы просмотреть выражения в генерированном компилятором классе **Program** и его методе **<Main>\$** (рис. 2.16).
12. На панели ILSpy выберите команду меню **File ▶ Open** (Файл ▶ Открыть).
13. Перейдите в следующую папку:

```
cs12dotnet8/Chapter02/DotNetEverywhere/bin/Release/net8.0/linux-x64
```

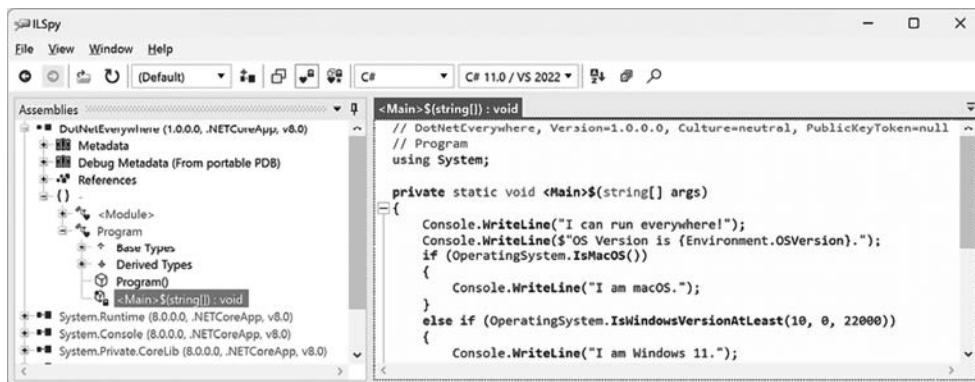


Рис. 2.16. Раскрытие метода `<Main>$` в программе ILSpy

14. Выберите сборку `System.Linq.dll` и нажмите кнопку `Open` (Открыть).
15. В дереве `Assemblies` (Сборки) разверните узел `System.Linq (8.0.0.0, .NETCoreApp, v8.0)`, затем пространство имен `System.Linq`, после чего — класс `Enumerable`, и щелкните по методу `Count<TSource>(this IEnumerable<TSource>) : int`.
16. В теле метода `Count` видно, как удачно реализована логика:
  - выполняется проверка параметра `source` и выбрасывается исключение `ArgumentNullException`, если он равен `null`;
  - осуществляется проверка на реализацию интерфейсов, содержащих собственное свойство `Count`, которое позволяет получить количество элементов более эффективно;
  - если такие интерфейсы не реализованы, элементы перечисляются вручную и счетчик увеличивается для каждого — это наименее эффективный вариант (рис. 2.17).



Разные инструменты декомпиляции могут генерировать немного отличающийся код — например, с разными именами переменных, — но при этом логика и функциональность останутся одинаковыми.

17. Откройте исходный код метода `Count` на языке C# в ILSpy, чтобы потом сравнить его с версией на промежуточном языке (Intermediate Language, IL):

```

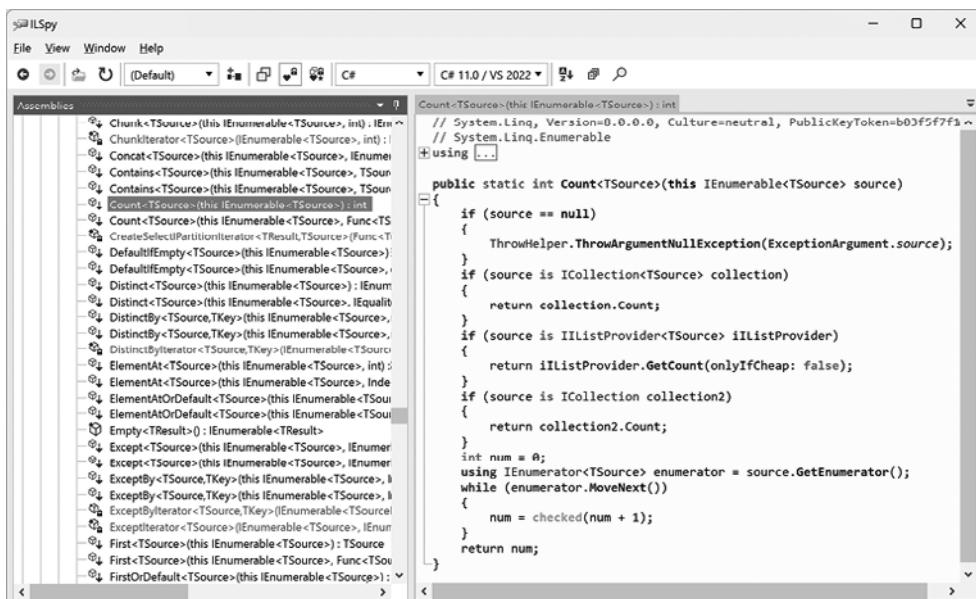
public static int Count<TSource>(this IEnumerable<TSource> source)
{
    if (source == null)
    {
        ThrowHelper.ThrowArgumentNullException(ExceptionArgument.source);
    }
    if (source is ICollection<TSource> collection)
    {
        return collection.Count;
    }
}

```

```

if (source is IIListProvider<TSource> iIListProvider)
{
    return iIListProvider.GetCount(onlyIfCheap: false);
}
if (source is ICollection collection2)
{
    return collection2.Count;
}
int num = 0;
using IEnumerator<TSource> enumerator = source.GetEnumerator();
while (enumerator.MoveNext())
{
    num = checked(num + 1);
}
return num;
}

```



**Рис. 2.17.** Декомпилированный метод Count класса Enumerable



**Дельный совет:** во многих постах в LinkedIn и статьях в технических блогах часто советуют всегда использовать свойство Count у коллекции, а не метод расширения Count() из LINQ. Как показано выше, такой совет необязателен: метод Count() сначала проверяет, реализует ли последовательность интерфейсы ICollection<T> или ICollection, и если да — использует свойство Count. Однако он не проверяет, является ли последовательность массивом, и не использует свойство Length. Поэтому, если вы работаете с массивами, предпочтительнее добавлять Length, а не Count().



Заключительная часть реализации метода Count демонстрирует, как на самом деле работает инструкция foreach. Она вызывает метод GetEnumerator, затем в цикле while — метод MoveNext. Для подсчета используется переменная типа int, значение которой увеличивается при каждой итерации. Все это выполняется внутри конструкции checked, поэтому при переполнении выбрасывается исключение. Именно поэтому метод Count может корректно подсчитывать только примерно до 2 миллиардов элементов.

18. На панели IL Spy в раскрывающемся списке Select language to decompile (Выбрать язык для декомпиляции) выберите IL, а затем изучите исходный код метода Count на промежуточном языке. Чтобы сэкономить место в книге, этот код здесь не приводится.



**Дельный совет:** IL-код может быть полезен, только если вы хорошо знаете C# и глубоко погружены в разработку под .NET. В этом случае важно понимать, как компилятор C# транслирует ваш код в IL. Но гораздо более полезны окна с декомпилированным C#-кодом, написанным профессионалами из Microsoft. Из таких примеров можно узнать много хороших практик — например, метод Count показывает, как надежно проверять аргументы на null.

19. Закройте IL Spy.



О том, как использовать расширение IL Spy в программе Code, можно узнать по адресу [github.com/markjprice/cs12dotnet8/blob/main/docs/code-editors/vscode.md#decompiling-using-the-ilspy-extension-for-visual-studio-code](https://github.com/markjprice/cs12dotnet8/blob/main/docs/code-editors/vscode.md#decompiling-using-the-ilspy-extension-for-visual-studio-code).

## Просмотр ссылок на исходный код в Visual Studio

Вместо декомпиляции в Visual Studio можно воспользоваться функцией просмотра оригинального исходного кода с помощью функции анализа ссылок на источник (Source Link). Эта функция недоступна в редакторе Code.

Разберемся, как она работает.

1. В Visual Studio включите поддержку Source Link:
  - выберите команду меню Tools ▶ Options (Средства ▶ Параметры);
  - в строке поиска введите navigation to source;
  - перейдите в раздел Text Editor ▶ C# ▶ Advanced (Текстовый редактор ▶ C# ▶ Дополнительно);
  - установите флажок Enable navigation to Source Link and Embedded sources (Включить навигацию по исходным ссылкам и встроенным источникам), затем нажмите кнопку OK.

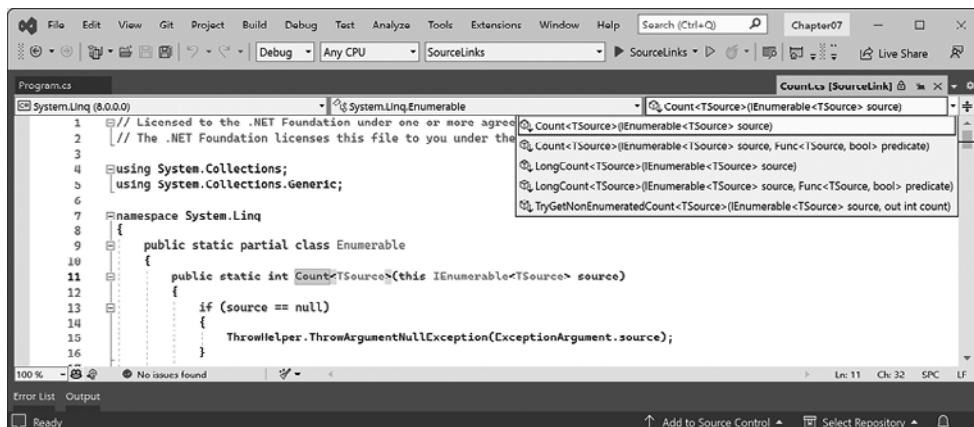
2. В Visual Studio создайте проект Console App (Консольное приложение) с именем SourceLinks и добавьте его в решение Chapter02.

3. В файле `Program.cs` удалите содержимое. Добавьте код, в котором создается переменная типа `string`, затем выводятся ее значение и количество символов в ней:

```
string name = "Timothée Chalamet";
int length = name.Count();
Console.WriteLine($"{name} has {length} characters.");
```

4. Щелкните правой кнопкой мыши по методу `Count()` и в контекстном меню выберите команду `Go To Implementation` (Перейти к реализации).

5. Откроется файл исходного кода с именем `Count.cs`, в котором определен частичный класс `Enumerable`, содержащий пять реализаций методов для подсчета элементов (рис. 2.18).



**Рис. 2.18.** Просмотр исходного файла с реализацией метода Count из LINQ

Ссылки на исходный код часто дают гораздо больше информации, чем декомпиляция. Они показывают, как устроен проект на самом деле, включая лучшие практики — например, как класс разбит на части для удобства сопровождения. В отличие от этого, при декомпиляции с помощью ILSpy вы просто увидите огромный список методов в классе `Enumerable`, без какой-либо структуры или контекста.



Подробнее о том, как работает Source Link и как его может поддерживать любой пакет NuGet, читайте на сайте [learn.microsoft.com/ru-ru/dotnet/standard/library-guidance/sourcelink](https://learn.microsoft.com/ru-ru/dotnet/standard/library-guidance/sourcelink).

6. Если вам все же удобнее работать с декомпиляцией, вы можете отключить функцию Source Link.

## Нет, предотвратить декомпиляцию техническими средствами невозможно

Меня иногда спрашивают, можно ли как-то защитить скомпилированный код от декомпиляции. Если вкратце — нет. И если подумать, становится понятно, почему это так. Задачу можно усложнить, воспользовавшись инструментами обfuscации, например *Dotfuscator*, но полностью исключить возможность декомпиляции не получится.

Любое скомпилированное приложение содержит инструкции, предназначенные для платформы, операционной системы и аппаратного обеспечения, на котором оно запускается. Эти инструкции должны выполнять те же действия, что и исходный код, только в менее читаемом виде. И поскольку они должны быть понятны для выполнения, они так же понятны и для декомпиляции. Если бы код нельзя было декомпилировать, его нельзя было бы и запустить.

Виртуальные машины эмулируют работу аппаратного обеспечения и могут фиксировать все взаимодействия между запущенным приложением и системой, на которой оно работает — или, по крайней мере, считает, что работает.

Если бы существовал способ полностью защитить код, то он оказался бы недоступен и для отладки. Но если у скомпилированного приложения есть PDB-файл, вы можете подключить отладчик и просмотреть строки кода. Но и без PDB-файла все равно можно подключиться и получить общее представление о том, как работает программа.

Это справедливо для всех языков программирования — не только для платформы .NET (C#, Visual Basic, F#), но и для C, C++, Delphi и даже ассемблера. Ко всем этим языкам можно подключиться для отладки, дизассемблирования или декомпиляции.

Некоторые инструменты, которые применяются для таких задач, приведены в табл. 2.5.

**Таблица 2.5.** Профессиональные отладчики, декомпиляторы и дизассемблеры

| Тип                | Продукт | Описание                                                                                                              |
|--------------------|---------|-----------------------------------------------------------------------------------------------------------------------|
| Виртуальная машина | VMware  | Используется профессионалами, например аналитиками вредоносных программ, для запуска приложений в изолированной среде |
| Отладчик           | SoftICE | Работает внутри операционной системы, обычно запускается в виртуальной машине                                         |
| Отладчик           | WinDbg  | Знает больше о внутренних структурах данных Windows, чем большинство других отладчиков                                |

*Продолжение ↗*

**Таблица 2.5 (окончание)**

| Тип          | Продукт  | Описание                                                      |
|--------------|----------|---------------------------------------------------------------|
| Дизассемблер | IDA Pro  | Популярный инструмент среди аналитиков вредоносного ПО        |
| Декомпилятор | Hex-Rays | Плагин для IDA Pro, декомпилирует приложения, написанные на С |
| Декомпилятор | DeDel    | Предназначен для декомпиляции приложений на языке Delphi      |
| Декомпилятор | dotPeek  | Декомпилятор платформы .NET компании JetBrains                |



**Дельный совет:** отладка, дизассемблирование и декомпиляция чужого ПО, как правило, нарушают лицензионные соглашения и во многих странах считаются незаконными. Вместо попыток защитить интеллектуальную собственность с помощью технических средств иногда единственным реальным способом защиты остается закон.

## Снижение уровня C#-конструкций

Одна из полезных возможностей плагина *IL Viewer* для Rider — это умение преобразовывать высокоуровневый код на C# в более низкоуровневый, но все еще читаемый C#, а не сразу в IL-код. Если у вас нет Rider, позже вы узнаете, как можно воспользоваться онлайн-инструментом [Sharplab.io](#).

Но что вообще такое «снижение уровня конструкций»?

Так называют технику, при которой компилятор C# переписывает удобные конструкции языка — так называемый «синтаксический сахар» — в более примитивные элементы языка, находящиеся на более низком уровне абстракции.

Напомню, что компиляция — это процесс преобразования исходного кода на C# в промежуточный IL-код, который затем во время выполнения превращается в машинный код конкретного процессора (если, конечно, вы не используете компилятор AOT!).

А снижение уровня конструкций — это как бы дополнительный этап: перевод удобных высокоуровневых возможностей языка в более низкоуровневые конструкции, но все еще в рамках самого языка C#.

Вероятно, самый известный пример снижения уровня конструкций — это переписывание цикла `foreach`:

```
// Цикл foreach – это высокоуровневая конструкция C#
foreach(Person person in people)
{
    Console.WriteLine(person);
}
```

```
// Компилятор снижает ее до кода, основанного на интерфейсе IEnumerable
// и вызовах его членов
List<Person>.Enumerator enumerator = people.GetEnumerator();
try
{
    while (enumerator.MoveNext())
        Console.WriteLine((object) enumerator.Current);
}
finally
{
    enumerator.Dispose();
}
```

Существуют и другие распространенные примеры конструкций языка высокого уровня, которые компилятор автоматически преобразует в более низкоуровневый код.

- Ключевое слово `record` преобразуется в обычный `class`, реализующий интерфейс `IEquatable`
- Конструкции `async` и `await` превращаются в конечный автомат — благодаря этому асинхронный код выполняется без блокировки основного потока
- Лямбда-выражения и анонимные методы преобразуются либо в делегаты, либо в деревья выражений
- Инструкции `using` и `lock` разворачиваются в блоки `try...finally`, чтобы гарантировать корректную очистку ресурсов.

Рассмотрим несколько примеров снижения уровня конструкций, воспользовавшись программой Rider и сайтом SharpLab.io.

1. Запустите Rider. (Если Rider у вас не установлен, просто следуйте шагам и посмотрите снимки экрана — особенно шаг 7.)
2. В Rider откройте решение `Chapter02`, затем создайте проект `Console App` (Консольное приложение) с именем `LoweringCode`.
3. В файле `Program.cs` удалите содержимое и добавьте код, в котором определяется список из двух человек и используется цикл `foreach` для вывода их в консоль:

```
List<Person> people = [ new("Bob", 47), new("Alice", 23) ];

foreach(Person person in people)
{
    Console.WriteLine(person);
}

record Person(string FirstName, int Age);
```



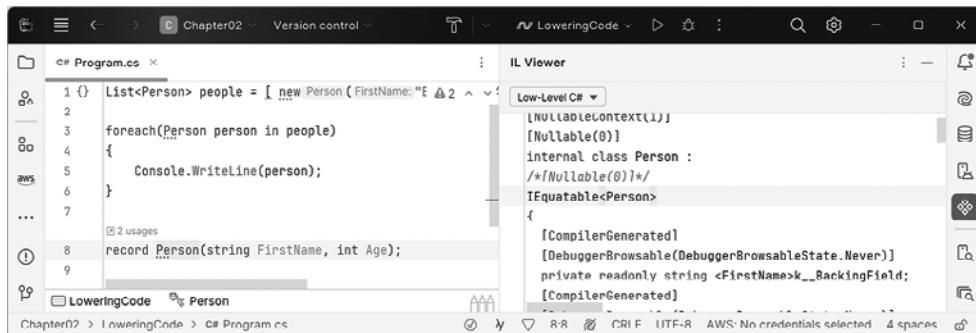
Если вы определяете тип в файле, который используется как программа верхнего уровня, этот тип должен располагаться в нижней части файла.

4. В Rider создайте проект LoweringCode.
5. В инструменте IL Viewer выберите режим Low-Level C# (C# низкого уровня), щелкните на закрывающей скобке `foreach` и обратите внимание: IL Viewer автоматически покажет эквивалентный код C# со сниженным уровнем конструкций (рис. 2.19).



**Рис. 2.19.** В инструменте IL Viewer показан C#-код со снижением уровня конструкций для цикла `foreach`

6. В файле `Program.cs` щелкните кнопкой мыши на записи `record`, затем немного прокрутите окно IL Viewer вверх и посмотрите, как выглядит эквивалентный код: это обычный класс, реализующий интерфейс `IEquatable<Person>` (рис. 2.20).



**Рис. 2.20.** Инструмент IL Viewer показывает C#-код со снижением уровня конструкций для типа `record`

7. Откройте предпочтаемый браузер и перейдите на сайт [sharplab.io](http://sharplab.io).
8. В окне SharpLab.io удалите все содержимое и вставьте код, который определяет список из двух человек и выводит их в консоль с помощью цикла `foreach`.

```
using System.Collections.Generic;

List<Person> people = [ new("Bob", 47), new("Alice", 23) ];
```

```

foreach(Person person in people)
{
    System.Console.WriteLine(person);
}

record Person(string FirstName, int Age);

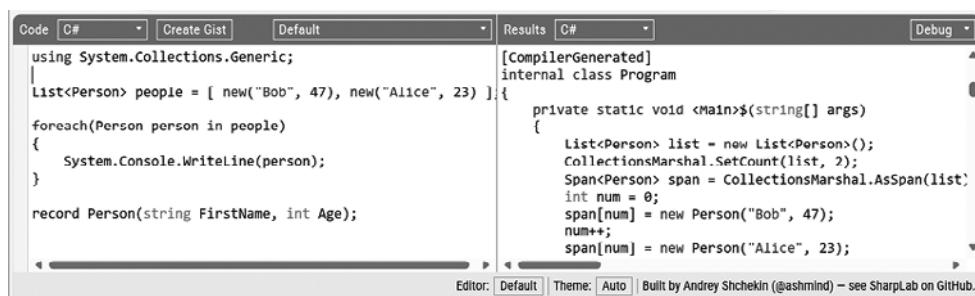
```

9. Обратите внимание на сгенерированный компилятором код для класса `Program` и метода `<Main>$`, который отображается в правой части окна (см. рис. 2.21 и фрагмент кода ниже).

```

[CompilerGenerated]
internal class Program
{
    private static void <Main>$(string[] args)
    {

```



**Рис. 2.21.** Сайт SharpLab.io показывает результат декомпиляции и снижения уровня конструкций на платформе .NET

10. Обратите внимание, как выражение инициализации коллекции преобразуется в использование класса `Span`, который позволяет эффективно заполнить список экземпляров `Person`.

```

List<Person> list = new List<Person>();
CollectionsMarshal.SetCount(list, 2);
Span<Person> span = CollectionsMarshal.AsSpan(list);
int num = 0;
span[num] = new Person("Bob", 47);
num++;
span[num] = new Person("Alice", 23);

```

11. Посмотрите, как инструкция `foreach` преобразована в прямые вызовы перечислителя (`GetEnumerator`, `MoveNext`, `Current`).

```

List<Person>.Enumerator enumerator = list2.GetEnumerator();
try
{
    while (enumerator.MoveNext())
    {
        Person current = enumerator.Current;

```

```
        Console.WriteLine(current);
    }
}
finally
{
    ((IDisposable)enumerator).Dispose();
}
```

12. Обратите внимание, как запись `record` транслируется в обычный класс с реализацией интерфейса `IEquatable<Person>`, а также с резервными полями и свойствами.

```
internal class Person : IEquatable<Person>
{
    [CompilerGenerated]
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    private readonly string <FirstName>k__BackingField;
    ...
    public string FirstName
    {
        [CompilerGenerated]
        get
        {
            return <FirstName>k__BackingField;
        }
        [CompilerGenerated]
        init
        {
            <FirstName>k__BackingField = value;
        }
    }
}
```

13. Посмотрите, как сгенерирован метод `ToString`, — он включает все значения свойств, обеспечивая читаемый вывод.

```
[CompilerGenerated]
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("Person");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(' ');
    }
    stringBuilder.Append('}');
    return stringBuilder.ToString();
}

[CompilerGenerated]
protected virtual bool PrintMembers(StringBuilder builder)
{
    RuntimeHelpers.EnsureSufficientExecutionStack();
    builder.Append("FirstName = ");
    builder.Append((object)FirstName);
```

```

builder.Append(", Age = ");
builder.Append(Age.ToString());
return true;
}

```



В эквиваленте записи со снижением уровня конструкций заметно больше кода — особенно в части реализации сравнения на равенство. Стоит уделить пару минут, чтобы внимательно все это изучить.

14. Запустите консольный проект — либо в Rider, либо на сайте SharpLab.io. В SharpLab.io выберите в раскрывающемся списке Result (Результат) пункт Run (Запуск) и посмотрите на результат выполнения:

```

Person { FirstName = Bob, Age = 47 }
Person { FirstName = Alice, Age = 23 }

```

Теперь, когда мы заглянули под капот C# и разобрались, как он работает на более низком уровне, чтобы писать более понятный и производительный код, самое время перейти к созданию собственных шаблонов проектов и элементов проекта.

## Пользовательские шаблоны проектов и элементов

В состав .NET SDK входит множество встроенных шаблонов проектов и элементов. Они позволяют создавать новые решения и файлы с помощью команды `dotnet new`, например: `console`, `classlib`, `web`, `mvc`, `blazor` и др.



**В копилку знаний:** полный список встроенных шаблонов проектов и элементов доступен по адресу [learn.microsoft.com/ru-ru/dotnet/core/tools/dotnet-new-sdk-templates](https://learn.microsoft.com/ru-ru/dotnet/core/tools/dotnet-new-sdk-templates).

Один из самых удобных способов настроить работу с редактором под себя — создать собственный шаблон проекта .NET. Такие шаблоны поддерживаются всеми редакторами кода. Вы можете создавать шаблоны как целых проектов, так и отдельных файлов. Они распространяются в виде пакетов NuGet (`.nupkg`) и устанавливаются с помощью команды `dotnet new install`.

Шаблоны проектов и элементов устроены одинаково и включают следующие компоненты.

- Все необходимые папки и файлы, в том числе исходный код, встроенные ресурсы, примеры данных и т. д.
- Файл `template.json`, в котором описано, как шаблон должен отображаться в пользовательском интерфейсе и как на его основе создавать проект или файл.

Превратить обычный .NET-проект в шаблон можно за пару шагов — достаточно добавить в него папку `.template.config` и положить в нее файл `template.json`.

Файл `template.json` обычно содержит следующую информацию (табл. 2.6).

**Таблица 2.6.** Основные элементы файла `template.json`

| Член                             | Описание                                                                                                                                                                        | Обязательный |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------|
| <code>\$schema</code>            | Ссылка на JSON-схему, расположенную по адресу <a href="http://json.schemastore.org/template">http://json.schemastore.org/template</a>                                           | Да           |
| <code>author</code>              | Имя автора шаблона                                                                                                                                                              | Нет          |
| <code>classifications</code>     | Список тегов, помогающих находить шаблон. Например: Common, Web, MAUI, Cloud, Test, ASP.NET, Blazor, Mobile, Console, .NET Aspire, Web API, Service                             | Нет          |
| <code>identity</code>            | Уникальный идентификатор шаблона                                                                                                                                                | Да           |
| <code>name</code>                | Полное название шаблона, отображаемое в интерфейсе. Например: <code>Console App</code> (Консольное приложение)                                                                  | Да           |
| <code>shortName</code>           | Краткое имя для использования с помощью команды <code>dotnet new</code> . Например: <code>console</code>                                                                        | Да           |
| <code>sourceName</code>          | Имя в исходном дереве файлов, которое заменяется на имя, указанное пользователем с помощью параметра <code>-n</code> или <code>--name</code> . По умолчанию — имя текущей папки | Нет          |
| <code>preferNameDirectory</code> | Логическое значение: указывает, создавать ли папку с именем проекта, если оно не задано явно                                                                                    | Нет          |



**В копилку знаний:** полную схему файла `template.json` можно найти по адресу [json.schemastore.org/template](http://json.schemastore.org/template). Документацию по формату файла — на сайте [github.com/dotnettemplating/wiki](https://github.com/dotnettemplating/wiki).

Если пользовательские шаблоны размещены в соответствующих папках, редактор кода или инструмент командной строки `dotnet` смогут их обнаружить и использовать.

В Visual Studio шаблоны автоматически отображаются в следующих локациях.

- Шаблоны проектов — в диалоговом окне `Create a new project` (Создать новый проект).
- Шаблоны элементов — в диалоговом окне `Add New Item` (Добавить новый элемент).

## Создание проекта для шаблона

Чтобы создать пользовательский шаблон проекта, нужно сначала добавить обычный проект, который и станет основой шаблона. Существует даже отдельный шаблон проекта, предназначенный специально для создания шаблонов проектов.

В этом разделе мы создадим шаблон для консольного приложения, включающий дополнительные файлы для настройки культуры, используемой приложением; для импорта часто применяемых типов.

1. В терминале введите команду для установки набора шаблонов, которые помогают создавать и настраивать собственные шаблоны проектов:

```
dotnet new install Microsoft.TemplateEngine.Authoring.Templates
```

2. Обратите внимание на результат выполнения команды — он будет примерно таким, как показано в следующем выводе:

```
The following template packages will be installed:  
Microsoft.TemplateEngine.Authoring.Templates  
  
Success: Microsoft.TemplateEngine.Authoring.Templates::8.0.101 installed the  
following templates:  
Template Name           Short Name     Language Tags  
-----  
Template Package         templatepack      Template Authoring  
template.json configuration file template.json JSON      Template Authoring
```

3. В Visual Studio создайте проект **Template Package** (Пакет шаблонов) / **templatepack**, присвойте ему имя **ConsolePlusTemplate** и добавьте в решение **Chapter02**. Подсказка: чтобы быстрее найти нужный шаблон, введите в поле поиска фразу **Template Authoring** — это отфильтрует список и покажет только шаблоны для создания собственных шаблонов.
4. В файле **ConsolePlusTemplate.csproj** проекта внесите необходимые изменения в конфигурацию:

```
<Project Sdk="Microsoft.NET.Sdk">  
  
  <PropertyGroup>  
    <!-- Метаданные пакета. Заполните указанные ниже свойства,  
        отмеченные как TODO. -->  
    <!-- Следуйте рекомендациям на https://learn.microsoft.com/  
        ru-ru/nuget/create-packages/package-authoring-best-practices -->  
    <PackageId>Packt.ConsolePlus.Template</PackageId>  
    <PackageVersion>1.0</PackageVersion>  
    <Title>Console App (Packt)</Title>  
    <Authors>Mark J. Price;Packt Publishing</Authors>  
    <Description>A console app with extra imported types and a method to  
        configure the culture.</Description>
```

```

<PackageTags>dotnet-new;templates;packt;console</PackageTags>
<PackageProjectUrl>https://www.github.com/markjprice/tools-skillsnet8
</PackageProjectUrl>

<!-- Укажите тип пакета как 'Template', чтобы он отображался
    как шаблон на nuget.org и был доступен в новом механизме поиска
    команды dotnet new -->
<PackageType>Template</PackageType>
<TargetFramework>net8.0</TargetFramework>
<IncludeContentInPack>true</IncludeContentInPack>
<IncludeBuildOutput>false</IncludeBuildOutput>
<ContentTargetFolders>content</ContentTargetFolders>
<NoWarn>$(NoWarn);NU5128</NoWarn>
<NoDefaultExcludes>true</NoDefaultExcludes>
<PackageReadmeFile>README.md</PackageReadmeFile>
</PropertyGroup>

<PropertyGroup>
    <LocalizeTemplates>false</LocalizeTemplates>
</PropertyGroup>

<ItemGroup>
    <PackageReference Include="Microsoft.TemplateEngine.Tasks" Version="*"
        PrivateAssets="all" IsImplicitlyDefined="true"/>
</ItemGroup>

<ItemGroup>
    <Content Include="content\**\*" Exclude="content\**\bin\**;content\
        **\obj\**" />
    <Compile Remove="**\**" />
</ItemGroup>

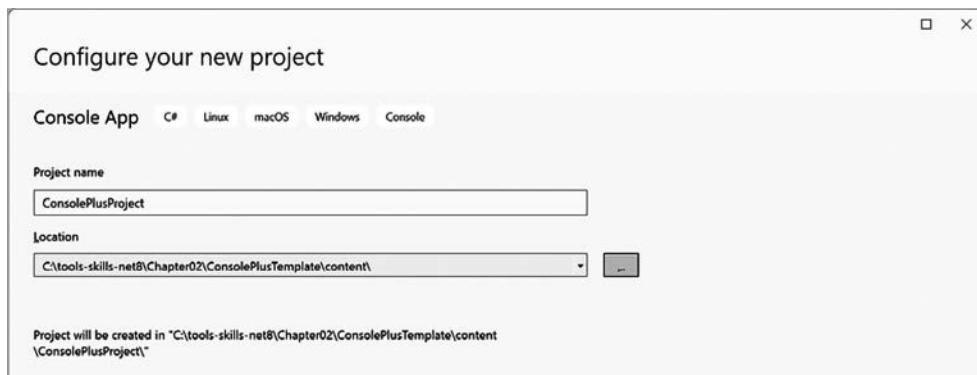
<ItemGroup>
    <None Include="README.md" Pack="true" PackagePath="" />
</ItemGroup>

</Project>

```

Обратите внимание на следующие параметры конфигурации:

- параметр **<PackageType>** должен иметь значение **Template**, чтобы пакет отображался как шаблон в результатах поиска команды **dotnet new**;
  - параметр **<IncludeContentInPack>** должен быть установлен в **true**, чтобы в пакет включались все папки и файлы, указанные в группе элементов **<Content>**.
5. В папке **content** удалите папку **SampleTemplate**.
  6. В Visual Studio создайте проект **Console App** (Консольное приложение) с именем **ConsolePlusProject** и укажите папку **content** в качестве расположения проекта. Убедитесь, что вы вручную изменили значение параметра **Location** (Расположение) на **content** (рис. 2.22).



**Рис. 2.22.** Настройка расположения проекта консольного приложения

7. В файле `ConsolePlusProject.csproj` проекта добавьте импорт часто используемых типов:

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



Если вы уже завершили раздел, в котором создавался фрагмент XML-кода, можете воспользоваться им и просто добавить новый элемент `<ItemGroup>`.

8. В проекте `ConsolePlusProject` создайте файл класса `Program.Functions.cs`.
9. В созданном файле удалите содержимое и добавьте код, определяющий частичный класс `Program` с методом `ConfigureConsole`:



Если вы завершили раздел с фрагментами C#-кода, можете вставить этот код с помощью сочетания клавиш **ccc**.

```
using System.Globalization;

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

```

if (!useComputerCulture)
{
    CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo(culture);
}

WriteLine($"CurrentCulture: {CultureInfo.CurrentCulture.DisplayName}");
}
}

```

10. В файле `Program.cs` удалите содержимое и добавьте код, вызывающий метод `ConfigureConsole`, как показано ниже:

```

ConfigureConsole(); // По умолчанию используется культура en-US

// Альтернативы:
// ConfigureConsole("fr-FR"); // Французский язык во Франции
// ConfigureConsole(useComputerCulture: true); // Локальная культура компьютера

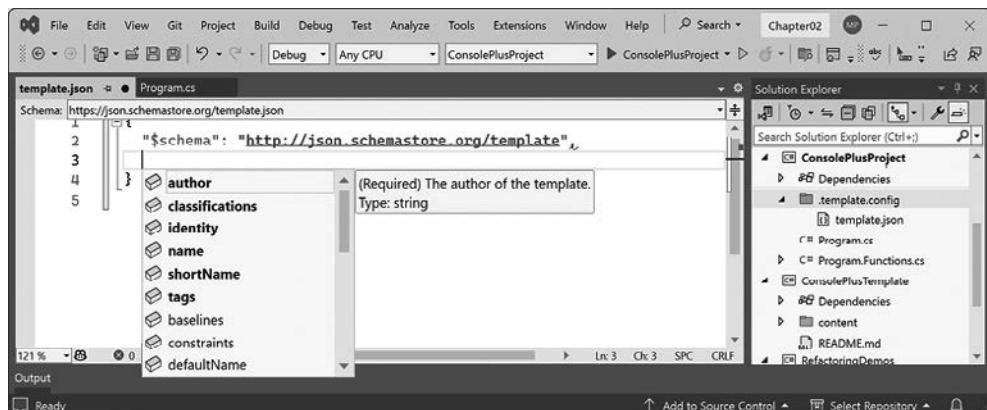
// Добавьте свой код ниже:

```

11. В папке проекта `ConsolePlusProject` создайте папку с именем `.template.config`.  
 12. В эту папку добавьте файл `template.json`.  
 13. В файле `template.json` измените содержимое, чтобы определить схему JSON шаблона.

```
{
    "$schema": "http://json.schemastore.org/template",
}
```

14. В середине файла `template.json` перейдите на пустую строку и нажмите сочетание клавиш `Ctrl+Пробел`, чтобы открыть интерфейс IntelliSense. Обратите внимание: схема распознана, и обязательные элементы выделены полужирным шрифтом (рис. 2.23).



**Рис. 2.23.** Определение схемы при редактировании файла `template.json`

15. Заполните оставшуюся часть файла нужными параметрами:

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Mark J. Price",
  "classifications": [ "Console", "Packt" ],
  "identity": "Packt.ConsolePlus.CSharp",
  "name": "Packt Console+ App",
  "shortName": "consoleplus",
  "tags": {
    "language": "C#",
    "type": "project"
  }
}
```

16. Сохраните файл.
17. В файле `ConsolePlusProject.csproj` проекта, если программа Visual Studio автоматически добавила элемент для включения `template.json` в сборку в качестве содержимого, оберните эту строку в XML-комментарии, чтобы исключить файл из состава пакета:

```
<!-- Если Visual Studio добавила элемент, показанный ниже, вы можете удалить его
или закомментировать, чтобы файл не включался в состав пакета шаблона -->
<!--
<ItemGroup>
  <Content Include=".template.config\template.json" />
</ItemGroup>
-->
```

18. Откройте в терминале папку проекта `ConsolePlusTemplate`.
19. Выполните команду для создания NuGet-пакета:

```
dotnet pack
```

20. Обратите внимание на результат выполнения — он должен выглядеть примерно так, как показано в следующем выводе:

```
Successfully created package 'C:\tools-skills-net8\Chapter02\
ConsolePlusTemplate\bin\Release\ConsolePlusTemplate.1.0.0.nupkg'.
```

21. Установите получившийся NuGet-пакет с помощью команды:

```
dotnet new install .\bin\Release\ConsolePlusTemplate.1.0.0.nupkg
```

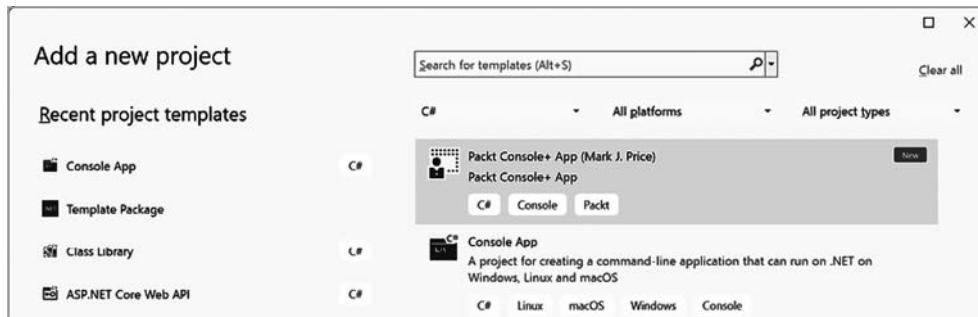
22. Посмотрите на результат выполнения команды — он должен подтверждать успешную установку шаблона:

```
Success: ConsolePlusTemplate::1.0.0 installed the following templates:
Template Name      Short Name   Language   Tags
-----          -----          -----      -----
Packt Console+ App  consoleplus  [C#]       Console/Packt
```

## Тестирование шаблона проекта

Теперь протестируем созданный шаблон.

1. В Visual Studio создайте проект и добавьте его в текущее решение.
2. Чтобы быстрее найти шаблон, введите в поле поиска слово `packt`. Если вы этого не сделаете, шаблон все равно отобразится в верхней части списка на странице `New project` (Создание нового проекта) (рис. 2.24).



**Рис. 2.24.** Поиск шаблона проекта

3. Нажмите кнопку `Next` (Далее).
4. В поле имени проекта введите `TemplateTest`, затем нажмите кнопку `Create` (Создать).
5. Просмотрите созданные файлы проекта:
  - файл проекта с импортами статических и глобальных типов;
  - файл `Program.Functions.cs`;
  - файл `Program.cs` с готовыми операторами кода.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 2.1. Дополнительные онлайн-материалы

В этой книге основное внимание уделяется программам Visual Studio и Visual Studio Code. Для аналогичного обзора инструментов в Rider вы можете ознакомиться с онлайн-разделом по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/rider/ch02-tools.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/rider/ch02-tools.md).

Чтобы узнать о последних улучшениях в Visual Studio, прочитайте статью по адресу [devblogs.microsoft.com/visualstudio/visual-studios-full-year-in-review-2023/](https://devblogs.microsoft.com/visualstudio/visual-studios-full-year-in-review-2023/).

А чтобы узнать об обновлениях в GitHub Copilot, обратитесь к публикации по адресу [devblogs.microsoft.com/visualstudio/github-copilot-in-visual-studio-a-recap-of-2023/](https://devblogs.microsoft.com/visualstudio/github-copilot-in-visual-studio-a-recap-of-2023/).

Разработчики активно внедряют инструменты искусственного интеллекта в свою работу — об этом свидетельствуют результаты опроса Stack Overflow Survey 2023 и следующие ссылки.

- Рассуждения об инструментах ИИ: [survey.stackoverflow.co/2023/#section-sentiment-and-usage-ai-tool-sentiment](https://survey.stackoverflow.co/2023/#section-sentiment-and-usage-ai-tool-sentiment).
- Преимущества использования ИИ в разработке: [survey.stackoverflow.co/2023/#section-developer-tools-benefits-of-ai-tools](https://survey.stackoverflow.co/2023/#section-developer-tools-benefits-of-ai-tools).
- Искусственный интеллект в процессе разработки: [survey.stackoverflow.co/2023/#section-developer-tools-ai-in-the-development-workflow](https://survey.stackoverflow.co/2023/#section-developer-tools-ai-in-the-development-workflow).

О сотрудничестве OpenAI и Stack Overflow по использованию вопросов и ответов Stack Overflow для обучения моделей читайте в статье по адресу [stackoverflow.co/company/press/archive/openai-partnership](https://stackoverflow.co/company/press/archive/openai-partnership).

## Упражнение 2.2. Практические задания

Чтобы повысить эффективность своей работы, создайте несколько собственных фрагментов — небольших заготовок кода, которые можно быстро вставлять в проект. Полезные статьи по работе с GitHub Copilot в Visual Studio.

- Новый интерфейс GitHub Copilot в Visual Studio: [devblogs.microsoft.com/visualstudio/introducing-the-new-copilot-experience-in-visual-studio/](https://devblogs.microsoft.com/visualstudio/introducing-the-new-copilot-experience-in-visual-studio/).
- Как пользоваться GitHub Copilot Chat в Visual Studio: [devblogs.microsoft.com/visualstudio/how-to-use-github-copilot-chat-in-visual-studio/](https://devblogs.microsoft.com/visualstudio/how-to-use-github-copilot-chat-in-visual-studio/).
- Как использовать комментарии в качестве подсказок для GitHub Copilot в Visual Studio: [devblogs.microsoft.com/visualstudio/how-to-use-comments-to-prompt-github-copilot-visual-studio/](https://devblogs.microsoft.com/visualstudio/how-to-use-comments-to-prompt-github-copilot-visual-studio/).
- Быстрее и продуктивнее с GitHub Copilot: команды с косой чертой и контекстные переменные: [devblogs.microsoft.com/visualstudio/copilot-chat-slash-commands-and-context-variables/](https://devblogs.microsoft.com/visualstudio/copilot-chat-slash-commands-and-context-variables/).
- Как освоить команды с косой чертой GitHub Copilot в Visual Studio: [devblogs.microsoft.com/visualstudio/mastering-slash-commands-with-github-copilot-in-visual-studio/](https://devblogs.microsoft.com/visualstudio/mastering-slash-commands-with-github-copilot-in-visual-studio/).

## Упражнение 2.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Опишите, как воспользоваться функцией рефакторинга Change Signature (Изменить сигнатуру) в Visual Studio. Укажите, как получить к ней доступ и какие параметры можно изменить в процессе.

2. Объясните, как используется файл `.editorconfig` для стандартизации стиля форматирования кода. Что представляет собой этот файл? Приведите пример настройки отступов в 4 пробела.
3. В чем разница между фрагментом расширения и фрагментом с добавлением кода вокруг? Как ими пользоваться? Приведите пример для каждого варианта.
4. Какой атрибут элемента `<Code>` обязателен при определении фрагмента кода? Какие значения этого атрибута встречаются чаще всего?
5. Какой шаблон необходимо установить для создания пользовательских шаблонов проектов на платформе .NET SDK?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 2.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-2---making-the-most-of-the-tools-in-your-code-editor](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-2---making-the-most-of-the-tools-in-your-code-editor).

## Резюме

В этой главе вы узнали:

- какие инструменты и функции являются общими для всех редакторов кода;
- как пользоваться встроенными инструментами Visual Studio;
- как работать с инструментами в Visual Studio Code;
- как декомпилировать .NET-сборки для изучения и анализа;
- как создать собственный шаблон проекта и применять его для ускорения разработки.

В следующей главе мы перейдем к Git — популярной системе контроля версий, которая поможет вам эффективно контролировать изменения в исходном коде.

# 3

# Управление исходным кодом с помощью Git

Эта глава посвящена *системам управления исходным кодом* (Source Control Management, SCM) на примере Git. Термины *SCM* и *VCS* (*version control system* — система контроля версий) часто используются как синонимы. Такие системы играют ключевую роль в разработке ПО — особенно в командной работе, но полезны и для индивидуальных разработчиков.

Существует множество графических интерфейсов для работы с Git, и большинство редакторов кода уже включают встроенную интеграцию. Однако в этой главе все концепции и примеры будут показаны с использованием интерфейса командной строки (CLI), так как он стабилен, универсален и одинаково работает на всех платформах.

Как только вы разберетесь в принципах Git и освоите управление через терминал, вы без труда перейдете на любой удобный графический инструмент. В конце главы приведены ссылки на дополнительные материалы, доступные онлайн, — они касаются интеграции Git с такими инструментами, как Visual Studio.

## Внедрение систем управления исходным кодом

Системы управления исходным кодом — это совокупность практик и инструментов, которые разработчики используют для отслеживания и управления изменениями в исходном коде на всех этапах жизненного цикла разработки. Такие системы особенно важны при командной работе: они позволяют безопасно вносить изменения, отслеживать историю правок и при необходимости возвращаться к предыдущим версиям кода.

## Функциональные возможности систем управления исходным кодом

Системы управления исходным кодом обеспечивают ряд ключевых возможностей.

- **Отслеживание версий.** Каждое изменение в кодовой базе — добавление, удаление или модификация файлов — фиксируется как отдельный коммит

с уникальным идентификатором. Благодаря этому можно отслеживать историю изменений и анализировать эволюцию проекта.

- **Совместная работа.** Система управления исходным кодом позволяет нескольким разработчикам работать с локальными копиями одного проекта, а затем сливать свои изменения с общей кодовой базой. Благодаря этому упрощается параллельная работа над одним проектом, независимо от географического расположения участников команды. Ранее основная ветка называлась *master*, но теперь чаще используется название *main*.
- **Разрешение конфликтов.** Когда несколько разработчиков вносят изменения в одну и ту же часть кода, система помогает обнаружить и устраниТЬ конфликт, сохраняя целостность проекта.
- **Возврат к предыдущим версиям.** Возможность отката к стабильной версии кода особенно важна при возникновении ошибок. Такой подход упрощает возврат к рабочему состоянию и отмену неудачных изменений.
- **Прозрачность изменений.** Система фиксирует, кто внес изменения и с каким комментарием, обеспечивая полную прослеживаемость и позволяя увязать изменения с конкретными задачами или требованиями.
- **Резервное копирование и восстановление.** Хотя резервное копирование файловой системы по-прежнему рекомендуется, репозиторий SCM может служить надежной резервной копией — особенно для небольших команд и индивидуальных разработчиков. В случае сбоя исходный код можно восстановить из хранилища.

Понимание принципов работы систем управления исходным кодом важно для каждого .NET-разработчика, поскольку такие системы являются неотъемлемой частью современного процесса разработки.

## Типы систем управления исходным кодом

Существуют два основных типа систем управления исходным кодом: *централизованные* (Centralized Version Control Systems, CVCS) и *распределенные* (Distributed Version Control Systems, DVCS). Рассмотрим их различия.

- **Централизованные.** В такой системе есть один центральный репозиторий, содержащий все файлы проекта и его историю. Разработчики получают файлы из этого хранилища и возвращают туда свои изменения. Центральный репозиторий считается единственным источником истины. Централизованный подход упрощает управление ветками и слияниями, поскольку изменения поступают регулярно и с меньшим расхождением во времени. Однако он создает единую точку отказа: если с центральным хранилищем возникнут проблемы, это затронет всю команду.
- **Распределенные.** Каждый участник команды имеет полную локальную копию репозитория, включая всю историю изменений. Разработчики работают неза-

висимо, коммитят, создают ветки и выполняют слияния локально, не мешая другим. Связь с центральным репозиторием требуется только для синхронизации изменений. Такой подход повышает надежность и гибкость — даже при отсутствии подключения можно продолжать работу.

В распределенных системах взаимодействие между разработчиками происходит через отправку и получение изменений из удаленных репозиториев. Именно в эти моменты выполняется слияние веток, а значит, возможны и конфликты. Такой подход позволяет участникам работать автономно и независимо друг от друга, при этом отсутствует единая точка отказа. Однако в управлении такие системы сложнее, особенно для больших команд и менее опытных разработчиков.

Выбор между централизованной и распределенной SGM зависит от характера проекта, размера команды, уровня подготовки участников и выбранного рабочего процесса. В последние годы распределенные системы получили широкое распространение благодаря своей надежности и гибкости.

## Популярные системы управления исходным кодом

Среди наиболее распространенных инструментов управления исходным кодом можно назвать следующие:

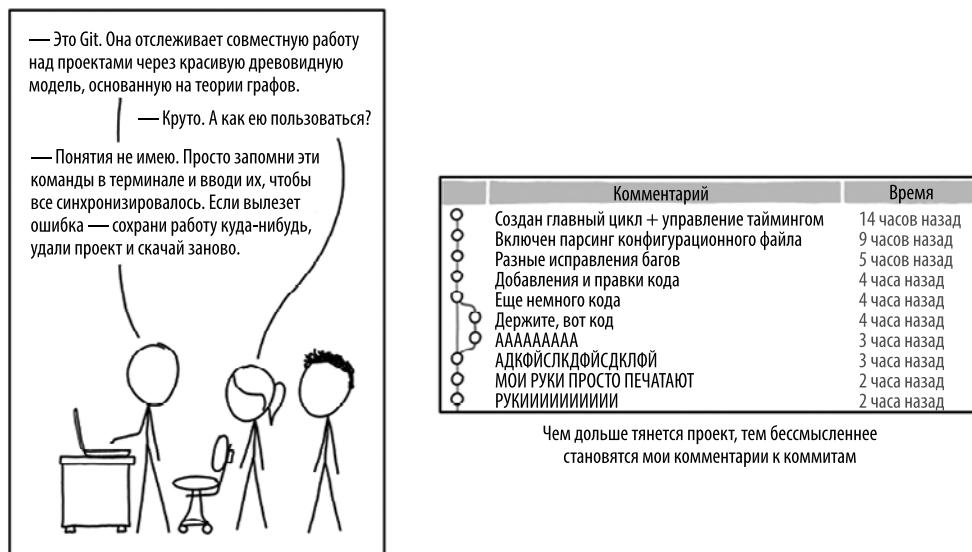
- **Mercurial** (распределенная) — [www.mercurial-scm.org/](http://www.mercurial-scm.org/);
- **Git** (распределенная) — [git-scm.com/](http://git-scm.com/);
- **Subversion (SVN)** (централизованная) — [subversion.apache.org/](http://subversion.apache.org/).

Из них наиболее популярна Git — благодаря гибкой системе ветвления и слияния, распределенной архитектуре и широким возможностям настройки под разные процессы разработки.

## Знакомство с Git

Git — самая популярная распределенная система управления исходным кодом. Она играет ключевую роль в современной разработке ПО. Система Git была создана Линусом Торвальдсом в 2005 году и известна своей скоростью, эффективностью и отказоустойчивостью.

При этом Git также славится своей сложностью. По этому поводу существует множество мемов. Один из них — про команду `git clone`, которая ведет к простой работе, и `git fork`, открывающую путь в запутанный лабиринт, намекая на сложность работы с ответвлениями репозиториев. На рис. 3.1 показаны два комикса с популярного портала xkcd.



**Рис. 3.1.** Комиксы xkcd о Git



Комикс о Git: [xkcd.com/1597/](http://xkcd.com/1597/).

Коммит в Git: xkcd.com/1296/.

Главное, что нужно понимать: чтобы уверенно работать с Git, не обязательно все о ней знать. Достаточно освоить основные команды и разобраться в базовых концепциях — это уже будет больше, чем у множества разработчиков, которые просто запоминают пару приемов и используют их вслепую.

## Основные возможности Git

Git предлагает разработчику мощный набор возможностей.

- **Распределенный контроль версий.** Git – это распределенная SGM, а значит, каждый разработчик имеет полную локальную копию репозитория с полной историей изменений. Благодаря этому можно работать автономно, не мешая остальным, что особенно удобно при децентрализованной или распределенной разработке.
  - **Ветвление и слияние.** Git предоставляет мощные средства для создания и объединения веток. Разработчик может спокойно работать над новой функцией или исправлением в отдельной ветке, не затрагивая основную кодовую базу. Когда работа завершена, изменения можно слить обратно в основную ветку.
  - **История и коммиты.** Git хранит полную историю изменений, каждое из которых оформляется как коммит с уникальным идентификатором, автором,

датой и пояснением. Такая история помогает отслеживать эволюцию проекта. Правда, как показывает рис. 3.1, со временем комментарии к коммитам могут становиться все менее осмысленными. На этом фоне особенно полезна интеграция инструментов ИИ с редактором кода — например, генерация сообщений коммитов на основе изменений. Но по-настоящему ценную информацию — зачем вносились то или иное изменение — по-прежнему может объяснить только сам разработчик. И тут без усилий не обойтись.



Большинство ИИ-интеграций с Git сейчас реализовано в графических интерфейсах и средах разработки. Прямой поддержки ИИ в терминале Git пока немного, но ее можно настроить вручную — с помощью сценариев или сторонних инструментов. Такие решения могут автоматически формировать описания для коммитов на основе внесенных изменений. Некоторые инструменты способны анализировать структуру проекта и подсказывать, когда лучше создать ветку, выполнить слияние или перебазирование. Более продвинутые ИИ-модели даже умеют оценивать потенциальные последствия слияний и заранее предупреждать о возможных конфликтах. Все это помогает сделать процесс разработки более предсказуемым и управляемым.

- **Скорость и эффективность.** Система Git разработана с прицелом на высокую производительность. Она уверенно справляется с крупными кодовыми базами, содержащими миллионы файлов и коммитов, без заметных проблем с производительностью.
- **Открытый исходный код и кросс-платформенность.** Git — это проект с открытым исходным кодом и огромным сообществом. Он работает на Windows, macOS и в большинстве дистрибутивов Linux, поэтому это отличный выбор для кросс-платформенной разработки, включая .NET. GitHub, платформа, принадлежащая Microsoft, обеспечивает хостинг репозиториев, инструменты для совместной работы и тесно интегрируется со всеми современными редакторами кода.

Git — мощный и универсальный инструмент. Хотя поначалу эта система может показаться непростой, ее освоение позволяет значительно упростить рабочий процесс и повысить эффективность командной работы. Наибольшую сложность представляет не сама Git, а выстраивание процессов — каждая команда в рамках приложений и сервисов должна самостоятельно определить подход к управлению изменениями и согласовать правила работы с репозиторием.

## Почему Git сложно изучать

Управление Git через терминал может оказаться непростой задачей. Базовые действия, такие как коммиты и перенос изменений, довольно просты. Но, как только возникает нестандартная ситуация, например случайный коммит или неудачное перебазирование, становится ясно: без понимания внутренней структуры

Git справиться непросто. Все дело в том, что Git опирается на концептуальную модель графа, разобраться в которой не так просто — об этом с иронией напоминает рис. 3.1.

Не случайно самые популярные вопросы на Stack Overflow касаются отмены коммита, изменения его описания или удаления ветки. В этой главе я постараюсь помочь вам освоить Git чуть глубже, чем просто запомнить несколько команд. Без перегруза — только то, что действительно помогает понять, как все работает. Даже если в будущем вы будете работать только с графическими инструментами, начать стоит с терминала. Так вы разберетесь, что происходит «за кадром» редактора, и поймете, как Git работает на самом деле.

## Роли в команде при работе с Git

При использовании Git важно знать две основные роли, о которых часто упоминается в официальной документации: *автор* и *коммиттер*. Автор — это тот, кто пишет код. Коммиттер — это тот, кто публикует изменения в репозитории. Часто один и тот же человек выступает в обеих ролях — сначала как автор, а затем как коммиттер. Знание этих терминов помогает лучше понимать сообщения Git и читать документацию без недоразумений.

## Загрузка последней версии Git

Возможно, Git уже установлена в вашей системе. Но чтобы убедиться, что у вас актуальная версия, ее можно загрузить с сайта [git-scm.com/downloads](https://git-scm.com/downloads).

На момент написания книги (начало 2024 года) последняя версия — 2.43.

Проверить установленную версию Git можно через терминал с помощью команды:

```
git --version
```

В операционной системе Windows результат будет выглядеть примерно так:

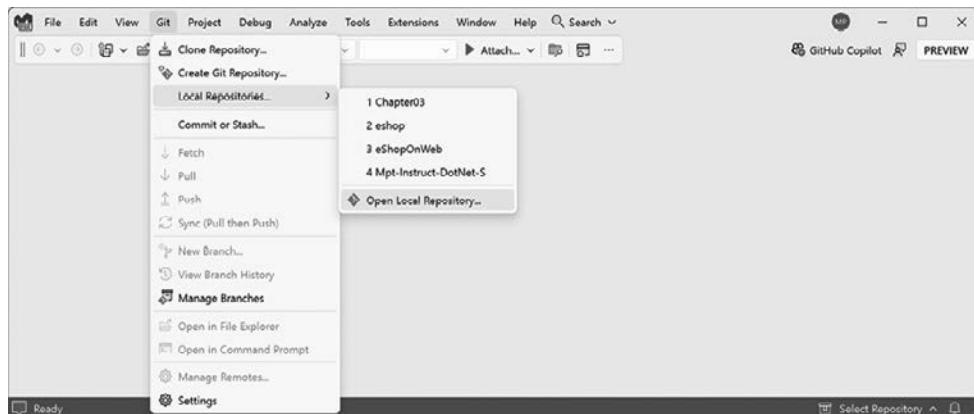
```
git version 2.43.0.windows.1
```

## Интеграция Git в Visual Studio

Git — настолько важная часть современной разработки, что в Visual Studio для нее выделен отдельный пункт меню **Git** (рис. 3.2).

На протяжении всей главы вы можете выполнять действия либо в терминале, либо с помощью встроенной поддержки Git в Visual Studio. Я считаю полезным видеть основные команды Git, даже если вы чаще пользуетесь графическим интерфейсом. Так вы лучше поймете, какие операции выполняет ваш редактор кода «за кадром». Интеграция Git с редактором упрощает работу и снижает вероятность ошибок:

интерфейс не только облегчает взаимодействие, но и может предупредить о потенциальных ошибках — в отличие от терминала, где ответственность целиком на пользователе.



**Рис. 3.2.** Меню Git в Visual Studio

## Настройка идентификационных данных Git

После установки Git нужно указать имя и адрес электронной почты — они будут использоваться при каждом коммите для фиксации автора изменений. Введите в терминале следующие команды, подставив свои данные вместо примера:

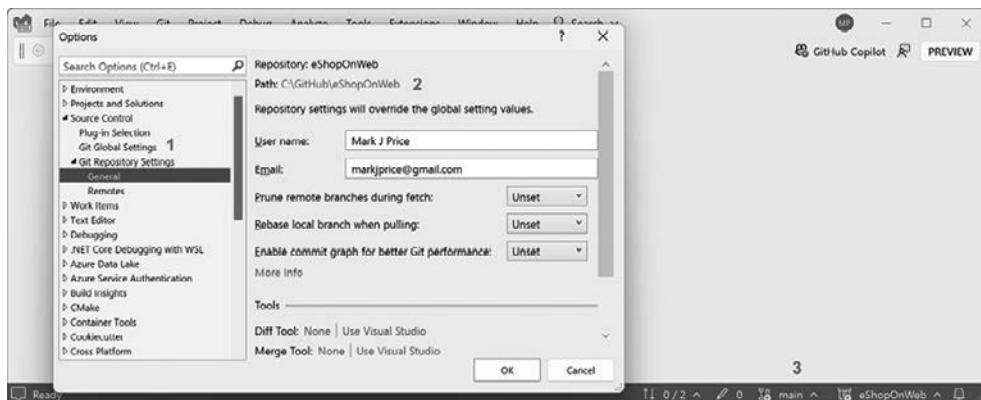
```
git config --global user.name "Mark J Price"
git config --global user.email markjprice@gmail.com
```

Если вы работаете с разными репозиториями — например, отдельно для рабочих и личных проектов, — можно задать имя и адрес электронной почты только для конкретного репозитория. Для этого перейдите в его каталог и выполните те же команды без флага `--global`:

```
git config user.name "Mark J Price"
git config user.email markp@corporation.com
```

Эти команды записывают значения конфигурации в файл `.git/config` текущего репозитория. Локальные настройки переопределяют глобальные (если они заданы), но действуют только в пределах конкретного проекта. Ввод команды `git config` без флага `--global` гарантирует, что параметры будут применяться лишь к текущему репозиторию. Такой подход дает возможность точно управлять идентификационными данными в разных проектах.

Чтобы настроить параметры Git в Visual Studio, выберите команду меню `Git > Settings` (`Git > Настройки`) (рис. 3.3).



**Рис. 3.3.** Настройка параметров Git в Visual Studio

На рисунке показаны следующие элементы:

- глобальные настройки Git;
- параметры, заданные только для текущего репозитория;
- настройки отображения состояния Git в строке состояния редактора.

## Настройка SSH-подписей

Git поддерживает использование SSH-подписей для подтверждения подлинности коммитов. Такой подход предполагает, что сервер Git настроен на прием только тех изменений, которые подписаны с помощью SSH-ключей. Если в вашем проекте такой уровень проверки не используется, настройку можно пропустить.



**Дельный совет:** SSH и SSH-подписи — это дополнительная мера безопасности, которая помогает убедиться, что коммит действительно сделан указанным автором и не был изменен после создания.

Конкретные действия зависят от того, с какой средой вы работаете — GitHub, корпоративным сервером или локальным хранилищем. Ниже приведены общие шаги настройки SSH-подписей.

1. **Создайте SSH-ключи.** Убедитесь, что у вас и у других участников команды есть рабочая пара SSH-ключей. При создании ключей используйте электронную почту в качестве метки и выберите надежный современный алгоритм Ed25519:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```



Утилита ssh-keygen доступна в Windows, macOS и Linux. Она входит в состав пакета OpenSSH, который является стандартом работы с SSH в большинстве операционных систем.



**Дельный совет:** параметр `-t ed25519` задает использование алгоритма Ed25519, который считается предпочтительным с точки зрения безопасности и производительности. В качестве альтернативы можно использовать RSA с минимальной длиной ключа 2048 бит: `-t rsa -b 2048`.

2. **Добавьте открытые ключи на Git-сервер.** Каждый участник команды должен добавить свой открытый SSH-ключ в свою учетную запись на Git-сервере. Обычно это осуществляется через пользовательские настройки Git-сервера. Например, в GitHub нужно перейти в раздел **Settings ▶ SSH and GPG keys ▶ New SSH key** (Настройки ▶ Ключи SSH и GPG ▶ Новый ключ SSH).
3. **Используйте SSH для работы с репозиторием.** На сервере стоит настроить репозиторий так, чтобы выполнение таких операций, как `git push` или `git fetch`, допускалось только по SSH. Для GitHub это значит использовать SSH-адреса и избегать HTTPS-ссылок при выполнении операций, таких как `git push` или `git fetch`.
4. **Подписание коммитов с помощью SSH-ключей.** Начиная с версии Git 2.34 появилась возможность подписывать коммиты с использованием SSH-ключей — по аналогии с GPG, но с меньшими усилиями. Чтобы включить эту функцию, обновите Git до актуальной версии и настройте использование SSH-ключа для подписи. Благодаря этому можно проверять авторство коммитов без дополнительных инструментов.

```
git config --global user.signingkey <your-ssh-public-key-id>
git config --global commit.gpgformat ssh
```

После настройки можно подписывать коммиты с помощью ключа, добавив флаг `-S` при выполнении команды:

```
git commit -S -m "your commit message"
```

Проверку подписи также можно выполнить:

```
git log --show-signature
```

Если необходимо обеспечить серверную проверку SSH-подписей, можно использовать серверные хуки, например `pre-receive`, чтобы отклонять неподписанные или недействительные коммиты. Такая проверка не всегда включена по умолчанию, поэтому может потребоваться дополнительная настройка или собственный сценарий для валидации подписи.

## Настройка ветки по умолчанию

Сначала проверьте текущую конфигурацию Git и файлы, в которых она определена:

```
git config --list --show-origin
```

В выводе можно увидеть имя и адрес электронной почты, заданные ранее, а также имя ветки по умолчанию — обычно это `master`, как показано в примере (фрагмент) ниже:

```
file:C:/Program Files/Git/etc/gitconfig diff.astextplain.textconv=astextplain
...
file:C:/Program Files/Git/etc/gitconfig init.defaultbranch=master
...
file:C:/Users/markj/.gitconfig user.name=Mark J Price
file:C:/Users/markj/.gitconfig user.email=markjprice@gmail.com
```

Одни параметры задаются в конфигурации системы, другие — в пользовательском файле настроек.

Если вы хотите изменить имя ветки по умолчанию с `master` на более нейтральное, например `main`, выполните соответствующую команду для обновления конфигурации:

```
git config --global init.defaultBranch main
```



Система Git использует последнее найденное значение для каждого уникального параметра. Поэтому настройки, заданные в конфигурации пользователя, будут иметь приоритет над системными.

## Как получить справку по Git

Чтобы открыть веб-страницу со справкой по нужной команде Git, используйте одну из следующих команд:

```
git help <команда>
git <команда> --help
```

Чтобы получить краткую сводку параметров команды прямо в терминале, выполните:

```
git <команда> -h
```

## Основы работы с Git

Теперь, когда настройка Git завершена и основные концепции уже знакомы, самое время перейти к практике и попробовать себя в работе с репозиторием Git.

## Начало работы с репозиторием Git

Существуют два основных способа начать работу с репозиторием Git:

- инициализировать репозиторий в уже существующей локальной папке;
- клонировать удаленный репозиторий в локальный каталог.

Чтобы клонировать репозиторий, сначала создайте нужную папку и перейдите в нее. Затем выполните команду `git clone`, указав адрес удаленного хранилища.

```
git clone https://github.com/markjprice/tools-skills-net8
```

Если вы хотите превратить существующую папку в репозиторий Git, достаточно перейти в нее и выполнить команду `git init`.

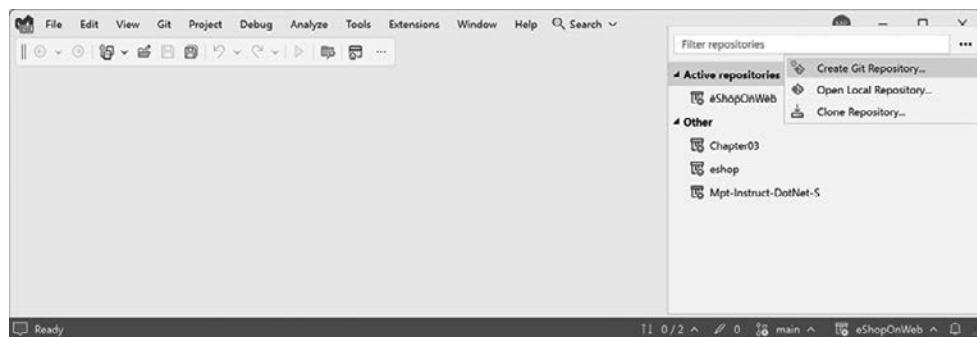
```
git init
```

Команда `init` создает скрытую подпапку с именем `.git`, в которой Git хранит все данные, связанные с отслеживанием версий, как показано в примере ниже:

```
Initialized empty Git repository in C:/tools-skills-net8/Chapter03/RepoDemo/.git/
```

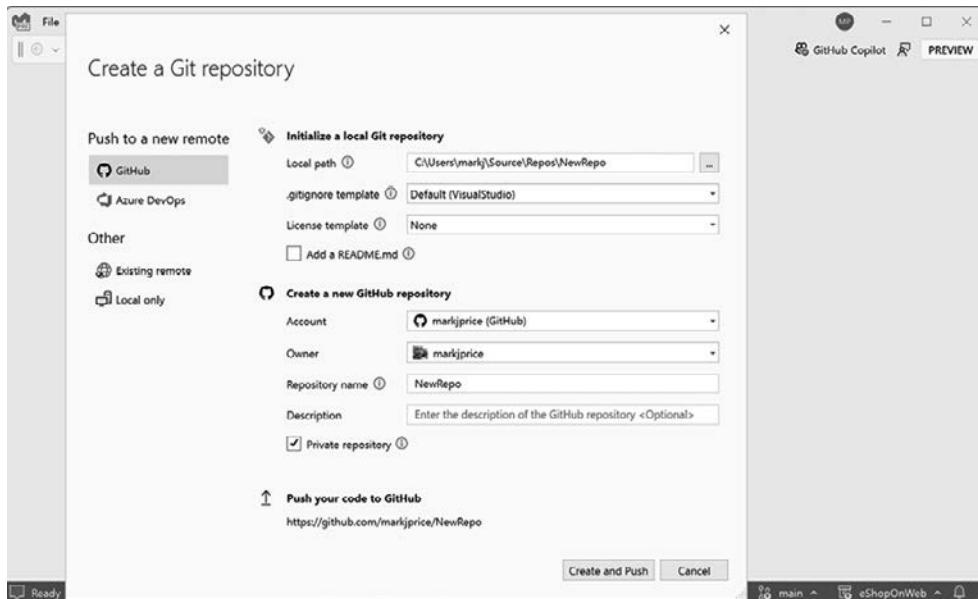
В Visual Studio можно создать новый репозиторий Git одним из следующих способов.

- Выберите команду меню `Git > Create New Repository` (`Git > Создать новый репозиторий`), если в данный момент не открыт ни один репозиторий.
- Щелкните на названии текущего репозитория в строке состояния, затем — нажмите кнопку с тремя точками и выберите пункт `Create Git Repository` (`Создать репозиторий Git`) (рис. 3.4).



**Рис. 3.4.** Меню для создания нового репозитория Git в Visual Studio

После этого откроется диалоговое окно Create a Git repository (Создать репозиторий Git), в котором нужно задать параметры (рис. 3.5).



**Рис. 3.5.** Создание нового репозитория Git в Visual Studio



Выберите вариант Local only (Только локальный), чтобы скрыть параметры для создания удаленного репозитория на GitHub. Удобно, если вы хотите начать с локального проекта.

## Создание и добавление файлов в репозиторий Git (теоретически)

На следующем этапе нужно создать файлы проекта — например, исходный код и вспомогательные документы — и добавить их в репозиторий.

Если в папке уже есть нужные файлы, их потребуется вручную добавить в отслеживаемые. Таким образом, Git начнет следить за изменениями в этих файлах. Например, можно добавить все файлы с расширением .cs, а также файл readme.md.

```
git add *.cs
git add readme.md
```

После этого нужно зафиксировать изменения (выполнить коммит) в локальном репозитории. Лучше сразу указать сообщение коммита с помощью параметра `-m`, чтобы пояснить, что было добавлено.

```
git commit -m 'Initial version'
```

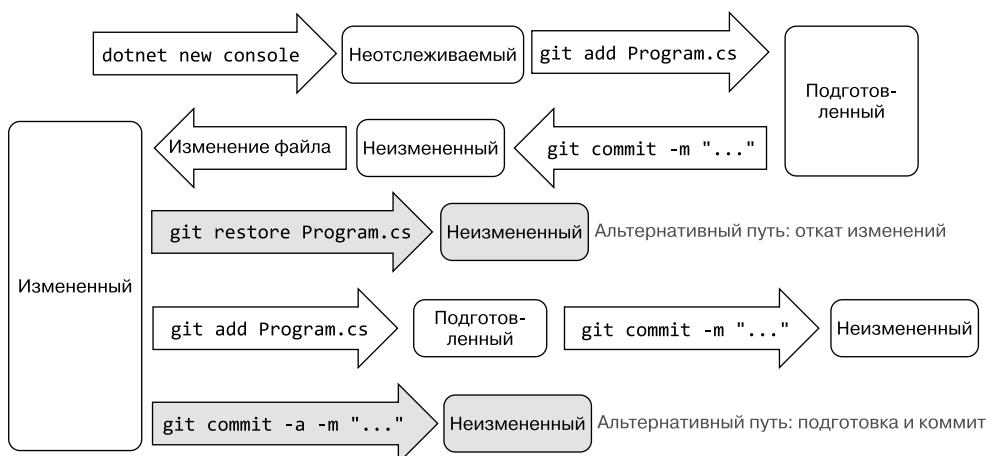
## Отслеживание изменений в Git

Файл в каталоге, используемом как репозиторий Git, может находиться в одном из четырех состояний. Эти состояния описаны в табл. 3.1.

**Таблица 3.1.** Состояния файлов в репозитории Git

Состояние	Описание
Неотслеживаемый	Файл, за которым Git пока не следит
Подготовленный	Файл, добавленный в индекс (например, с помощью команды <code>git add</code> ), но еще не закоммиченный
Неизмененный	Отслеживаемый файл, который не менялся после последнего коммита
Измененный	Отслеживаемый файл, в который были внесены изменения

Сводная диаграмма потока состояний показана на рис. 3.6.



**Рис. 3.6.** Поток состояний файлов в Git



Область подготовки можно пропустить, добавив ключ -a к команде commit. В этом случае Git автоматически добавит в коммит все измененные файлы, которые уже отслеживаются.

Теперь посмотрим, как этот процесс работает на практике.

## Создание репозитория Git на практике

Чтобы выполнить действия с помощью терминала Git, сделайте следующее.

1. Создайте каталог с именем **Chapter03** внутри каталога **tools-skills-net8**.
2. В терминале перейдите в папку **Chapter03**.
3. Введите команду для проверки текущего состояния — так вы узнаете, отслеживается ли каталог системой Git.

```
git status
```

4. Изучите результат — он приведен в примере ниже.

```
fatal: not a git repository (or any of the parent directories): .git
```

5. Инициализируйте папку как репозиторий Git с помощью команды **git init**.

```
git init
```

6. Обратите внимание на результат и сообщение, которое Git выводит после инициализации — см. пример ниже.

```
Initialized empty Git repository in C:/tools-skills-net8/Chapter03/.git/
```

Если вы пользуетесь Visual Studio, выполните следующее.

1. Запустите Visual Studio (если она уже открыта, сначала закройте ее).
2. В диалоговом окне **Start** (Начало работы) выберите пункт **Continue without code** (Продолжить без кода).
3. Выберите команду меню **Git ▶ Create Git Repository** (**Git ▶ Создать репозиторий Git**).
4. В окне **Create a Git repository** (Создать репозиторий Git) выберите вариант **Local only** (Только локально), затем задайте следующие параметры:
  - **Local path** (Локальный путь): **C:\tools-skills-net8\Chapter03**;
  - **.gitignore template** (Шаблон .gitignore): **No .gitignore** (Нет шаблона .gitignore) — мы добавим его позже;
  - **License template** (Шаблон лицензии): **None** (Отсутствует);
  - **Add a README.md** (Добавить README.md): флагок снят.
5. Нажмите кнопку **Create** (Создать).

## Создание проекта

Теперь создадим файлы нового проекта и добавим их в репозиторий.

В терминале сделайте следующее.

1. Создайте консольное приложение во вложенной папке `RepoDemo`.

```
dotnet new console -o RepoDemo
```

2. Создайте файл решения с именем `Chapter03.sln` и добавьте в него проект `RepoDemo`.

```
dotnet new sln  
dotnet sln add RepoDemo
```

3. Проверьте текущий Git-статус каталога `Chapter03`.

```
git status
```

4. Изучите результат — он приведен в примере ниже.

```
On branch main  
  
No commits yet  
  
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  RepoDemo/  
  
nothing added to commit but untracked files present (use "git add" to track)
```

5. Добавьте в область подготовки папку `RepoDemo` и все вложенные файлы.

```
git add RepoDemo
```

6. Еще раз проверьте текущий Git-статус каталога `Chapter03`.

```
git status
```

7. Взгляните на результат — он приведен ниже.

```
On branch main  
  
No commits yet  
  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
  new file: RepoDemo/Program.cs  
  new file: RepoDemo/RepoDemo.csproj  
  new file: RepoDemo/obj/RepoDemo.csproj.nuget.dgspec.json  
  new file: RepoDemo/obj/RepoDemo.csproj.nuget.g.props
```

```
new file: RepoDemo/obj/RepoDemo.csproj.nuget.g.targets
new file: RepoDemo/obj/project.assets.json
new file: RepoDemo/obj/project.nuget.cache
```



Обратите внимание! Мы не охватили файлы из каталога `obj`, поскольку это не исходный код. Эти файлы автоматически создаются компилятором C# Roslyn.

8. Рекурсивно удалите все файлы в папке `RepoDemo` с помощью ключа `-r`.

```
git rm --cached RepoDemo -r
```

9. Добавьте в область подготовки только два ключевых исходных файла из `RepoDemo`, как указано в следующих двух командах.

```
git add RepoDemo/Program.cs
git add RepoDemo/RepoDemo.csproj
git add Chapter03.sln
```

10. Проверьте Git-статус каталога `Chapter03`.

```
git status
```

11. Ознакомьтесь с результатом — он показан в примере ниже.

```
On branch main
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file: RepoDemo/Program.cs
    new file: RepoDemo/RepoDemo.csproj
    new file: Chapter03.sln

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    RepoDemo/obj/
```

12. Проверьте статус, указав короткий ключ `-s`, чтобы вывести краткую информацию.

```
git status -s
```

13. Взгляните на результат — он приведен ниже.

```
A RepoDemo/Program.cs
A RepoDemo/RepoDemo.csproj
A Chapter03.sln
?? RepoDemo/obj/
```



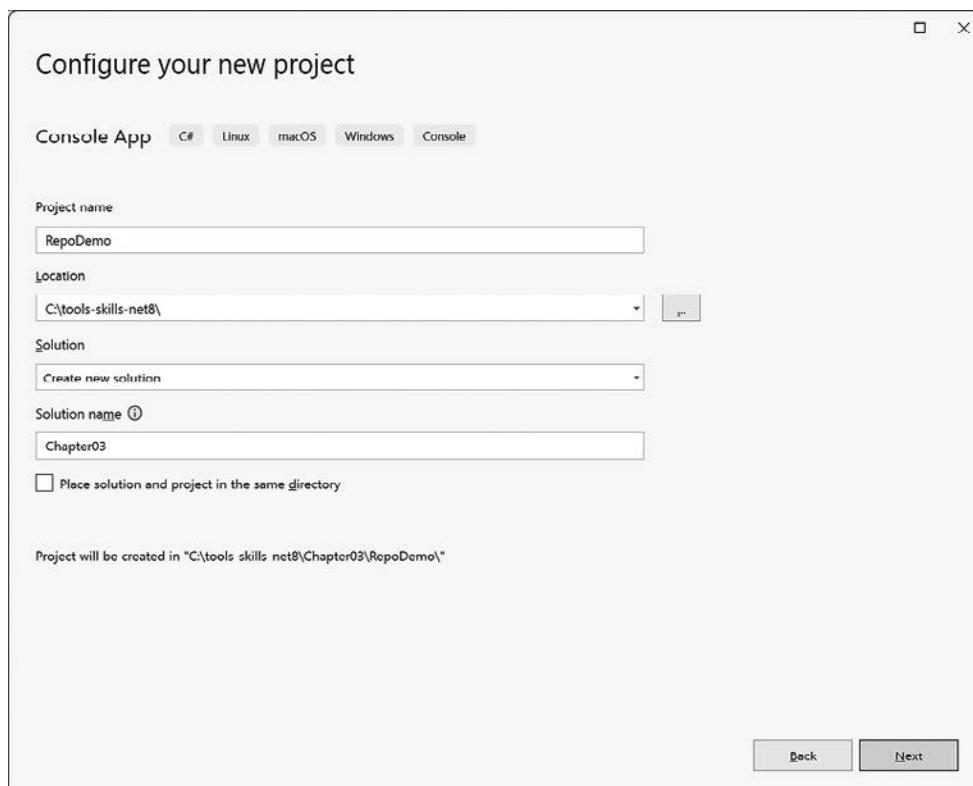
В коротком выводе статуса (`git status -s`) слева отображаются один или два символа. Левый символ указывает на состояние файла в области подготовки (индексе), правый — на состояние файла в рабочем дереве.

- ? означает неотслеживаемый файл — в зависимости от положения символа либо в рабочем дереве, либо в индексе.
- A означает, что файл подготовлен (добавлен в индекс).
- M указывает на то, что файл был изменен.

В предыдущем примере изменений не было, поэтому символ M не отображается.

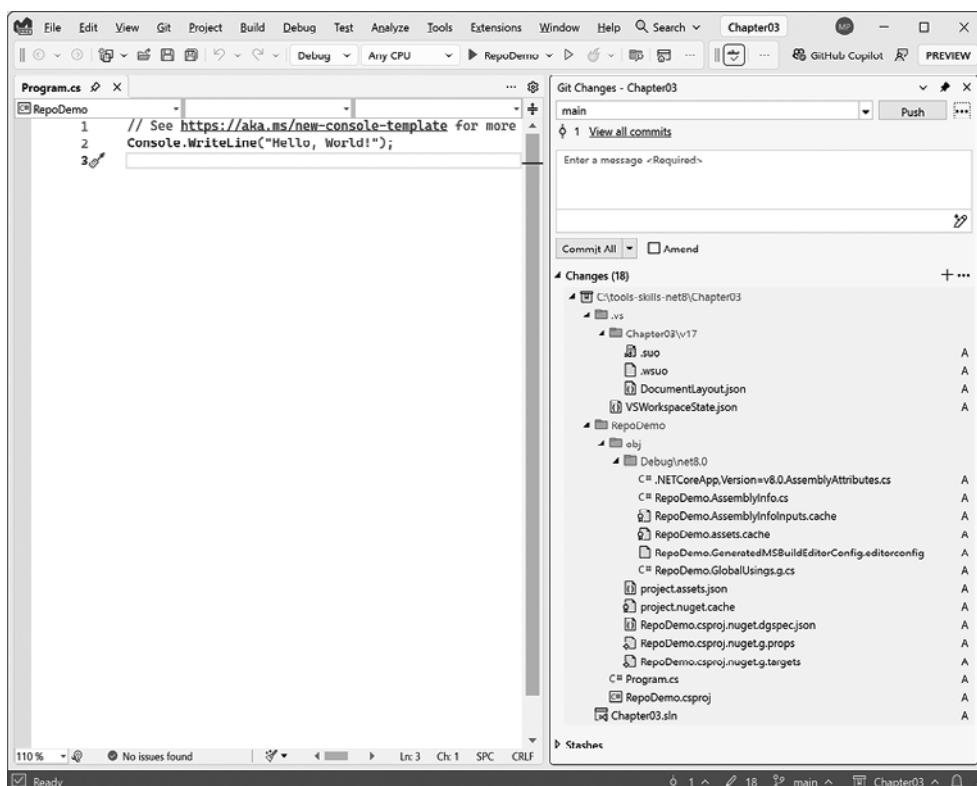
В Visual Studio нужно сделать следующее.

1. Выберите команду меню **File ▶ New ▶ Project** (Файл ▶ Новый ▶ Проект).
2. Выберите шаблон **Console App** (Консольное приложение) и нажмите кнопку **Next (Далее)**.
3. Заполните параметры в окне **Configure your new project** (Настройка нового проекта) (рис. 3.7).



**Рис. 3.7.** Создание проекта консольного приложения и решения

4. Нажмите кнопку Next (Далее), затем кнопку Create (Создать).
5. В строке состояния нажмите кнопку Changes (Изменения), после чего в окне Git Changes (Изменения Git) можно просмотреть текущие изменения (рис. 3.8).



**Рис. 3.8.** Окно Git Changes в Visual Studio



Иногда в репозиторий случайно попадают лишние файлы — именно поэтому так важно добавить файл .gitignore. Мы сделаем это чуть позже.

6. Наведите указатель мыши на нужный файл и щелкните на значке +, чтобы добавить его в область подготовки. Повторите действие для файлов `RepoDemo.csproj` и `Chapter03.sln`, чтобы в области подготовки оказались три файла (рис. 3.9).

Перемещение файлов в область подготовки — лишь первый шаг. Теперь пора выполнить коммит.

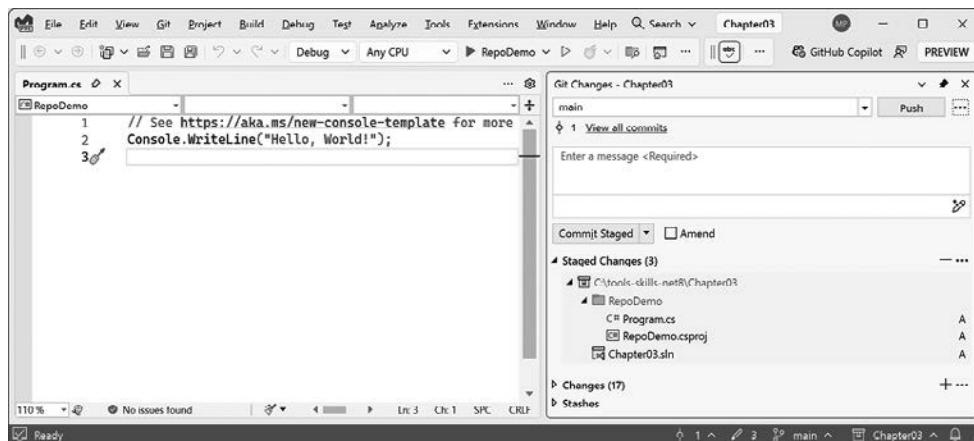


Рис. 3.9. Файлы в области подготовки (индексе) в Visual Studio

## Коммиты файлов

Теперь зафиксируем изменения в файлах, добавленных в область подготовки.

В терминале сделайте следующее.

- Выполните коммит двух файлов, добавив поясняющее сообщение.

```
git commit -m "Initial version"
```

- Изучите результат — он приведен в следующем выводе.

```
[main (root-commit) 6549a99] Initial version
3 files changed, 63 insertions(+)
create mode 100644 RepoDemo/Program.cs
create mode 100644 RepoDemo/RepoDemo.csproj
create mode 100644 Chapter03.sln
```



При выполнении коммита Git сохраняет моментальный снимок состояния файлов, к которому в будущем можно вернуться или с которым можно сравнить последующие изменения.

Если вы работаете в Visual Studio, сделайте следующее.

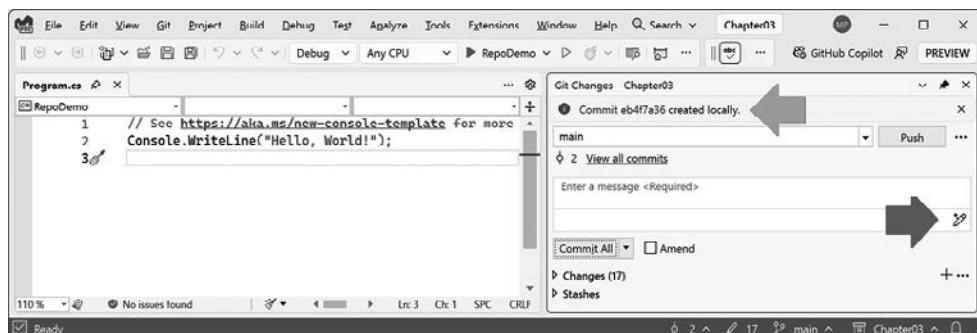
- В верхней части окна Git Changes (Изменения Git) введите сообщение коммита — например *Initial version* — и нажмите кнопку Commit Staged (Коммит подготовленных файлов).

Обратите внимание на значок в виде «карандаша с блестками» — он используется для добавления сообщения, сгенерированного искусственным интеллектом (рис. 3.10, стрелка, направленная вправо).



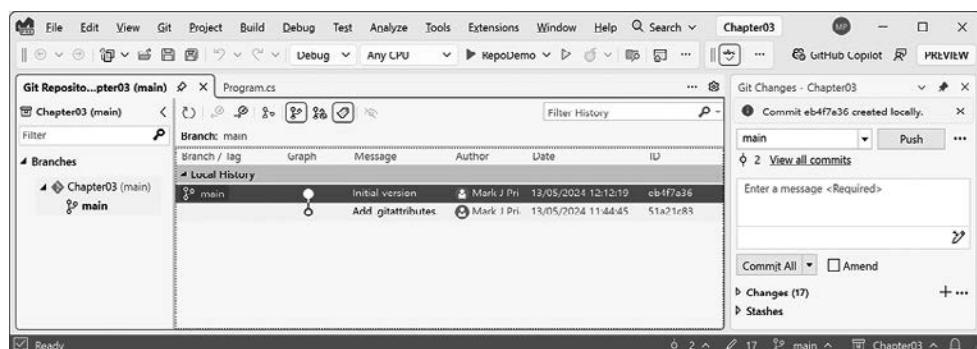
Кнопка обозначается как Commit Staged (Коммит подготовленных файлов), если в область подготовки добавлено лишь несколько файлов. Если вы не выделяли файлы для подготовки, будет отображаться кнопка Commit All (Коммит всех файлов).

- После нажатия появится уведомление об успешной локальной фиксации (см. рис. 3.10, стрелка, направленная влево).



**Рис. 3.10.** Информационное сообщение о коммите в Visual Studio

- Нажмите кнопку View all commits (Просмотреть все коммиты). В окне Git Repository — Chapter 3 (main) будет показан список коммитов — последний находится вверху и содержит сообщение Initial version (Начальная версия) (рис. 3.11).



**Рис. 3.11.** Окно Git Repository в Visual Studio

## Отмена коммита

Git позволяет отменить коммит разными способами — в зависимости от того, нужно ли сохранить изменения, удалить их или просто скорректировать сообщение. Несколько распространенных методов представлены в табл. 3.2.

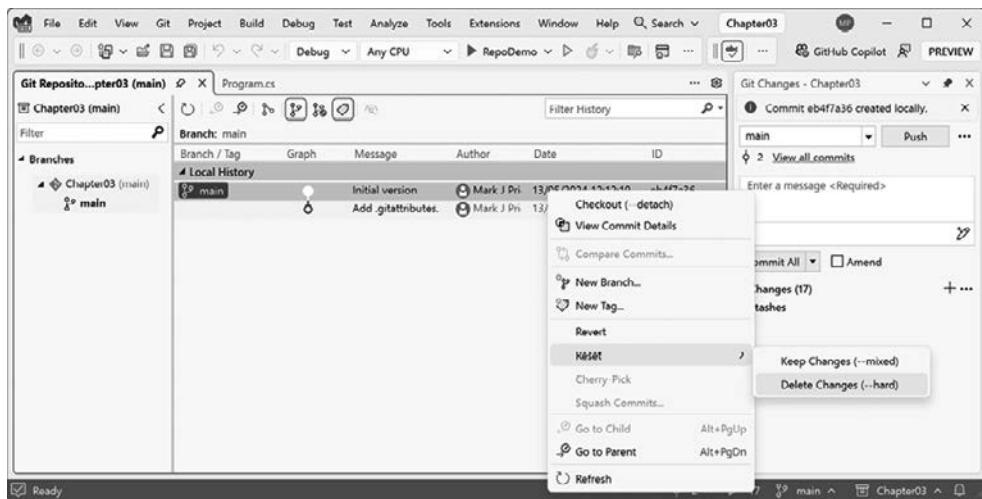
**Таблица 3.2.** Команды Git для отмены коммитов

Команда Git	Описание
<code>git reset --soft HEAD</code>	<b>Мягкий сброс.</b> Отменяет последний коммит, но сохраняет изменения в области подготовки. Можно повторно ввести сообщение или внести дополнительные правки
<code>git reset HEAD</code>	<b>Смешанный сброс.</b> Удаляет коммит и переносит изменения обратно в рабочий каталог, оставляя их неотслеживаемыми
<code>git reset --hard HEAD</code>	<b>Жесткий сброс.</b> Полностью удаляет последний коммит и любые изменения в рабочем каталоге. Используйте с осторожностью — можно потерять ценные файлы
<code>git revert &lt;хеш_коммита&gt;</code>	<b>Откат коммита.</b> Создает коммит, который отменяет изменения из указанного коммита. Подходит для совместной работы, так как история не переписывается
<code>git commit --amend</code>	<b>Изменение коммита.</b> Позволяет скорректировать последний коммит, например исправить сообщение. Меняет историю, поэтому не рекомендуется в общих репозиториях

Выбирайте подходящий способ отмены коммита в зависимости от задачи.

- Если вы работаете над проектом в одиночку и хотите полностью удалить последний коммит, используйте жесткий сброс.
- Если необходимо сохранить изменения для доработки, подойдет мягкий или смешанный сброс.
- Если вы работаете в команде и важно сохранить историю репозитория, безопаснее использовать команды `revert` или `amend`.

Если вы работаете в Visual Studio, откройте окно `Git Repository` (Репозиторий Git), щелкните правой кнопкой мыши по нужному коммиту и выберите доступные действия — например, сброс или удаление изменений (рис. 3.12).



**Рис. 3.12.** Контекстное меню для коммитов в Git, включая возможности отмены и сброса в Visual Studio

## Очистка коммита

Если нужно очистить рабочий каталог, удалив не отслеживаемые системой Git файлы — например, остатки сборки или временные файлы после конфликтов слияния, — воспользуйтесь командой `git clean`.

Эта команда навсегда удаляет файлы, которые не входят в индекс (область подготовки) и не были включены ни в один коммит. Так вы сможете поддерживать порядок в рабочем каталоге.

К примеру, `git clean -d` удаляет не только неотслеживаемые файлы, но и неотслеживаемые каталоги.



**Внимание!** Команда `git clean` может удалить файлы безвозвратно. Чтобы сначала проверить, что будет удалено, используйте параметр `-n` — он выполнит пробный запуск и выведет список файлов. После проверки можно выполнить команду с параметром `-f`, чтобы действительно удалить файлы.

Прежде чем применять `git clean` к важным данным, убедитесь, что у вас есть резервная копия или сохраненная версия файлов в надежном месте.

## Сохранение

В Git *сохранение (stashing)* — это удобная возможность временно отложить изменения, которые вы внесли, но еще не готовы зафиксировать. Благодаря такому подходу можно быстро переключиться на другую задачу, не теряя при этом текущий прогресс. Сохранение особенно полезно, когда необходимо временно

очистить рабочий каталог, переключиться на другую ветку, получить обновления или внести срочные правки в другом месте.

Когда вы выполняете команду `git stash`, Git сохраняет как подготовленные, так и неподготовленные изменения в специальной структуре, похожей на стек, — это и есть хранилище `stash`. После этого рабочий каталог возвращается к состоянию последнего коммита (`HEAD`).

Изменения не теряются — они просто откладываются и остаются доступными для восстановления в любой момент.

Сохранить текущие изменения может понадобиться по разным причинам. Ниже приведены наиболее распространенные ситуации.

- **Переключение контекста.** Если нужно срочно перейти на другую ветку и заняться другой задачей, можно временно сохранить текущие изменения, переключиться, а затем восстановить их позже.
- **Получение обновлений.** Если в рабочем каталоге есть незавершенные правки, которые могут конфликтовать с изменениями из основной ветки, сначала сохраните свои изменения, получите обновления, а потом примените сохраненное.
- **Эксперименты.** Когда хочется опробовать новую идею, но нет уверенности, что изменения пригодятся, можно сохранить текущую версию, попробовать что-то новое, а затем либо восстановить сохраненное, либо отбросить его.

Основные команды Git для сохранения изменений приведены в табл. 3.3.

**Таблица 3.3.** Основные команды Git для сохранения изменений

Команда	Описание	Пример
Сохранение изменений	Сохраняет изменения. С помощью ключа <code>-m</code> можно добавить описание для удобной идентификации. Если изменения не указаны, сохраняются все	<code>git stash push -m "Описательное сообщение"</code>
Перечисление сохранений	Отображает список сохраненных изменений. Каждая запись индексируется и показывает сообщение, если оно задано	<code>git stash list</code>
Применение последнего	Повторно применяет последнее сохранение, но не удаляет его из списка	<code>git stash apply</code>
Применение по индексу	Применяет конкретное сохранение по индексу	<code>git stash apply stash@{0}</code>
Извлечение сохранения	Применяет последнее сохранение и удаляет его из списка. Можно указать индекс	<code>git stash pop</code>
Удаление сохранения	Удаляет конкретное сохранение. Если индекс не указан, удаляется последнее	<code>git stash drop stash@{0}</code>
Очистка всех сохранений	Полностью очищает список сохранений	<code>git stash clear</code>

Вы можете сохранить только отдельные файлы или изменения с помощью команды `git stash push ваш_файл.txt` — в этом случае сохраняются только изменения, связанные с указанным файлом.



**Дельный совет:** если вы часто используете сохранения, список может быстро переполниться устаревшими записями. Регулярно просматривайте и удаляйте устаревшие записи, чтобы сохранить порядок.

Важно помнить: сохранения — это локальная информация. Они не передаются на сервер при выполнении команды `push`. Если вы часто полагаетесь на команду `stash`, убедитесь, что у вас есть резервная копия на случай сбоя.

Сохранения позволяют удобно переключаться между задачами, отслеживая прогресс и поддерживая порядок в рабочем каталоге.

Если вы работаете в Visual Studio, в окне Git Changes (Изменения Git) у кнопки Commit All / Commit Staged (Коммит всех файлов / Коммит подготовленных файлов) есть дополнительные параметры для сохранения изменений:

- Stash All (Сохранить все) — включает неотслеживаемые файлы (`--include-untracked`);
- Stash All and Keep Staged (Сохранить все и оставить подготовленные файлы в индексе) — (`--keep-index`).

## Игнорирование файлов

Файлы, созданные компилятором Roslyn C#, например содержимое каталогов `obj` и `bin`, не следует добавлять в репозиторий. Чтобы указать Git, какие файлы следует игнорировать, создается файл с именем `.gitignore`.

Правила для файла `.gitignore`.

- Пустые строки и строки, начинающиеся с `#` (считываются комментариями), игнорируются.
- Используются упрощенные шаблоны с метасимволами: `*` — любое количество символов, `[aeiou]` — любой из указанных символов, `?` — любой один символ.
- По умолчанию шаблоны применяются ко всей структуре каталогов рекурсивно. Чтобы ограничить правило текущей директорией, начинайте шаблон с символа `/`.
- Завершение шаблона символом `/` указывает, что речь идет о каталоге.
- Символы `**` используются для соответствия вложенным каталогам, например `a/**/z` соответствует и `a/c/z`, и `a/d/e/f/z`.
- Чтобы отменить игнорирование, поставьте перед шаблоном `!`.

Пример файла `.gitignore` приведен ниже:

```
# Игнорировать все файлы, в имени которых содержится .nuget, например
# project.nuget.cache и RepoDemo.csproj.nuget.dgspec.json.
*.nuget.*

# Игнорировать все PDF-файлы в каталоге doc и его вложенных папках.
doc/**/*.pdf

# Игнорировать все файлы в каталогах с именами obj и bin.
obj/
bin/
```



**В копилку знаний:** полный набор готовых примеров для `.gitignore` опубликован на сайте [github.com/github/gitignore](https://github.com/github/gitignore).

Интерфейс .NET из SDK может создать подходящий файл `.gitignore`, оптимизированный для проектов на .NET. Visual Studio также предлагает создать его при инициализации репозитория, но мы пропустили этот шаг.

Если репозиторий Git уже существует, вот как можно добавить `.gitignore`.

1. Открыв каталог `Chapter03` в терминале, выполните команду для создания файла `.gitignore` с помощью интерфейса .NET SDK:

```
dotnet new gitignore
```

2. Откройте файл в любом текстовом редакторе и изучите его содержимое. Ниже приведен пример:

```
## Игнорировать временные файлы Visual Studio, результаты сборки
## и файлы, создаваемые популярными расширениями для Visual Studio.
##
## Содержимое получено с помощью команды `dotnet new gitignore`

# Файлы конфигурации окружения
.env

# Файлы, специфичные для пользователя
*.rsuser
*.suo
*.user
*.userosscache
*.sln.docstates

# Результаты сборки
[Dd]ebug/
[Dd]ebugPublic/
```

```
[Rr]elease/
[Rr]eleases/
...
[Bb]in/
[Oo]bj/
[Ll]og/
[Ll]ogs/
...
```

3. Закройте файл `.gitignore`.
4. В терминале выполните команду для получения текущего Git-статуса в каталоге `Chapter03`:

```
git status
```

5. В результате видно, что файлы в папке `obj` больше не отслеживаются, а сам файл `.gitignore` распознан Git — см. вывод ниже:

```
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```



**Дельный совет:** файл `.gitignore` нужно зафиксировать в репозитории, чтобы его правила применялись ко всей команде. При этом не стоит слепо использовать готовый шаблон — адаптируйте его под конкретные требования вашего проекта.

6. Если вы работаете в терминале, добавьте файл `.gitignore` в область подготовки, а затем зафиксируйте его:

```
git add .gitignore
git commit -m "Add Git ignore file."
```

7. В Visual Studio вы увидите, что все файлы, сгенерированные компилятором, больше не отображаются как измененные — они теперь игнорируются. Осталось зафиксировать сам файл `.gitignore`.

Это хороший повод попробовать сообщение о коммите, сгенерированное искусственным интеллектом. Нажмите «волшебную кнопку» — значок карандаша с блестками (помните, мы уже обращали на него внимание?) — и просмотрите предложенный вариант (рис. 3.13).

8. Нажмите кнопку `Insert AI Suggestion` (Вставить предложение ИИ) или `Discard` (Отклонить), чтобы ввести свое сообщение вручную.
9. Нажмите кнопку `Commit All` (Коммит всех файлов), чтобы завершить коммит.

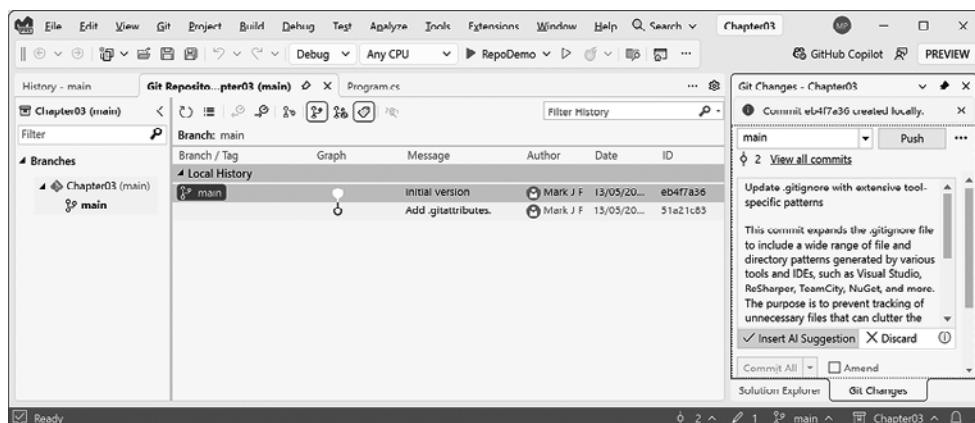


Рис. 3.13. Коммит файла .gitignore в Visual Studio

## Просмотр репозиториев Git

При работе с Git-репозиториями часто возникает необходимость просмотреть изменения в файлах.

### Просмотр различий в файлах

Чтобы узнать, какие строки были изменены, но еще не помещены в область подготовки, или какие подготовлены, но не зафиксированы, можно использовать команду `git diff`. Она показывает добавленные и удаленные строки — так называемый *патч*.

#### Пример

- Чтобы упростить работу с проектом, откройте решение `Chapter03` в Visual Studio или Rider либо откройте папку `Chapter03` в редакторе Code.
- В проекте `RepoDemo` измените строку в файле — замените сообщение `Hello, World!` на `Hello, Git!`.

```
Console.WriteLine("Hello, Git!");
```

- Сохраните файл.
- В терминале выполните команду для просмотра различий:

```
git diff
```

- Взгляните на результат в терминале и на рис. 3.14.

```
diff --git a/RepoDemo/Program.cs b/RepoDemo/Program.cs
index 3751555..4ec9f4e 100644
--- a/RepoDemo/Program.cs
+++ b/RepoDemo/Program.cs
```

```

@@ -1,2 +1,2 @@
// See https://aka.ms/new-console-template for more information
-Console.WriteLine("Hello, World!");
+Console.WriteLine("Hello, Git!");

```

```

Developer Command Prompt for VS 2022
C:\tools-skills-net8\Chapter03>git diff
diff --git a/RepoDemo/Program.cs b/RepoDemo/Program.cs
index 3751555..4ec9f4e 100644
--- a/RepoDemo/Program.cs
+++ b/RepoDemo/Program.cs
@@ -1,2 +1,2 @@
// See https://aka.ms/new-console-template for more information
-Console.WriteLine("Hello, World!");
+Console.WriteLine("Hello, Git!");

```

Рис. 3.14. Просмотр различий в терминале

6. Если вы работаете в Visual Studio, выберите команду меню View ▶ Git Changes (Вид ▶ Изменения Git), а затем щелкните кнопкой мыши на файле, чтобы просмотреть изменения (рис. 3.15).

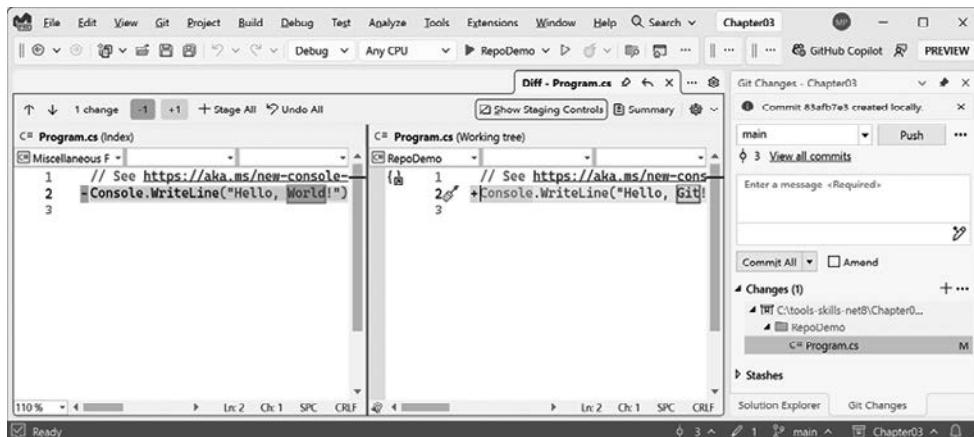


Рис. 3.15. Просмотр различий в Visual Studio



Если вы используете редактор Code, выберите команду меню View ▶ Source Control (Вид ▶ Управление исходным кодом), а затем щелкните на файле Program.cs, чтобы увидеть изменения (рис. 3.16).

В программе Code отображаются два неотслеживаемых файла — .gitignore и Chapter03.sln, помеченных символом U, а измененный файл — символом M. Наведите указатель мыши на файл: нажатие кнопки + отправляет файл в область подготовки, а кнопка со стрелкой отменяет изменения.



**Рис. 3.16.** Просмотр различий в редакторе Code

- Чтобы просмотреть различия, уже перемещенные в область подготовки, выполните в терминале следующую команду:

```
git diff --staged
```



Ключи `--staged` и `--cached` эквивалентны — можно использовать любой из них.

- Обратите внимание: в данный момент в области подготовки нет изменений. В Visual Studio окно Git Changes (Изменения Git) содержит раздел Staged Changes (Изменения в области подготовки), который автоматически скрывается, если в нем ничего нет.
- Внесите изменения в файл:

```
git add RepoDemo/Program.cs
```

Затем выполните команду, чтобы просмотреть подготовленные различия:

```
git diff --staged
```

- Вы увидите первую страницу вывода. Если команда возвращает много строк, вывод отображается с помощью встроенного постраничного просмотра (пейджера). В этом режиме можно использовать клавиши:

- `Page Down` — перейти к следующей странице;
- `Page Up` — вернуться на страницу назад;
- `Q` — выйти из режима просмотра в любой момент.

Когда появится надпись (`END`), нажмите клавишу `Q`, чтобы завершить просмотр.



Для отображения длинных выводов в терминале Git по умолчанию задействует режим постраничного просмотра. Чтобы отключить его, добавьте параметр `--no-pager` сразу после команды `git`. Например: `git --no-pager diff --staged`.

11. Выполните коммит с сообщением:

```
git commit -m "Changed output to say Hello Git."
```

12. Взгляните на результат в главной ветви — он должен выглядеть примерно так:

```
[main 6d2287d] Changed output to say Hello Git.  
1 file changed, 1 insertion(+), 1 deletion(-)
```

Если теперь снова выполнить команды `git diff` или `git diff --staged`, они ничего не покажут — потому что нет несохраненных изменений. А вот команда `git status` отобразит, что репозиторий обновлен:

```
On branch main  
nothing to commit, working tree clean
```

## Просмотр истории коммитов

Как только вы начнете выполнять фиксации, может понадобиться просмотреть историю коммитов. Для этого используется команда `git log`. У этой команды есть множество полезных параметров. В табл. 3.4 приведены наиболее распространенные из них.

**Таблица 3.4.** Популярные параметры команды `git log`

Переключатель	Описание
--patch или -p	Показывает подробные изменения (добавления, удаления, изменения строк). Можно ограничить количество, указав <code>-&lt;число&gt;</code> , например <code>-3</code> для трех последних патчей
--stat и --shortstat	Выводят сводку статистики по изменениям — сколько строк было добавлено, удалено и изменено
--name-status	Показывает список измененных файлов с указанием типа действия (A — добавлено, M — изменено, D — удалено)
--pretty	Форматирует вывод: можно выбрать <code>oneline</code> , <code>fuller</code> или задать собственный формат через <code>format</code> :
--graph	Добавляет к выводу текстовую диаграмму, отображающую структуру ветвлений и слияний

Рассмотрим несколько примеров.

1. В терминале выполните команду для отображения истории коммитов:

```
git log
```

2. В выводе отобразятся два коммита, которые мы сделали ранее. Они будут показаны в обратном хронологическом порядке — самый последний сверху. Каждый коммит содержит контрольную сумму SHA-1, имя и электронную почту автора, дату и время фиксации, сообщение коммита. Пример вывода показан ниже.

```
commit 6d2287d3ea630448f1bb112406b1834366652467 (HEAD -> main)
Author: Mark J Price <markjprice@gmail.com>
Date: Tue Jan 30 11:17:45 2024 +0000

    Changed output to say Hello Git and add solution and Git ignore files.

commit a4aa9557b71137a612cf7df430b3ad3c9faa3fde
Author: Mark J Price <markjprice@gmail.com>
Date: Tue Jan 30 09:25:55 2024 +0000

    Initial version
```



Контрольная сумма SHA-1 (Secure Hash Algorithm 1) — это криптографическая хеш-функция, создающая уникальное значение фиксированной длины (160 бит) для любых входных данных. Эта хеш-функция часто используется для проверки целостности данных. Хеш SHA-1 отображается как 40-значное шестнадцатеричное число. Подробнее с этим алгоритмом вы познакомитесь в главе 8.

3. В терминале выполните команду для отображения трех последних коммитов с подробными изменениями:

```
git log -p -3
```

4. Вы увидите результат с полным содержимым изменений для каждого из трех коммитов. Можно прокручивать страницы с помощью клавиш Page Up и Page Down, а выйти из просмотра — нажав клавишу Q.
5. Чтобы просмотреть краткую статистику по каждому коммиту, выполните команду:

```
git log --stat
```

6. В результате отобразятся оба коммита, сделанные ранее, с указанием количества добавленных и удаленных строк. Обратите внимание: изменение строки учитывается как удаление и вставка, то есть одно изменение будет отображено как + -, а два изменения — как 2 +-.

```
commit 6d2287d3ea630448f1bb112406b1834366652467 (HEAD -> main)
Author: Mark J Price <markjprice@gmail.com>
Date: Tue Jan 30 11:17:45 2024 +0000

    Changed output to say Hello Git and add solution and Git ignore files.
```

```
.gitignore          | 484 ++++++-----+-----+-----+-----+-----+-----+
Chapter03.sln      | 25 ++++
RepoDemo/Program.cs | 2 ++
3 files changed, 510 insertions(+), 1 deletion(-)

commit a4aa9557b71137a612cf7df430b3ad3c9faa3fde
Author: Mark J Price <markjprice@gmail.com>
Date:   Tue Jan 30 09:25:55 2024 +0000

Initial version

RepoDemo/Program.cs | 2 ++
RepoDemo/RepoDemo.csproj | 10 ++++++++
2 files changed, 12 insertions(+)
```

- Чтобы вывести еще более краткую статистику, выполните:

```
git log --pretty=oneline
```

- Вы увидите те же коммиты, но уже без списка файлов — только с общей сводкой по количеству изменений и сообщений коммитов.

```
6d2287d3ea630448f1bb112406b1834366652467 (HEAD -> main) Changed output to
say Hello Git and add solution and Git ignore files.
a4aa9557b71137a612cf7df430b3ad3c9faa3fde Initial version
```



**В копилку знаний:** для параметра --pretty доступны предустановленные стили — short, medium, full и fuller. Попробуйте каждый, чтобы увидеть разницу. Если нужен полный контроль над форматом, используйте команду format:<ваш формат>. Подробнее — на сайте [git-scm.com/docs/pretty-formats](https://git-scm.com/docs/pretty-formats).

- Выполните команду для вывода истории коммитов в пользовательском формате: отображается полный хеш (%H), имя автора синим цветом (%Cblue и %Creset), временная метка в формате ISO 8601 (%ai), а затем сообщение коммита (%s) с новой строки:

```
git log --pretty=format:"Author of %H was %Cblue%an%Creset, at
%ai%Message: %s%n"
```

- В результате отобразятся два коммита с форматированным выводом — как показано ниже и на рис. 3.17.

```
Author of 6d2287d3ea630448f1bb112406b1834366652467 was Mark J Price, at
2024-01-30 11:17:45 +0000
Message: Changed output to say Hello Git and add solution and Git ignore files.

Author of a4aa9557b71137a612cf7df430b3ad3c9faa3fde was Mark J Price, at
2024-01-30 09:25:55 +0000
Message: Initial version
```

```
C:\tools-skills-net8\Chapter03>git log --pretty=format:"Author of %H was %Cblue%an%Creset, at %ai%nTitle: %s%n"
"
Author of 6d2287d3ea630448f1bb112406b1834366652467 was Mark J Price, at 2024-01-30 11:17:45 +0000
Title: Changed output to say Hello Git and add solution and Git ignore files.

Author of a4aa9557b71137a612cf7df430b3ad3c9faa3fdc was Mark J Price, at 2024-01-30 09:25:55 +0000
Title: Initial version
```

**Рис. 3.17.** Форматированный вывод git log: имя автора выделено синим

- Выполните команду для отображения истории коммитов в компактном формате с коротким хешем (%h) и сообщением коммита (%s), дополнительно добавив визуальное отображение графа ветвлений:

```
git log --pretty=format:"%h %s" --graph
```

- В результате вы увидите два коммита, выполненные ранее, с короткими хешами и сообщениями — как показано в следующем примере:

```
* 6d2287d Changed output to say Hello Git and add solution and Git ignore
  files.
* a4aa955 Initial version
```

Сейчас вывод не выглядит особенно интересно, потому что в репозитории пока нет веток и слияний. Когда они появятся, график сформирует наглядное представление о структуре истории:

```
* 2d3acf9 Ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of https://github.com/dustin/grit.git
|\
| * 420eac9 Add method for getting the current branch
* | 30e367c Timeout code and tests
* | 5a09431 Add timeout protection to grit
* | e1193f8 Support for heads with slashes in them
|/
* d6016bc Require time for xmllschema
* 11d191e Merge branch 'defunkt' into local
```

## Фильтрация вывода журнала

Если не использовать фильтры, журнал коммитов быстро становится слишком громоздким. Git позволяет отфильтровать вывод по дате, автору, пути к файлу и даже содержимому сообщений коммитов.

Для ограничения истории по времени используйте параметры `--since` (или `--after`) и `--until` (или `--before`).

Чтобы отобразить коммиты за последние две недели, вы можете воспользоваться одной из следующих двух команд:

```
git log --since=2.weeks  
git log --after=2.weeks
```

Чтобы отобразить коммиты, сделанные начиная с 2024 года:

```
git log --since=2024-01-01  
git log --after=2024-01-01
```

Чтобы отобразить коммиты, сделанные до 2024 года:

```
git log --until=2024-01-01  
git log --before=2024-01-01
```

Фильтрация по автору, коммиттеру или файлам (путем к ним) выполняется следующим образом:

```
git log --author "Mark J Price"  
git log --committer "Mark J Price"  
git log -- RepoDemo/RepoDemo.csproj
```



**Дельный совет:** путь к файлу, по которому выполняется фильтрация, должен указываться в самом конце команды. При необходимости отделите его двойным дефисом `--`.

Если вы хотите найти коммиты, содержащие определенное слово (например, `ignore`), используйте параметр `--grep`:

```
git log -S "ignore"
```

## Управление удаленными репозиториями

*Удаленные репозитории* — это сетевые копии вашего проекта, размещенные, как правило, в интернете. Они позволяют обмениваться кодом с другими участниками команды и синхронизировать изменения. Работа с удаленными хранилищами включает в себя отправку и извлечение данных, а также управление их настройками.

Чтобы работать с удаленными репозиториями, потребуется учетная запись на GitHub (или другой платформе, поддерживающей Git). Если у вас ее еще нет, можно зарегистрироваться по адресу [github.com/join](https://github.com/join).



**В копилку знаний:** пошаговое руководство по регистрации доступно на сайте [docs.github.com/en/get-started/quickstart/creating-an-account-on-github](https://docs.github.com/en/get-started/quickstart/creating-an-account-on-github).

Чтобы просмотреть список настроенных удаленных репозиториев, выполните команду `git remote -v`.

Например, если вы клонировали репозиторий GitHub для этой книги, то в папке, куда он был клонирован, можно выполнить команду `git remote -v`. В результате вы увидите список удаленных репозиториев.

```
origin https://github.com/markjprice/tools-skills-net8.git (fetch)
origin https://github.com/markjprice/tools-skills-net8.git (push)
```

Обычно для одного и того же URL-адреса настроены как чтение (`fetch`), так и запись (`push`) в удаленное хранилище. Чтобы получить более подробную информацию об удаленном репозитории, выполните команду `show`:

```
git remote show origin
```

Вы увидите следующий результат:

```
* remote origin
  Fetch URL: https://github.com/markjprice/tools-skills-net8.git
  Push URL: https://github.com/markjprice/tools-skills-net8.git
  HEAD branch: main
  Remote branch:
    main tracked
  Local branch configured for 'git pull':
    main merges with remote main
  Local ref configured for 'git push':
    main pushes to main (up to date)
```

Посмотрим, как создать и привязать удаленный репозиторий GitHub к локальному проекту.

1. В каталоге `Chapter03` в терминале выполните команду, чтобы проверить, настроены ли какие-либо удаленные репозитории:

```
git remote -v
```

2. Если вы еще не добавляли удаленные репозитории, вывод будет пустым.
3. Откройте сайт [github.com](https://github.com) в браузере и войдите в свою учетную запись.
4. На главной странице нажмите кнопку `New repository` (Новый репозиторий) или перейдите по адресу [github.com/new](https://github.com/new).
5. Введите имя репозитория — `repodemo`. Остальные параметры можно оставить без изменений. Затем нажмите кнопку `Create repository` (Создать репозиторий).
6. Скопируйте предложенный URL-адрес репозитория (он будет уникальным для вашей учетной записи), затем добавьте его как удаленный репозиторий:

```
git remote add origin https://github.com/markjprice/repodemo.git
```

7. Чтобы убедиться, что привязка выполнена успешно, снова выведите список настроенных удаленных репозиториев:

```
git remote -v
```

8. Вы увидите вывод, в котором будет указан URL-адрес для чтения (fetch) и записи (push).

```
origin https://github.com/markjprice/repodemo.git (fetch)
origin https://github.com/markjprice/repodemo.git (push)
```

9. Разместите содержимое локальной ветки `main` вашего проекта в удаленном репозитории:

```
git push origin main
```



Чтобы отправить изменения в удаленный репозиторий, используйте команду `git push <удаленный_репозиторий> <имя_ветки>`. Например: `git push origin main`. Чтобы получить обновления из удаленного репозитория без автоматического слияния, выполните команду `git fetch <удаленный_репозиторий>`, например: `git fetch origin`. Для извлечения изменений с одновременным слиянием с локальной веткой используйте команду `git pull`.

10. При отправке изменений Git может запросить проверку подлинности. Выберите способ входа через браузер или мобильное приложение. Если для вашей учетной записи включена двухфакторная проверка, потребуется ввести код из приложения GitHub на мобильном устройстве.
11. После успешной проверки подлинности файлы проекта будут загружены в удаленный репозиторий.

```
info: please complete authentication in your browser...
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 8 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (11/11), 4.76 KiB | 4.76 MiB/s, done.
Total 11 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/markjprice/repodemo.git
 * [new branch]      main -> main
```

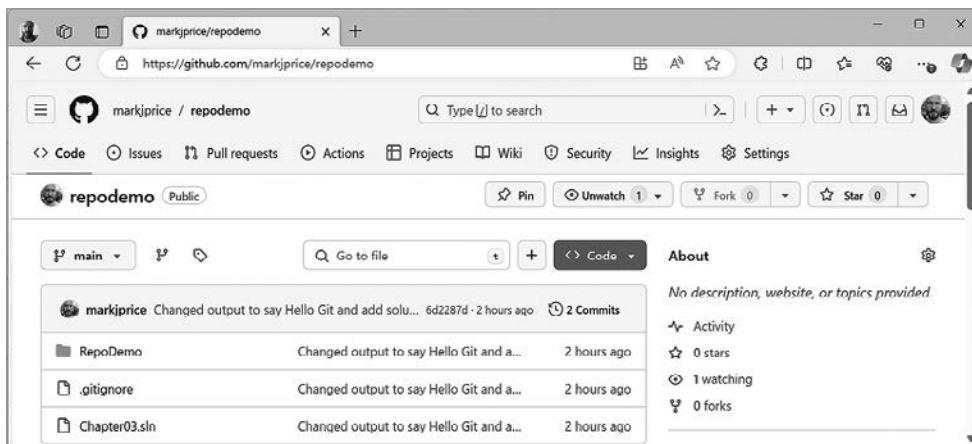
12. Чтобы просмотреть подробную информацию о настройке удаленного репозитория, выполните команду:

```
git remote show origin
```

13. В выводе будут указаны расположение репозитория, используемые протоколы, а также сведения об отслеживаемых ветках.

```
* remote origin
  Fetch URL: https://github.com/markjprice/repodemo.git
  Push URL: https://github.com/markjprice/repodemo.git
  HEAD branch: main
  Remote branch:
    main tracked
  Local ref configured for 'git push':
    main pushes to main (up to date)
```

14. Откройте в браузере URL-адрес вида [github.com/markjprice/repodemo](https://github.com/markjprice/repodemo) (в вашем случае имя учетной записи и репозитория будет другим). Вы увидите, что локальные файлы были успешно синхронизированы с удаленным репозиторием (рис. 3.18).



**Рис. 3.18.** Репозиторий RepoDemo, размещенный на сайте GitHub

Далее мы перейдем к работе с ветвлением и слиянием.

## Ветвление и слияние

*Ветвление* — ключевая особенность Git, делающая процесс разработки гибким, безопасным и особенно удобным в командной работе. Она позволяет эффективно управлять большими кодовыми базами и развивать проект в разных направлениях одновременно.

*Ветка* — это самостоятельная линия развития. Можно представить ее как отдельную рабочую область со своей историей изменений. Все коммиты, сделанные в ветке, не затрагивают остальную часть проекта, пока вы не решите объединить их обратно. Получается своеобразный «ответвленный путь» (*форк*) от основной истории.

С помощью веток удобно экспериментировать, внедрять новые функции или исправлять ошибки — не затрагивая основную версию. Ветки работают как песочница, где можно спокойно пробовать идеи, не рискуя повредить важный код.

Обычно ветки используются в следующих сценариях.

- **Ветки для новых функций.** Каждая новая функция разрабатывается в отдельной ветке. Когда работа завершена, ветка сливается с основной (обычно — с `main`).
- **Ветки релизов.** Как только проект готов к выпуску, создается отдельная ветка для подготовки релиза. Такой подход позволяет устранять баги параллельно с продолжением разработки в основной ветке.
- **Ветки для срочных исправлений.** Если в стабильной версии обнаружен критический дефект, создается ветка от этой версии, выполняется исправление, и изменения сливаются обратно как в стабильную, так и в основную ветку.

Git упрощает и ускоряет работу с ветками — создание, переключение и слияние занимают всего доли секунды. Такой подход поощряет частое использование ветвления, вплоть до нескольких раз в день, и значительно отличается от старых систем контроля версий, где подобные операции были громоздкими. Умение уверенно работать с ветками и слиянием действительно меняет взгляд на весь процесс разработки — становится проще экспериментировать, разделять задачи и сотрудничать в команде.

## Пример ветвления и слияния

В этом упражнении вы пошагово пройдете типичный сценарий работы с ветками и слиянием. План такой.

1. Выполните часть работы в основной ветке `main` проекта `RepoDemo` — добавьте базовую функцию калькулятора для сложения двух чисел.
2. Создайте новую ветку для разработки функции настройки культуры консоли.
3. Напишите код этой функции в новой ветке.
4. Переключитесь обратно на ветку `main`, чтобы внести исправление в метод `Add`.
5. Создайте отдельную ветку для исправления ошибки.
6. Объедините ветку с исправлением обратно в `main`.
7. Вернитесь к работе над функцией настройки культуры, продолжив ее разработку.

Приступим!

1. В проекте `RepoDemo` создайте файл с именем `Calculator.cs`.
2. В этом файле определите класс `Calculator` с методом `Add`. Допустите в методе намеренную ошибку — мы вернемся к ней позже и исправим:

```
public static class Calculator
{
    public static int Add(int a, int b)
    {
        return a * b;
    }
}
```

3. В файле `Program.cs` воспользуйтесь классом `Calculator`, чтобы сложить числа 2 и 3, и выведите результат на консоль:

```
Console.WriteLine(Calculator.Add(2, 3));
```

4. В каталоге `Chapter03` в терминале проверьте текущий статус репозитория:

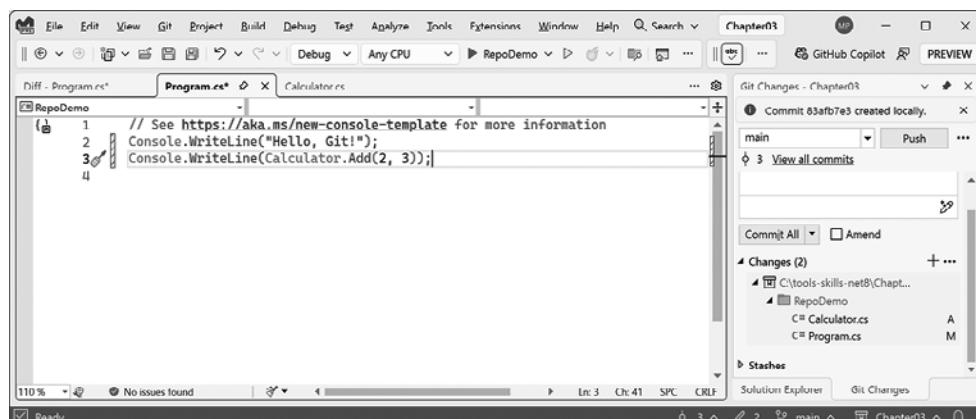
```
git status
```

5. В выводе вы увидите, что нужно добавить два изменения в область подготовки: один файл — новый, другой — изменен.

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified: RepoDemo/Program.cs

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    RepoDemo/Calculator.cs
```

6. В программе Visual Studio, в окне `Git Changes` (Изменения Git) один файл отмечен буквой A (добавлен), другой — буквой M (изменен), как показано на рис. 3.19.



**Рис. 3.19.** Добавленные и измененные файлы в окне `Git Changes` в Visual Studio

7. В каталоге `Chapter03` с помощью терминала добавьте в индекс неотслеживаемый файл:

```
git add RepoDemo/Calculator.cs
```



Если вы работаете в Visual Studio, нажмите кнопку +, чтобы переместить файл `Calculator.cs` в раздел `Staged Changes` (Изменения в области подготовки). Аналогично переместите измененный файл `Program.cs`, так как Visual Studio не поддерживает одновременное добавление и фиксацию изменений, в отличие от терминала, который мы используем в следующем шаге.

8. Добавьте измененный файл в область подготовки, затем выполните коммит обоих файлов с сообщением:

```
git commit -a -m "Add calculator functionality and call the Add method."
```



В Visual Studio введите сообщение о коммите и нажмите кнопку `Commit Staged` (Коммит подготовленных файлов).

9. В терминале обратите внимание, что коммит был выполнен в ветке `main`, и вы увидите подтверждение в выводе:

```
[main dfc1ef8] Add calculator functionality and call the Add method.
 2 files changed, 8 insertions(+), 1 deletion(-)
  create mode 100644 RepoDemo/Calculator.cs
```

10. Убедитесь, что коммитов больше не требуется:

```
git status
```

11. Создайте ветку `configure-console` для работы над функцией настройки консоли:

```
git branch configure-console
```

12. Переключитесь на эту ветку:

```
git switch configure-console
```



Создание и переключение на новую ветку можно объединить в одно действие с помощью команды `git switch -c <имя_ветки>`. Ранее для переключения веток использовалась команда `git checkout <имя_ветки>` или `git checkout -b <имя_ветки>`.

13. Если вы работаете в Visual Studio, в строке состояния щелкните кнопкой мыши на текущей ветке  main ^, нажмите кнопку New Branch (Новая ветка), введите имя `configure-console`, основанное на ветке `main`, установите флашок Checkout branch (Переключиться на ветку), затем нажмите кнопку Create (Создать).
14. В файле `RepoDemo.csproj` проекта статически и глобально импортируйте класс `Console`.

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

15. В папке проекта `RepoDemo` создайте файл с именем `Program.Helpers.cs`.
16. В этом файле определите частичный класс `Program` с методом `ConfigureConsole`.



Если вы настроили фрагмент кода, просто введите `ccc` и дважды нажмите клавишу `Tab`.

```
using System.Globalization;

partial class Program
{
    static void ConfigureConsole(string culture = "en-US",
        bool useComputerCulture = false)
    {
        // Чтобы консоль могла отображать символы Юникода, такие как знак евро
        OutputEncoding = System.Text.Encoding.UTF8;

        if (!useComputerCulture)
        {
            CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo(culture);
        }

        WriteLine($"CurrentCulture: {CultureInfo.CurrentCulture.DisplayName}");
    }
}
```

17. В верхней части файла `Program.cs` вызовите метод `ConfigureConsole`, чтобы настроить консоль.

```
ConfigureConsole();
```

18. В каталоге `Chapter03` с помощью команды в терминале добавьте неотслеживаемый файл в индекс.

```
git add RepoDemo/Program.Helpers.cs
```



Если вы работаете в Visual Studio, нажмите кнопку + Stage All (Добавить в область подготовки все) в верхней части окна Changes (Изменения), чтобы переместить все новые и измененные файлы в раздел Staged Changes (Изменения в области подготовки).

- Добавьте измененный файл в индекс, затем выполните коммит с сообщением:

```
git commit -a -m "Add new feature to configure the console culture."
```



В Visual Studio введите сообщение и нажмите кнопку Commit Staged (Коммит подготовленных файлов). Можно также попробовать использовать сгенерированное ИИ сообщение.

- Убедитесь, что коммит действительно был сделан в ветке `configure-console` и что изменены три файла.

```
[configure-console f90cddb] Add new feature to configure the console culture.
3 files changed, 23 insertions(+)
create mode 100644 RepoDemo/Program.Helpers.cs
```

- Выполните команду `git status`, чтобы убедиться, что больше ничего фиксировать не нужно.
- Переключитесь обратно на ветку `main`.

```
git switch main
```



В программе Visual Studio щелкните на значке ветки в строке состояния и выберите ветку `main`.

- Обратите внимание, что файлы в папке `RepoDemo` вернулись к своему состоянию до переключения на ветку `configure-console`.



В Visual Studio выполните команду меню `Git > Manage Branches` (`Git > Управление ветками`), а затем выберите ветку `configure-console`.

- Создайте ветку с именем `calc-hotfix` и переключитесь на нее.

```
git switch -c calc-hotfix
```

- В файле `Calculator.cs` исправьте ошибку, заменив оператор умножения на оператор сложения.

```
return a + b;
```

26. Добавьте измененный файл в область подготовки и зафиксируйте изменения с сообщением о коммите:

```
git commit -a -m "Fix the Add method bug."
```

27. Убедитесь, что коммит был сделан в ветке `calc-hotfix`. В выводе будет указано, какие файлы были изменены.

```
[calc-hotfix 9a9776a] Fix the Add method bug.
 1 file changed, 1 insertion(+), 1 deletion(-)
```



**Дельный совет:** прежде чем двигаться дальше, запустите тесты и убедитесь, что исправление действительно работает.

28. Переключитесь обратно на основную ветку:

```
git switch main
```

29. Выполните слияние ветки исправлений с веткой `main`:

```
git merge calc-hotfix
```

30. Взгляните на результат, как показано в следующем выводе:

```
Updating dfc1ef8..9a9776a
Fast-forward
 RepoDemo/Calculator.cs | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Если вы работаете в Visual Studio, щелкните правой кнопкой мыши на ветке, которую хотите слить, и выберите в контекстном меню команду `Merge '<имя ветки>' into '<имя ветки>'` (Слийте "<имя ветки>" в "<имя ветки>") (рис. 3.20).

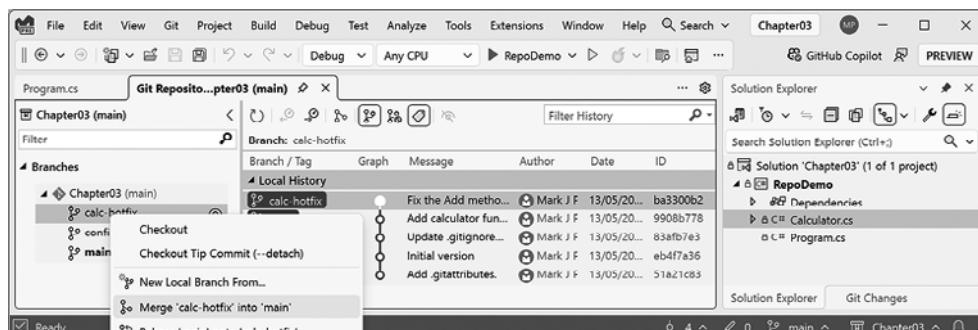


Рис. 3.20. Слияние ветки исправлений с основной веткой в Visual Studio

## Удаление и просмотр веток

После того как исправление было объединено с веткой `main`, вы можете удалить ветку с исправлением с помощью следующей команды:

```
git branch -d calc-hotfix
```

Но пока мы оставим эту ветку.

Теперь можно переключиться обратно на ветку `configure-console` и продолжить работу над новой функцией:

```
git switch configure-console
```

Чтобы вывести список всех локальных веток, выполните следующую команду:

```
git branch
```

В выводе текущая активная ветка отмечена звездочкой, как показано в следующем примере:

```
calc-hotfix
configure-console
* main
```

## Краткое описание распространенных команд Git

Наиболее часто используемые команды Git, а также способы их отмены приведены в табл. 3.5.

**Таблица 3.5.** Распространенные команды Git

Команда Git	Назначение	Отмена
<code>config</code>	Просмотр (--list) или установка параметров конфигурации	Недоступно
<code>init</code>	Инициализирует текущий каталог как репозиторий Git. Создается подкаталог <code>.git</code>	Удалите каталог <code>.git</code>
<code>add &lt;шаблон_файла&gt;</code>	Добавляет файлы, соответствующие шаблону, в индекс (область подготовки). <code>-r</code> — для рекурсии	<code>rm --cached &lt;шаблон_файла&gt;</code> или <code>rm --staged</code>
<code>commit -m &lt;сообщение&gt;</code>	Фиксирует подготовленные файлы с указанным сообщением. Устанавливаются автор и временная метка	<code>commit --amend, revert</code> или <code>reset</code> с нужным флагом

<b>Команда Git</b>	<b>Назначение</b>	<b>Отмена</b>
<code>status</code>	Показывает текущее состояние файлов в репозитории. <code>-s</code> — сокращенный формат	Недоступно
<code>log</code>	Показывает историю коммитов. <code>-p</code> — подробности об изменениях	Недоступно
<code>branch &lt;имя_ветки&gt;</code>	Создает ветку или удаляет существующую	Недоступно
<code>switch &lt;имя_ветки&gt;</code>	Переключает на указанную ветку. С <code>-c</code> — создает новую. Старый вариант — <code>checkout</code> (создание с <code>-b</code> )	Недоступно
<code>merge &lt;имя_ветки&gt;</code>	Объединяет указанную ветку с текущей	Недоступно

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и тщательнее изучив ключевые темы главы.

### Упражнение 3.1. Дополнительные онлайн-материалы

Подробную информацию о Git и работе с этой системой в программе Visual Studio можно получить на сайте [learn.microsoft.com/ru-ru/visualstudio/version-control/?view=vs-2022](https://learn.microsoft.com/ru-ru/visualstudio/version-control/?view=vs-2022).

Бесплатные курсы по работе с GitHub доступны по адресу [learn.microsoft.com/ru-ru/training/github/](https://learn.microsoft.com/ru-ru/training/github/).

Если вы хотите подтвердить свои навыки, можно пройти сертификацию GitHub.

- **Основы GitHub.** Подтверждает базовые знания Git и работу с репозиториями GitHub, включая принципы совместной разработки и понимание экосистемы платформы.
- **GitHub Copilot.** Оценивает умение использовать инструмент автодополнения кода на базе ИИ для ускорения и оптимизации повседневной разработки на разных языках программирования.
- **GitHub Actions.** Проверяет навыки в автоматизации процессов, построении и оптимизации CI/CD-конвейеров с помощью настраиваемых рабочих процессов.
- **GitHub Advanced Security.** Подтверждает компетенции в обеспечении безопасности, включая выявление уязвимостей, защиту рабочих процессов и соблюдение стандартов надежности кода.

- **GitHub Administration.** Демонстрирует опыт в управлении репозиториями, настройке рабочих процессов и организации командного взаимодействия в рамках проектов на GitHub.

Подробнее на сайте [resources.github.com/learn/certifications/](https://resources.github.com/learn/certifications/).

Полезным ресурсом для начинающих пользователей Git может стать сайт [think-like-a-git.net](http://think-like-a-git.net). На нем сказано: «Вы должны уметь создавать репозиторий, добавлять и фиксировать файлы — и примерно понимать, зачем может понадобиться ветвление». Однако сайт будет по-настоящему полезен только в том случае, если вы уже освоили основы Git. Он ориентирован скорее на тех, кто уже использует систему и хочет глубже разобраться в ее устройстве. Прежде чем обращаться к этому ресурсу, убедитесь, что вы поняли ключевые идеи, изложенные в этой главе.

## Упражнение 3.2. Практические задания

ВидеоДуки по Git для пользователей со средним и продвинутым уровнем можно найти по адресу: [devblogs.microsoft.com/visualstudio/intermediate-and-advanced-git/](https://devblogs.microsoft.com/visualstudio/intermediate-and-advanced-git/).

Некоторые полезные онлайн-книги по Git.

- **Pro Git** — авторы Скотт Чакон и Бен Страб. Книга издана под лицензией Creative Commons Attribution-NonCommercial-ShareAlike 3.0. Книга доступна бесплатно на множестве языков, включая русский. Прочитать ее можно по адресу [git-scm.com/book/en/v2](http://git-scm.com/book/en/v2).
- **Conversational Git** — автор Алан Хон. Он написал эту книгу, чтобы представить Git глазами человека, который использует систему ежедневно, но при этом помнит, каким сложным может показаться ее освоение в начале. Книга доступна по адресу: [alanhohn.com/extras/conversational-git/](http://alanhohn.com/extras/conversational-git/).

## Упражнение 3.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Какие пять ключевых характеристик должна обеспечивать система управления исходным кодом?
2. Почему инструмент Git может быть сложным для изучения, особенно для начинающих?
3. В каких четырех состояниях может находиться файл в каталоге репозитория Git?
4. В каких случаях применяется команда `git diff`?
5. Насколько часто используется ветвление в Git?

## Упражнение 3.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-3---source-code-management-using-git](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-3---source-code-management-using-git).

## Резюме

В этой главе вы узнали, как:

- установить и настроить Git на локальном компьютере;
- создавать локальный репозиторий или клонировать удаленный;
- отслеживать изменения в файлах и управлять их состояниями;
- отменять коммиты и очищать рабочее пространство;
- использовать механизм сохранения (stash);
- просматривать историю и различия между коммитами;
- работать с ветвлением и слиянием в Git;
- продолжить изучение Git с помощью видеоматериалов и книг.

В следующей главе мы перейдем к отладке и анализу использования памяти в .NET-приложениях.

# 4

# Отладка и устранение проблем с памятью

Эта глава посвящена инструментам и приемам отладки, а также способам выявления и устранения проблем с памятью. Для разработчиков .NET анализ памяти играет ключевую роль: он помогает находить утечки, понимать распределение ресурсов и повышать производительность приложений.

## Стратегии отладки

Отладка может оказаться самой сложной частью процесса разработки. Профессиональные разработчики .NET нередко тратят на нее больше времени, чем на само написание кода, поэтому, планируя обучение, стоит учитывать, сколько времени вы уделите навыкам программирования и сколько — отладке.

Цель отладки — выявление и устранение ошибок. Эти ошибки могут быть самыми разными: от небольших визуальных недочетов, слегка влияющих на внешний вид приложения, до серьезных сбоев, которые могут привести к потере данных или уязвимостям в системе безопасности. Благодаря отладке можно своевременно обнаружить эти проблемы и устраниить их.

Однако устранение ошибок — не единственная причина заниматься отладкой. Особенно для начинающих разработчиков отладка — это способ глубже понять, как работает приложение, как оно «думает». Отладка — мощный инструмент для обучения.

Постоянная практика отладки помогает не только находить баги, но и реорганизовывать код, улучшать производительность и поддерживать чистоту и масштабируемость проекта. Понимая, почему код не сработал так, как ожидалось, вы сможете избежать аналогичных ошибок в будущем.

Мы начнем эту главу с разбора стратегий отладки проекта .NET — независимо от того, какие инструменты вы используете.

## Обзор стратегий отладки

Рассмотрим несколько стратегий, которые помогут эффективно отлаживать проекты .NET.

- **Разберитесь в сути проблемы.** Прежде чем погружаться в код, важно точно понять, в чем заключается ошибка. Воспроизведите проблему, внимательно изучите сообщения об ошибках и определите, в каком контексте она возникает. Часто причина скрыта глубже в коде, чем место, где ошибка себя проявляет. Поспешные правки могут только все усугубить. Помните главный принцип: «Не навреди!»
- **Используйте весь потенциал отладчика.** Точки останова помогают не только приостанавливать выполнение программы, но и исследовать ее состояние в нужных местах. Условные точки останова особенно удобны, если ошибка проявляется при определенных условиях. Не забывайте и про панель **Immediate** (Немедленное выполнение) для проверки выражений «на лету», а также про панель **Call Stack** (Стек вызовов) для анализа последовательности вызовов, приведших к текущему моменту. Эти инструменты мы подробно рассмотрим в следующих разделах главы.
- **Пишите и запускайте модульные тесты.** Модульное тестирование — отличный способ локализовать ошибки. Разбивая приложение на небольшие, изолированные фрагменты и проверяя каждый из них в отдельности, вы можете точно определить, где происходит сбой. Также полезно создавать тесты, воспроизводящие выявленные ошибки, — это помогает убедиться, что проблема действительно устранена и не вернется снова. Один из самых популярных фреймворков для модульного тестирования в .NET — xUnit. Вы подробнее познакомитесь с ним в главе 11.
- **Используйте статический анализ кода.** Инструменты статического анализа, такие как компиляторные анализаторы Roslyn, позволяют находить потенциальные ошибки еще до выполнения программы. Они помогают улучшить стиль и качество кода. В Visual Studio для этого можно использовать встроенные анализаторы и расширение ReSharper, а в Rider — встроенные подсказки.
- **Применяйте средства профилирования памяти и производительности.** Профилировщики позволяют выявлять утечки памяти, неэффективные SQL-запросы, избыточное потребление ресурсов и другие узкие места. Вы узнаете, как использовать эти инструменты, в следующих разделах этой главы.
- **Ведение журнала и наблюдаемость.** Надежное журналирование помогает понять, что происходило в приложении до того, как возникла ошибка. Используйте инструменты вроде Serilog, чтобы фиксировать важные события, параметры и состояния. В главе 5 вы познакомитесь с OpenTelemetry, современным расширяемым фреймворком для журналирования и мониторинга.

Отладка — это не только наука, требующая системного подхода, но и искусство, в котором важны терпение и интуиция. Методичность и вдумчивость часто оказываются важнее спешки. Все описанные выше стратегии помогут вам сформировать надежный инструментарий для отладки проектов на .NET.

А сейчас давайте подробнее остановимся на ключевом элементе любой отладки — понимании сути проблемы. Одна из самых распространенных ошибок среди разработчиков — стремление как можно быстрее применить решение, не вникая в истинную причину сбоя. Такой подход не только не устраивает исходную проблему, но и может привести к появлению новых.

## Анализ проблемы

Доскональное понимание проблемы — это важнейший первый шаг в отладке. И нередко все оказывается гораздо сложнее, чем кажется на первый взгляд. Ниже — более подробный разбор этой стратегии.

- Постарайтесь стабильно воспроизвести проблему.** Начните с повторения ошибки в контролируемой среде — например, в среде разработки или на стенде, имитирующем рабочую систему. Так вы сможете отлаживать проблему, не влияя на рабочее окружение. Очень важно добиться повторяемости: если ошибка возникает нестабильно, отладка становится гораздо сложнее. Найдите надежный способ воспроизведения ошибки, состоящий, скажем, из семи шагов, и постарайтесь максимально упростить сценарий. Например, если сбой стабильно воспроизводится после семи действий, попробуйте исключить часть шагов и выявить, какие из них действительно вызывают проблему.
- Сберите всю возможную информацию.** Записи в журнале, сообщения об ошибках, трассировка стека — все это может дать ключ к разгадке. Если сообщение поступило от пользователя, важно получить максимум сведений — желательно автоматизированно. Не полагайтесь только на его описание: пользователь может неверно интерпретировать, что именно пошло не так. Постарайтесь выяснить, что он делал перед ошибкой, какие действия привели к сбою и можно ли быстро зафиксировать ситуацию, например, с помощью снимка экрана.



Можно использовать службы вроде Azure Application Insights или Sentry, чтобы автоматически собирать информацию о действиях пользователя в момент возникновения исключения. Однако в этой книге такие платные облачные решения не рассматриваются.

- Документируйте свои наблюдения.** Записывайте все, что удалось выяснить, даже если на первый взгляд это не приближает к решению. Эти сведения могут оказаться полезными позже — особенно если проблема повторится или всплынет в другом контексте.
- Анализируйте контекст.** Обратите внимание на недавние изменения в коде. Не появилась ли ошибка сразу после коммита или развертывания? В таких

случаях может помочь команда `git bisect`, позволяющая поэтапно вычислить коммит, в котором ошибка впервые проявилась. Также учитывайте различия между средами — например, между разработкой, тестированием и продуктивной системой. Иногда сбои возникают только при определенной конфигурации или на реальных данных. Классический пример — строка подключения в профайлсе, у которой может быть сокращен тайм-аут, из-за чего те же запросы начинают завершаться с ошибкой.



Подробнее о команде `git bisect` можно узнать на сайте [git-scm.com/docs/git-bisect](http://git-scm.com/docs/git-bisect).

5. **Обратитесь к истории кода.** Используйте систему контроля версий, такую как Git, чтобы понять, кто и когда вносил изменения в проблемный участок. Команда `git blame` (или аналогичная функция в редакторе) позволяет отследить, кто и когда внес конкретные строки в код. Этот инструмент поможет понять, зачем код был написан именно так, и выяснить, не связаны ли с ним другие проблемы. Загляните также в комментарии и документацию — они могут дать полезные подсказки о задумке и поведении кода.
6. **Зафиксируйте разницу между ожидаемым и фактическим поведением.** Четко опишите, как должна работать система и как она ведет себя на практике. Такой подход помогает точнее определить проблему и убедиться, что вы решаете именно ту задачу, которую нужно.
7. **Формулируйте гипотезы и проверяйте их.** На основе собранной информации выдвигайте предположения о возможных причинах ошибки. Затем подтверждайте или опровергайте их с помощью тестов и экспериментов. Когда вы четко понимаете, каким должен быть результат, удобно начать с написания модульного теста, который воспроизводит сбой. Так вы сможете отлаживать код, не запуская все приложение. Исправляйте ошибку и повторяйте тест до тех пор, пока он не начнет проходить успешно.
8. **Используйте отладчик по максимуму.** Изучите отладочные возможности вашей среды разработки. Умение эффективно ставить и снимать точки останова, переходить по шагам, анализировать переменные и стек вызовов значительно ускоряет поиск и устранение ошибок. Эти навыки — основа качественной и продуктивной отладки.
9. **Сотрудничайте и обсуждайте.** Поговорите о проблеме с коллегами. Часто само объяснение задачи вслух помогает глубже осмыслить ситуацию и найти решение. Если вы работаете в одиночку, попробуйте прием «резиновой уточки» — проговорите проблему неодушевленному предмету. А еще лучше — обсудите ее с искусственным интеллектом, который понимает контекст вашего проекта, например с GitHub Copilot Chat.
10. **Разбейте задачу на части.** Если проблема слишком сложная, попробуйте разделить ее на меньшие компоненты. Протестируйте отдельные участки кода по частям — так будет проще локализовать сбой и упростить диагностику.

11. **Обязательно проверьте исправление.** Этот шаг часто игнорируются. Найдя причину ошибки и исправив ее, вы можете почувствовать облегчение и уверенность — и забыть повторно воспроизвести исходную ситуацию. Но важно убедиться, что проблема действительно устранена. Также проверьте, не вызвало ли исправление побочных эффектов — особенно если затронуты общие участки кода. Если у вас есть автоматизированные тесты, обязательно запустите их, чтобы убедиться в отсутствии регрессий.



**Дельный совет:** программное обеспечение — это не магия. За каждой ошибкой всегда стоит логическая причина. Часто решение оказывается на удивление простым, и вы ругаете себя за то, что не увидели очевидного.

Цель на этом этапе — не просто как можно быстрее устранить проблему, а действительно разобраться, почему она возникла. Глубокое понимание причин приводит к более надежным и устойчивым решениям, а заодно помогает лучше разобраться в системе в целом. В результате будущая отладка станет быстрее, точнее и понятнее.

## Как начать отладку

Начинайте с самого начала. Даже если проблема проявляется, скажем, при оформлении заказа в интернет-магазине, сначала убедитесь, что базовая инициализация прошла успешно: подключение к базе данных, настройки учетной записи пользователя и другие стартовые операции.

Представьте, что у вас есть несколько догадок: может быть, дело в строке подключения, может — в модели сущности, а может — в клиентском коде. Возникает соблазн сразу внести изменения во все сразу, чтобы «сработало». Но если проблема исчезнет, вы не узнаете, что именно ее решило. Меняйте только один элемент за раз, даже если на это уйдет больше времени. Одно изменение — один тест. Только так можно точно определить причину и убедиться, что исправление безопасно и обоснованно.

А теперь представьте худший вариант: вы внесли несколько изменений, но ошибка все еще не исчезла. Возможно, одно из изменений и было правильным, но остальные — наоборот, мешают. В результате вы теряете ориентир, и диагностика становится еще сложнее.

## Когда стоит остановиться

Иногда лучше *не* исправлять ошибку. Особенно это касается мелких сбоев в пользовательском интерфейсе. Если он работает и выполняет свою функцию, но при этом что-то визуально «подглючивает», возможно, стоит переключиться на более

важную задачу. То же касается автоматизированных функций, которые могут быть выполнены вручную. Если надежность автоматики вызывает сомнения, иногда проще отключить ее вовсе, чем тратить ресурсы на отладку ненадежной логики. Особенно если пользователи способны справиться без нее. Здравый смысл и приоритеты проекта здесь важнее идеальной чистоты кода.

## Интерактивная отладка в Visual Studio

Одна из ключевых стратегий при отладке — это умение эффективно использовать интерактивные средства вашей среды разработки. В этом разделе вы научитесь находить и устранять ошибки в интерактивном режиме прямо во время разработки.



Команды для запуска отладки находятся в меню Debug (Отладка) в Visual Studio или в меню Run (Запуск) в Code и Rider.

### Создание кода для изучения объектов во время выполнения

Перейдем к практике: создадим консольное приложение, которое ссылается на библиотеку классов, а затем используем встроенные средства отладки, чтобы наблюдать за объектами в процессе выполнения программы.

1. В редакторе кода создайте проект Class Library (Библиотека классов) / `classlib`, присвойте ему имя `DebugLibrary` и добавьте в решение `Chapter04`.
2. В файле `DebugLibrary.csproj` настройте проект так, чтобы для всех файлов кода выполнялся статический импорт пространств имен `System.Console`, `System.Convert` и `System.Text.Encoding`, а также чтобы все предупреждения компилятора обрабатывались как ошибки.

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

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

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

</Project>
```

3. Переименуйте файл `Class1.cs` в `WebConfig.cs`.
4. В этом файле определите класс с набором свойств, которые могли бы использоваться, например, в веб-приложении — эти свойства пригодятся для демонстрации работы отладчика.

```
namespace Packt.Shared;

public class WebConfig
{
    public string? DbConnectionString { get; set; }

    public string? Base64Encoded { get; set; }

    public string? JsonWebToken { get; set; }

    public WebConfig()
    {
        DbConnectionString = "Server="
            + "(localdb)\\mssqllocaldb;"
            + "Database=Northwind;"
            + "Trusted_Connection=true";

        byte[] data = UTF8.GetBytes("Debugging is cool!");
        Base64Encoded = ToBase64String(data);

        // Установка строки JWT из примера на сайте jwt.io
        JsonWebToken =
            "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9." +
            "eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I" +
            "kpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ." +
            "SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c";
    }

    public void OutputAll()
    {
        WriteLine($"DbConnectionString: {DbConnectionString}");
        WriteLine($"Base64Encoded: {Base64Encoded}");
        WriteLine($"JsonWebToken: {JsonWebToken}");
    }
}
```

5. Создайте файл класса `Product.cs`.
6. В этом файле определите класс `Product`, добавив нужные свойства и методы для демонстрации отладки.

```
namespace Northwind.EntityModels;

public class Product
{
    public int ProductId { get; set; }
    public string ProductName { get; set; } = null!;
    public int? SupplierId { get; set; }
    public int? CategoryId { get; set; }
    public string? QuantityPerUnit { get; set; }
    public decimal? UnitPrice { get; set; }
```

```

    public short? UnitsInStock { get; set; }
    public short? UnitsOnOrder { get; set; }
    public short? ReorderLevel { get; set; }
    public bool Discontinued { get; set; }
}

```

7. В редакторе кода создайте проект **Console App** (Консольное приложение) / **console**, присвойте ему имя **DebugApp** и добавьте его в решение **Chapter04**.
8. В файле **DebugApp.csproj** настройте статический импорт класса **System.Console** для всех файлов кода, настройте обработку предупреждений как ошибок и добавьте ссылку на проект **DebugLibrary**.

```

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

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

  <ItemGroup>
    <ProjectReference Include=".\\DebugLibrary\\DebugLibrary.csproj" />
  </ItemGroup>

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

</Project>

```

9. В файле **Program.cs** удалите все содержимое и добавьте код для создания экземпляра класса **WebConfig** и вывода его свойств, а также для создания экземпляра класса **Product**. Благодаря этому вы сможете наблюдать за состоянием объектов во время отладки.

```

using Packt.Shared; // Создание и использование объекта WebConfig
using Northwind.EntityModels; // Создание и использование экземпляра класса
Product

WebConfig config = new();
config.OutputAll();

Product product = new()
{
    ProductId = 1,
    CategoryId = 1,
    ProductName = "Chai",
    UnitPrice = 1,
    UnitsInStock = 1,
    Discontinued = false
};

WriteLine("Press enter to exit.");

```

## Установка точки останова и начало отладки

Точка останова позволяет приостановить выполнение программы в нужной строке, чтобы изучить состояние объектов, переменных и стека вызовов. Это один из самых мощных инструментов интерактивной отладки. Ниже перечислены ситуации, где особенно полезно устанавливать точки останова при работе над проектами .NET.

- **Области с недавними изменениями.** Если ошибка возникла после последних изменений, установите точки останова именно в этих местах. Такой прием помогает быстрее определить, связано ли поведение программы с внесенными правками и как именно они повлияли на выполнение.
- **Известные проблемные участки.** Если вы знаете, что определенные части кода сложны или часто вызывают ошибки, имеет смысл ставить точки останова именно там — особенно если связанная функциональность начинает вести себя непредсказуемо.
- **Начало метода, где возникает проблема.** Как только вы определили, в каком методе проявляется ошибка, установите точку останова в его начале. Так вы сможете проследить, как обрабатываются входные параметры и что происходит внутри метода шаг за шагом.
- **До и после критически важных операций.** Если код взаимодействует с внешними сервисами, API или базами данных, устанавливайте точки останова перед отправкой запроса и сразу после получения ответа. Так вы убедитесь, что данные передаются и обрабатываются корректно.
- **Обработка пользовательского ввода.** Если ошибка может быть связана с тем, как программа воспринимает и обрабатывает ввод пользователя (в пользовательском интерфейсе, HTTP-запросах или формах), установите точки останова в местах, где происходит прием и первичная обработка данных. Так вы убедитесь, что ввод поступает в ожидаемом виде и корректно используется в дальнейшем.
- **Блоки обработки исключений.** Точки останова в блоках `catch` позволяют изучить состояние приложения в момент перехвата ошибки. Так вы убедитесь, что исключение обрабатывается корректно, и сможете проанализировать, что именно произошло не так и почему ошибка возникла в конкретной ситуации.
- **Модульные тесты.** Если тест не проходит из-за ошибки, точка останова внутри самого теста помогает понять, на каком этапе что-то идет не так. Так вы убедитесь, как именно выполняется тест, и сможете отследить сложные или нестабильные сценарии. К этой теме мы еще вернемся в одном из следующих разделов.
- **Точки входа и выхода из функций.** В больших и сложных функциях имеет смысл ставить точки останова как в начале, так и в конце. Такой прием помогает проследить, какие значения передаются на вход и что возвращается в итоге.

- **Места изменения состояния.** Если вы подозреваете, что ошибка связана с тем, что значение переменной меняется неожиданным образом, поставьте точки останова везде, где эта переменная может быть изменена. Так вы отследите, в каком именно месте происходит нежелательное изменение.

Установим точку останова и запустим отладку в Visual Studio.

1. Щелкните кнопкой мыши на последнем выражении в методе `Main`. Мы хотим, чтобы до этого момента весь код выполнился — это позволит создать оба объекта и затем проанализировать их с помощью отладчика.
2. Выберите команду меню `Debug > Toggle Breakpoint` (Отладка > Включить/отключить точку останова) или нажмите клавишу `F9`. На левой вертикальной панели появится красный круг, а строка кода будет выделена красным — это означает, что точка останова успешно установлена. Повторное нажатие клавиши `F9` отключает точку. Вы также можете щелкнуть на поле слева от строки кода, чтобы быстро включить или отключить точку останова. Щелчок правой кнопкой мыши по точке откроет контекстное меню с дополнительными действиями — например, удаление, временное отключение, настройка условий или добавление действий, которые будут выполняться при срабатывании.
3. Выберите команду меню `Debug > Start Debugging` (Отладка > Начать отладку) или нажмите клавишу `F5`. Visual Studio запустит консольное приложение и приостановит его выполнение при достижении точки останова. Это состояние называется режимом приостановки. После перехода в режим отладки на экране могут появиться дополнительные панели: `Locals` (Локальные переменные), `Watch 1` (Контрольные значения), `Call Stack` (Стек вызовов), `Exception Settings` (Настройки исключений) и `Immediate Window` (Окно немедленного выполнения). Также появится панель инструментов `Debugging` (Отладка). Стока, которая будет выполнена следующей, выделяется желтым цветом, а желтая стрелка на левой панели указывает на текущую позицию выполнения.
4. В терминале обратите внимание на вывод свойств. Некоторые значения могут отображаться в закодированном виде, что затрудняет их анализ — особенно если вы пытаетесь отлаживать код вручную. Именно поэтому важно использовать инструменты отладки, которые позволяют увидеть объекты в удобной форме.

```
ConnectionString: Server=(localdb)\mssqllocaldb;Database=Northwind;Trusted_Connection=true
Base64Encoded: RGVidWdnaw5nIGlzIGNvb2wh
JsonWebToken: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

5. На панели `Locals` (Локальные переменные) разверните переменную `config` и нажмите кнопку `View` (Просмотр) рядом со свойством `Base64Encoded`, чтобы открыть окно `Text Visualizer` (Визуализатор текста) (рис. 4.1).

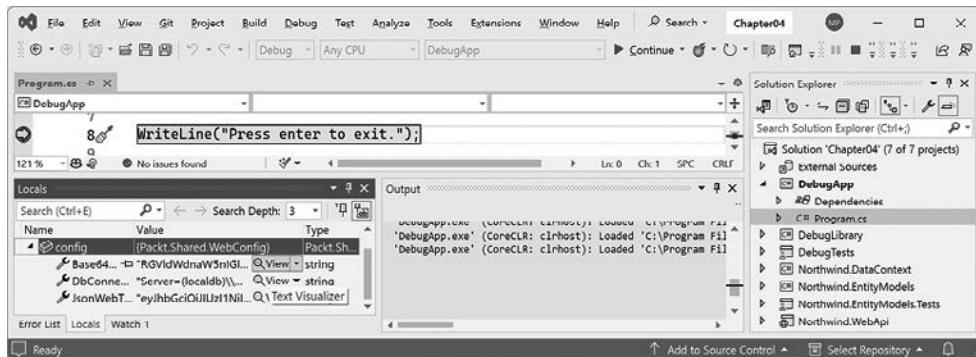


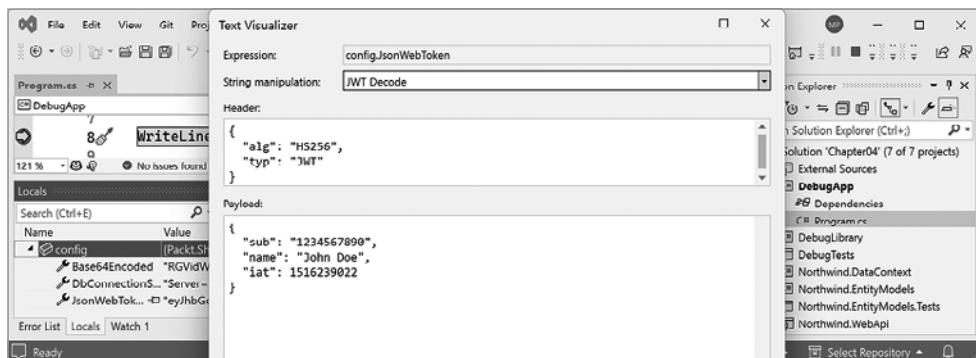
Рис. 4.1. Окно Locals и кнопки для просмотра значений свойств

6. В открывшемся окне Text Visualizer (Визуализатор текста) в раскрывающемся списке String manipulation (Манипуляции со строкой) выберите пункт Base64 Decode (Декодировать Base64). Вы увидите исходное сообщение: Debugging is cool! (рис. 4.2).



Рис. 4.2. Декодирование значения свойства в формате Base64 с помощью инструмента Text Visualizer

7. Закройте диалоговое окно Text Visualizer (Визуализатор текста).
8. На панели Locals (Локальные переменные) нажмите кнопку View (Просмотр) рядом со свойством JsonWebToken, чтобы снова открыть диалоговое окно Text Visualizer (Визуализатор текста).
9. В окне Text Visualizer (Визуализатор текста) в раскрывающемся списке String manipulation (Манипуляции со строкой) выберите пункт JWT Decode (Декодировать JWT). Теперь вы сможете прочитать расшифрованное содержимое токена в формате JSON — в том числе информацию о примененном алгоритме и имени пользователя (рис. 4.3).
10. Закройте диалоговое окно Text Visualizer (Визуализатор текста).

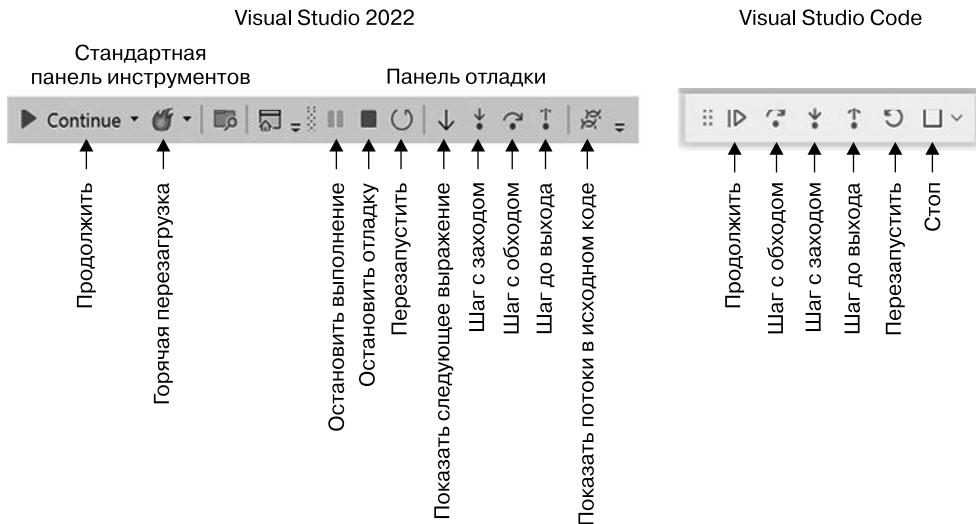


**Рис. 4.3.** Декодирование значения свойства в формате JWT с помощью инструмента Text Visualizer

## Навигация с помощью панели инструментов отладки

В Visual Studio на панели инструментов Standard (Стандартная) расположены две основные кнопки, связанные с отладкой: одна — для запуска или продолжения выполнения программы, другая — для горячей перезагрузки кода во время отладки. Дополнительно доступна панель Debug (Отладка), где собраны остальные инструменты отладки.

В Visual Studio Code отображается компактная плавающая панель с кнопками для управления отладкой — она обеспечивает быстрый доступ ко всем основным действиям (рис. 4.4).



**Рис. 4.4.** Панели инструментов отладки в Visual Studio и Visual Studio Code

В списке ниже описаны наиболее часто используемые кнопки на панелях отладки в Visual Studio и Visual Studio Code.

- **Start (Старт) / Continue (Продолжить) / F5** — кнопка контекстная: запускает проект, если он еще не выполняется, или продолжает выполнение с текущей позиции до завершения или до следующей точки останова.
- **Hot Reload (Горячая перезагрузка)** — позволяет применить изменения кода без перезапуска приложения.
- **Break All (Остановить выполнение)** — приостанавливает выполнение приложения на ближайшей доступной строке, позволяя немедленно начать отладку в текущем контексте.
- **Stop Debugging (Остановить отладку) / Shift+F5** — завершает текущий сеанс отладки.
- **Restart (Перезапустить) / Ctrl+Shift+F5** — останавливает отладку и немедленно запускает приложение заново с подключенным отладчиком.
- **Show Next Statement (Показать следующее выражение)** — перемещает текстовый курсор к следующей строке кода, которую выполнит отладчик.
- **Step Into (Шаг с заходом) / F11, Step Over (Шаг с обходом) / F10, Step Out (Шаг до выхода) / Shift+F11** (синие стрелки над точками) — три основные команды пошагового выполнения:
  - **Step Into (Шаг с заходом)** — заходит внутрь вызываемого метода и позволяет отладить его построчно;
  - **Step Over (Шаг с обходом)** — выполняет вызов метода целиком, не заходя внутрь;
  - **Step Out (Шаг до выхода)** — доходит до конца текущего метода и останавливается после возврата из него.
- Разница между выполнением с заходом в метод и без него особенно заметна при отладке вызовов методов.
- **Show Threads in Source (Показать потоки в исходном коде)** — открывает список активных потоков и позволяет переключаться между ними.

## Панели отладки

Во время отладки Visual Studio и Visual Studio Code отображают специальные панели, на которых можно отслеживать переменные, стек вызовов, точки останова и другие важные данные прямо во время выполнения кода.

- **VARIABLES (Переменные)**, включая панель **Locals (Локальные переменные)** — здесь автоматически отображаются имя, значение и тип всех локальных переменных. Следите за этой панелью во время выполнения кода, чтобы видеть текущее состояние.

- WATCH (Отслеживаемые значения) или Watch 1 (Контрольные значения) — показывает значения переменных и выражений, которые вы добавили вручную. Полезно, когда вы хотите отслеживать конкретные вычисления или состояния.
- CALL STACK (Стек вызовов) — отображает последовательность вызовов функций, ведущих к текущей точке выполнения. Позволяет быстро понять, как программа попала в данное состояние.
- BREAKPOINTS (Точки останова) — содержит список всех установленных точек останова. Здесь можно быстро отключать, удалять или настраивать условия и действия для каждой точки.

В режиме останова в нижней части редактора отображается дополнительная панель.

- DEBUG CONSOLE (Консоль отладки) в Visual Studio Code или Immediate Window (Окно немедленного выполнения) в Visual Studio — позволяет интерактивно взаимодействовать с отлаживаемым кодом. Вы можете, например, ввести имя переменной или выражение вроде `1 + 2` и нажать клавишу `Enter`, чтобы увидеть результат.

## Настройка отображения пользовательских типов в отладчике

Панели Locals (Локальные переменные) и Watch 1 (Контрольные значения) отображают текущие значения переменных и выражений при пошаговом выполнении кода в режиме отладки.

Простые типы — такие как числа и строки — отображаются корректно по умолчанию. Однако при работе с пользовательскими типами отладчик, если не указано иное, показывает результат вызова метода `ToString`. Если в вашем типе этот метод не переопределен, отобразится просто имя типа, например `Northwind.Entity-Models.Product`.

Чтобы отображать более информативные данные, вы можете задать, как именно объекты пользовательского типа должны отображаться на панелях отладки. Для этого в .NET предусмотрены специальные атрибуты, которые позволяют управлять выводом. Подробности приведены в табл. 4.1.



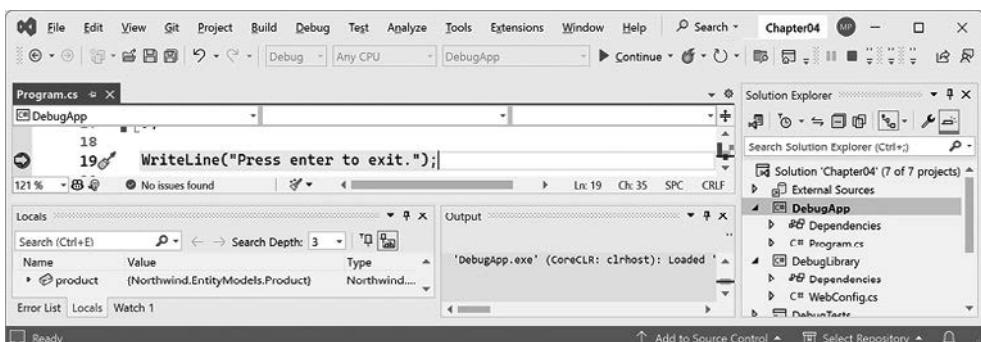
**Внимание!** Если в диалоговом окне Options (Параметры) (команда меню Tools ▶ Options (Средства ▶ Параметры)) на вкладке Debugging (Отладка) установлен флажок Show raw structure of objects in variables windows (Показывать необработанную структуру объектов в окнах переменных), атрибут `DebuggerDisplay` игнорируется. Убедитесь, что этот флажок сброшен, если хотите видеть эффект от настройки отображения.

**Таблица 4.1.** Атрибуты для настройки отображения объектов в отладчике

Атрибут	Описание
DebuggerDisplay	Позволяет задать строку, которая будет отображаться вместо стандартного <code>ToString()</code> . Внутри строки можно использовать фигурные скобки {} — они обозначают имя поля, свойства или метода. Например, <code>{ProductName}</code> выведет значение свойства <code>ProductName</code> . По умолчанию строки отображаются в кавычках. Чтобы убрать кавычки, добавьте суффикс , <code>noq</code>
DebuggerBrowsable	Используется для управления отображением отдельных полей или свойств. Применяется только по одному экземпляру на член. Поддерживает значения перечисления <code>DebuggerBrowsableState</code> : <ul style="list-style-type: none"> <li>• <code>Never</code> — элемент скрыт полностью;</li> <li>• <code>Collapsed</code> — отображается в свернутом виде (по умолчанию);</li> <li>• <code>RootHidden</code> — сам элемент скрыт, но если это коллекция или массив, отображаются только дочерние элементы</li> </ul>
DebuggerStepThrough	Указывает отладчику не заходить внутрь помеченного кода при пошаговом выполнении. Вместо этого отладчик «перешагивает» через него. Полезно для служебных или вспомогательных методов

Выполните следующие шаги.

1. Открыв проект `DebugApp`, выберите команду меню `Debug > Start Debugging` (Отладка > Начать отладку) или нажмите клавишу `F5`.
2. Обратите внимание: переменная `product` отображается со значением `{Northwind.EntityModels.Product}` — то есть используется имя типа по умолчанию (рис. 4.5).
3. Остановите отладку.

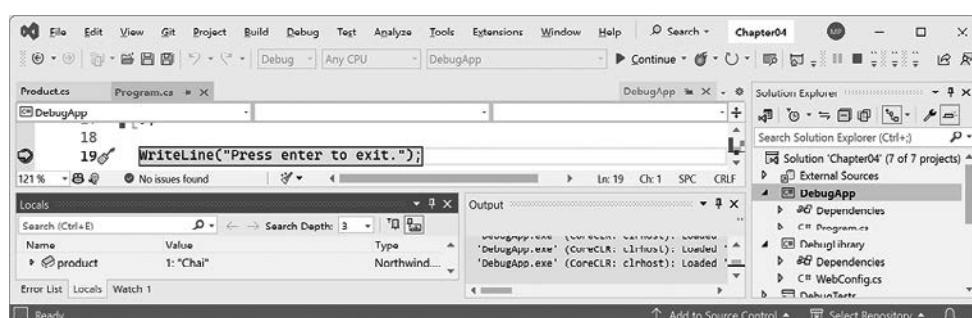


**Рис. 4.5.** По умолчанию класс отображается на панелях отладки по полному имени своего типа

4. В проекте `Northwind.EntityModels` в файле `Product.cs` добавьте атрибут `DebuggerDisplay`, чтобы изменить отображение и показывать значения `ProductId` и `ProductName` во время отладки:

```
[DebuggerDisplay("{ProductId}: {ProductName}")]
public class Product
```

5. В проекте `DebugApp` снова выберите команду меню `Debug > Start Debugging` (Отладка > Начать отладку) или нажмите клавишу F5.
6. Обратите внимание: теперь переменная `product` отображается в формате `1:"Chai"` — именно так, как указано в атрибуте `DebuggerDisplay` (рис. 4.6).



**Рис. 4.6.** Класс с пользовательским отображением значений на панелях отладки

7. В проекте `Northwind.EntityModels` в файле `Product.cs` добавьте суффикс `,nq` к имени свойства `ProductName` в атрибуте `DebuggerDisplay`, чтобы убрать кавычки вокруг строки. Пример:

```
[DebuggerDisplay("{ProductId}: {ProductName,nq}")]
public class Product
```

8. В проекте `DebugApp` выберите команду меню `Debug > Start Debugging` (Отладка > Начать отладку) или нажмите клавишу F5.
9. Обратите внимание: теперь объект `product` отображается как `1:Chai`, без кавычек вокруг имени.

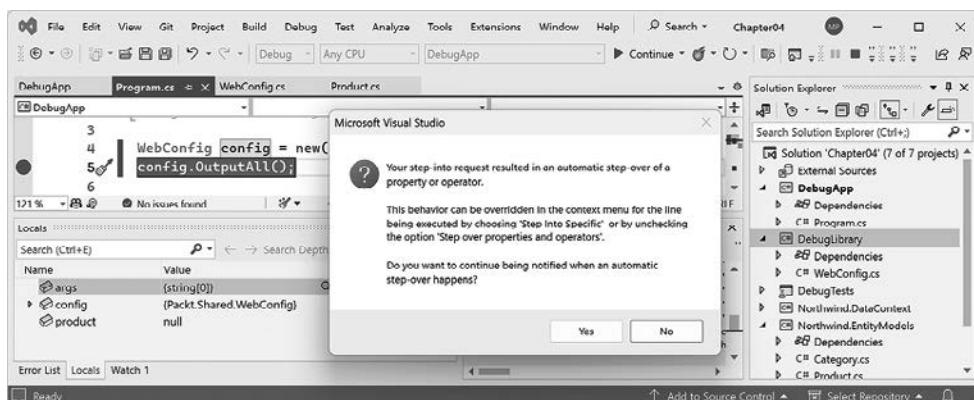
10. В проекте `DebugLibrary` в файле `WebConfig.cs` добавьте к свойству подключения к базе данных атрибут `DebuggerBrowsable`, чтобы скрыть его на панелях отладки:

```
[DebuggerBrowsable(DebuggerBrowsableState.Never)]
public string? DbConnectionString { get; set; }
```

11. В проекте `DebugLibrary` в файле `WebConfig.cs` пометьте метод `OutputAll` атрибутом `DebuggerStepThrough`, чтобы отладчик автоматически пропускал этот метод при пошаговом выполнении:

```
[DebuggerStepThrough]
public void OutputAll()
```

12. Установите точку останова на строке, вызывающей метод `OutputAll`.
13. В проекте `DebugApp` выберите команду меню `Debug > Start Debugging` (Отладка > Начать отладку) или нажмите клавишу `F5`.
14. Нажмите кнопку `Step Into` (Зайти в метод и выполнить), клавишу `F11` или выберите команду меню `Debug > Step Into` (Отладка > Зайти в метод и выполнить). Отладчик отобразит предупреждение о том, что вызов метода будет автоматически пропущен, и объяснит, как изменить это поведение (рис. 4.7).



**Рис. 4.7.** Предотвращение захода в метод или свойство, помеченное атрибутом `DebuggerStepThrough`

15. На панели `Locals` (Локальные переменные) разверните переменную `config` и убедитесь, что свойство `ConnectionString` не отображается.
16. Остановите отладку.

## Отладка тестовых проектов

Посмотрим, как отладить тестовый проект.

1. В редакторе кода создайте C#-проект `xUnit Test Project` (Тестовый проект `xUnit`) / `xunit`, присвойте ему имя `DebugTests` и добавьте в решение `Chapter04`.
2. В файле `DebugTests.csproj` проекта добавьте ссылку на проект библиотеки классов:

```
<ItemGroup>
  <ProjectReference Include="..\DebugLibrary\DebugLibrary.csproj" />
</ItemGroup>
```

3. Выполните сборку проекта `DebugTests`.
4. В проекте `DebugTests` переименуйте файл `UnitTest1.cs` в `DebugLibraryTests.cs`.
5. В этом файле добавьте тестовый метод, создающий объект `WebConfig` и проверяющий, что его свойства не равны `null`:

```

using Packt.Shared; // Подключение объекта WebConfig

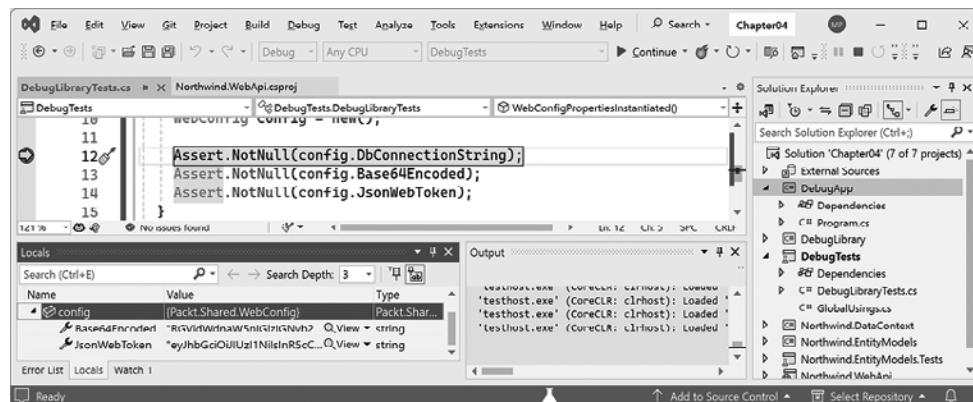
namespace DebugTests;

public class DebugLibraryTests
{
    [Fact]
    public void WebConfigPropertiesInstantiated()
    {
        WebConfig config = new();

        Assert.NotNull(config.DbConnectionString);
        Assert.NotNull(config.Base64Encoded);
        Assert.NotNull(config.JsonWebToken);
    }
}

```

- Установите точку останова на первой строке метода теста.
- Выберите команду меню **Test > Debug All Tests** (Тесты > Отладка всех тестов).
- Обратите внимание: выполнение остановится на установленной точке останова (рис. 4.8).
- Остановите отладку.



**Рис. 4.8.** Остановка на точке останова в тестовом проекте

## Отладка с помощью GitHub Copilot Chat

Инструмент GitHub Copilot Chat уже успел зарекомендовать себя как один из самых популярных помощников для разработчиков. Как сказал генеральный директор Microsoft Сатья Наделла, согласно отчету о прибыли за 2024 год, GitHub Copilot — самый широко используемый ИИ-инструмент для разработчиков в мире. На момент написания этой книги у GitHub Copilot было более 1,3 миллиона платных подписчиков.

Кроме того, Azure OpenAI используется более чем половиной компаний из списка Fortune 500, а платформа Azure уже поддерживает новейшие модели OpenAI, включая GPT-4 Turbo, GPT-4 with Vision и DALL·E 3.



**В копилку знаний:** подробнее об успехах GitHub Copilot можно прочитать по адресу [visualstudiomagazine.com/articles/2024/02/05/copilot-numbers.aspx](https://visualstudiomagazine.com/articles/2024/02/05/copilot-numbers.aspx).

Инструмент GitHub Copilot Chat может оказаться особенно полезным в процессе отладки, предлагая рекомендации и советы в режиме реального времени. Благодаря доступу к вашей кодовой базе он «понимает» контекст и способен не только генерировать фрагменты кода, но и объяснять, что именно не так, почему возникла проблема и как ее можно устраниить. Особенно он полезен при диагностике сложных ошибок, когда человек физически не может удержать в голове все детали текущего состояния проекта.

Рассмотрим несколько способов, с помощью которых GitHub Copilot Chat может помочь при отладке проекта на платформе .NET.

- **Анализ причин ошибок.** Если вы столкнулись, например, с исключением `NullReferenceException`, опишите ситуацию в Copilot Chat: где и при каких условиях возникает ошибка. Copilot может предположить возможные причины — например, неинициализированный объект или метод, неожиданно возвращающий `null`.
- **Рекомендации по отладке и профилированию.** Если приложение работает медленно и вы не знаете, как найти узкое место, Copilot Chat может подсказать, какие инструменты и приемы использовать. Например, он может порекомендовать Visual Studio Diagnostic Tools или JetBrains dotTrace для профилирования производительности и поиска горячих участков кода.
- **Генерация тестов.** Если вы хотите написать модульные тесты для обнаружения ошибки в конкретном методе, просто укажите его сигнатуру и опишите ожидаемое поведение. Copilot сгенерирует шаблон теста с использованием xUnit — включая тестовые данные, вызов метода с разными аргументами и утверждения (`Assert`) для проверки результата.
- **Поиск и устранение типичных ошибок.** При работе с асинхронным кодом вы можете столкнуться с тупиковыми ситуациями (deadlocks) или условиями гонки (race conditions). Copilot Chat помогает выявить распространенные ошибки, например блокировку асинхронного кода из-за вызовов `.Result` или `.Wait()`, и подсказывает лучшие практики, такие как корректное использование ключевого слова `await`, чтобы избежать подобных проблем.
- **Предложения по рефакторингу.** Если код работает, но выглядит громоздким, плохо читается или содержит повторяющиеся конструкции, Copilot Chat может предложить варианты рефакторинга: разбиение длинных методов на более

компактные, устранение дублирования и применение паттернов проектирования для повышения читаемости и повторного использования. О приемах рефакторинга подробнее рассказывается в главе 2.

- **Объяснение ошибок и исключений.** Если вы получили непонятное сообщение компилятора или исключение во время выполнения, просто вставьте его в Copilot Chat. Он объяснит смысл ошибки человеческим языком и предложит возможные способы устранения. Например, при ошибке `StackOverflowException` Copilot может подсказать, что стоит проверить наличие бесконечной рекурсии или чрезмерного использования стека.
- **Генерация кода для отладки.** Если вы хотите добавить журналирование в свое приложение, чтобы выяснить причину сложной ошибки, Copilot Chat может сгенерировать код для интеграции системы журналирования, например Serilog, включая настройку уровней журналирования и выходных каналов (`sink`).

Рассмотрим пример взаимодействия с GitHub Copilot Chat:

**Вы:** «Я получаю `NullReferenceException`, когда пытаюсь обратиться к свойству объекта `User` в цикле. Что может быть причиной этой ошибки?»

**Copilot:** Исключение `NullReferenceException` возникает, когда вы пытаетесь получить доступ к члену объекта, который равен `null`. Вот несколько моментов, на которые стоит обратить внимание.

- Убедитесь, что объект `User` инициализирован, перед тем как обращаться к его свойствам.
- Если `User` получен из коллекции, проверьте, что ни один из элементов этой коллекции не равен `null`.
- Добавьте явную проверку на `null` перед обращением к свойствам. Вот пример безопасного доступа к свойству:

```
foreach (var user in users)
{
    if (user != null)
    {
        Console.WriteLine(user.PropertyName);
    }
}
```

GitHub Copilot Chat — это мощный помощник для отладки, способный давать рекомендации по устранению типичных проблем, объяснять концепции, предлагать лучшие практики и генерировать полезные фрагменты кода. Он значительно сокращает время, затрачиваемое на отладку и улучшение .NET-проектов.

Перед тем как перейти к средствам устранения неполадок, связанных с памятью, важно разобраться в основах работы с памятью в .NET-приложениях. Об этом пойдет речь далее.

## Анализ стека и кучи памяти

Одна из самых распространенных проблем в программировании — это управление памятью. Чтобы эффективно устранять связанные с ней неполадки, нужно понимать, как устроено распределение памяти в проектах на платформе .NET.

В .NET (как и в большинстве современных платформ) память делится на две основные области: *стек* и *куча*.

- *Стековая память* работает быстрее, но имеет ограниченный объем. Она управляется напрямую процессором и функционирует по принципу «первым пришел — первым ушел» (LIFO). Благодаря этому данные, размещенные в стеке, легко помещаются в кэши процессора (уровней L1 и L2), поэтому доступ к ним особенно быстр.
- *Память кучи* работает медленнее, но ее объем значительно больше. Она используется для хранения объектов, срок жизни которых не ограничен рамками вызова метода.

Например, в Windows по умолчанию размер стека для приложений с архитектурой ARM64, x86 и x64 составляет 1 Мбайт. В современных дистрибутивах Linux это значение обычно равно 8 Мбайт. В системах macOS и Linux вы можете открыть терминал и ввести команду `ulimit -a`. Она покажет лимиты ресурсов, включая строку вроде `stack size (kbytes) 8192`, что означает ограничение 8192 Кбайт (то есть 8 Мбайт). Остальная память системы при этом может быть практически неограниченной. Из-за этого ограничения стек легко переполнить, особенно при глубокой или бесконечной рекурсии, что приводит к ошибке `StackOverflowException`.

Какой тип памяти используется — стек или куча, — зависит от того, как определен тип в .NET. Об этом мы поговорим в следующем разделе.

## Как ссылочные и типы значений хранятся в памяти

В C# есть три ключевых слова, с помощью которых можно определить тип: `class`, `record` и `struct`. Все эти типы могут включать одинаковые элементы — поля, свойства, методы и т. д. Главное различие между ними заключается в том, где и как их данные размещаются в памяти.

- Когда вы определяете тип с помощью ключевого слова `class` или `record`, вы создаете *ссылочный тип*. То есть сам объект размещается в куче, а в стеке хранится лишь ссылка (адрес в памяти) на этот объект плюс немного служебной информации. Ссылочные типы всегда используют немного памяти стека, даже если сами данные находятся в куче. (*Если вы представляете себе ссылку как указатель — вы недалеки от истины, но ссылки в C# безопасны по типу. Про указатели мы еще поговорим в этой главе.*)

- Когда вы определяете тип с помощью ключевого слова `struct` или `record struct`, вы создаете *тип значения*. Такой объект хранится в стеке, если он используется как локальная переменная или элемент другого значения.

Однако если структура (`struct`) содержит поля ссылочного типа (например, строку или объект), то эти поля будут храниться в куче. В таком случае данные структуры частично размещаются в стеке, а частично — в куче.



**Дельный совет:** все структуры (`struct`) в базовой библиотеке классов .NET (BCL), как правило, не содержат полей ссылочного типа. И вам стоит придерживаться того же подхода. В частности, вот неудачный пример:

```
public struct MyValueType
{
    // Это неудачный прием, потому что string – ссылочный тип
    public string Name;
}
```

Перечислю наиболее распространенные типы, определенные как `struct`:

- числовые типы пространства имен `System`: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`;
- другие встроенные типы из `System`: `char`, `DateTime`, `DateOnly`, `TimeOnly`, `bool`;
- типы из пространства имен `System.Drawing`: `Color`, `Point`, `PointF`, `Size`, `SizeF`, `Rectangle`, `RectangleF`.

Почти все остальные типы в .NET — это ссылочные типы (`class`), включая такие широко используемые типы, как `string` (`System.String`) и `object` (`System.Object`).



Помимо различий в том, где хранятся данные (в стеке или куче), есть и другие важные отличия:

- от структуры нельзя наследоваться;
- сравнение экземпляров `struct` выполняется по значению, а не по ссылке — то есть сравниваются все поля, а не адрес в памяти.

Допустим, в консольном приложении вызывается метод, где задействованы и ссылочные, и значимые типы:

```
void SomeMethod()
{
    int number1 = 49;
    long number2 = 12;
    System.Drawing.Point location = new(x: 4, y: 5);

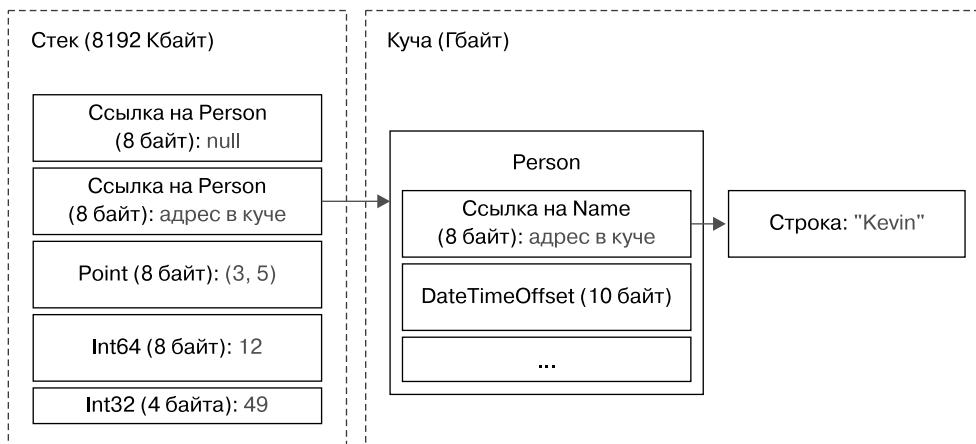
    Person kevin = new() { Name = "Kevin",
        Born = new(1988, 9, 23, 0, 0, TimeSpace.Zero) };

    Person sally;
}
```

Посмотрим, какая память выделяется в стеке и куче при выполнении этого метода (рис. 4.9 и список ниже).

- Переменная `number1` — это тип значения (value type), то есть `struct`, поэтому для нее выделяется место в стеке. Она занимает 4 байта памяти, так как представляет собой 32-битное целое число. Значение 49 хранится непосредственно в переменной.
- Переменная `number2` — также тип значения. Для нее в стеке выделяется 8 байт, поскольку она представляет собой 64-битное целое число.
- Переменная `location` — это тоже тип значения. Она занимает 8 байт в стеке, так как состоит из двух 32-битных чисел `x` и `y`.
- Переменная `kevin` — ссылочный тип (`class`), поэтому в стеке хранится только ссылка (64-битный адрес, 8 байт), а сам объект `Person` размещается в куче. Объем памяти в куче зависит от свойств объекта.
- Переменная `sally` — также ссылочный тип. В стеке для нее выделяется 8 байт, но, поскольку она пока не указывает ни на какой объект (`null`), память в куче не выделяется. Если позже присвоить `sally` значение переменной `kevin`, в `sally` будет скопирован тот же адрес памяти, что и у `kevin`, и обе переменные будут указывать на один и тот же объект `Person` в куче.

```
Sally = kevin; // Обе переменные указывают на один и тот же объект Person в куче
```



**Рис. 4.9.** Как распределяются типы значений и ссылочные типы в памяти

Вся память, выделенная для ссылочного типа, размещается в куче, за исключением ссылки на объект, которая хранится в стеке. Если, например, у класса `Person` есть поле типа `DateTimeOffset` (это тип значения), то значение этого поля тоже будет

храниться в куче — вместе с объектом `Person`, поскольку принадлежит ссылочному типу (см. рис. 4.9).

Если у типа значения есть поле ссылочного типа, то эта часть структуры будет храниться в куче. Но если тип значения состоит только из полей, которые тоже являются типами значений, весь объект может быть полностью размещен в стеке. Например, `Point` — это тип значения, состоящий из двух целых чисел `x` и `y`. Оба этих поля — тоже типы значений, поэтому весь объект `Point` помещается в стек. Если бы `Point` содержал поле типа `string` (ссылочный тип), то это поле — а точнее, байты строки — хранилось бы в куче.

Когда метод завершает выполнение, вся стековая память, выделенная внутри него, автоматически освобождается — это происходит мгновенно и безопасно, поскольку стек работает по принципу LIFO (последним пришел — первым ушел). А вот память, выделенная в куче, продолжает существовать даже после завершения метода — до тех пор, пока за нее не возьмется *сборщик мусора* (.NET Garbage Collector, GC), который отслеживает неиспользуемые объекты и освобождает занимаемую ими память. Сборщик мусора не работает мгновенно — он запускается по мере необходимости, чтобы не тормозить работу приложения. Об этом мы поговорим чуть позже.

После завершения одного метода приложение может вызвать другой, который получит новую стековую область. Таким образом, стек — это буквально стек: память выделяется на вершине, а затем удаляется оттуда, когда больше не нужна.

В C# разработчик не управляет вручную выделением и освобождением памяти. Память выделяется автоматически при входе в метод и освобождается при выходе из него. Такой подход называют *verifiably safe code* — это код, который можно статически проверить на корректное использование памяти и безопасность выполнения.

## Работа с небезопасным кодом в C#

В языке C# можно использовать *небезопасный код* (`unsafe`), чтобы вручную управлять памятью. Полезно в ситуациях, где требуется максимально высокая производительность. Одним из примеров такой работы является использование ключевого слова `stackalloc`, которое позволяет выделить блок памяти прямо в стеке. Эта память автоматически освобождается при выходе из метода, в котором она была выделена.



**В копилку знаний:** подробную информацию о небезопасном коде и использовании `stackalloc` можно найти по следующим адресам: [learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/unsafe-code](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/unsafe-code); [learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/stackalloc](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/stackalloc).

## Что нужно знать об указателях в C#

Указатели — это низкоуровневая конструкция, традиционно ассоциируемая с неуправляемыми языками программирования, такими как С и С++. Указатель представляет собой переменную, которая содержит адрес другой переменной в памяти.

Указатели позволяют эффективно работать с данными и изменять их напрямую. Они активно используются для реализации таких структур данных, как связанные списки, деревья и графы, где требуется *динамическое связывание элементов* без физического перемещения данных в памяти.

Однако при работе с указателями легко допустить ошибки. Например, выполнить разыменование неинициализированного указателя (так называемый дикий указатель) или получить висячий указатель, указывающий на уже освобожденную область памяти. Такие ошибки могут привести к сбоям, непредсказуемому поведению программы или даже уязвимостям безопасности — например, атакам через переполнение буфера.

Платформа .NET изначально спроектирована как управляемая среда, в которой разработчику не требуется вручную управлять памятью. Она обеспечивает автоматическую сборку мусора (GC), безопасность типов и защиту от большинства низкоуровневых ошибок. Тем не менее в .NET предусмотрена возможность работы с указателями — внутри небезопасных блоков кода (`unsafe`). Полезно при интеграции с неуправляемым (`native`) кодом, для высокоеффективных операций с памятью и при прямом доступе к низкоуровневым системным ресурсам.

Чтобы использовать указатели и писать небезопасный код в C#, нужно выполнить два условия:

- разрешить небезопасный код в параметрах проекта;
- пометить код как `unsafe`. С ним можно объявить блок кода, метод, тип или отдельного члена класса.

Небезопасный код обходит систему безопасности типов .NET, поэтому требует специальных разрешений для выполнения. Он позволяет напрямую работать с памятью, как в языках С и С++, включая объявление указателей, использование ключевого слова `stackalloc` для выделения памяти в стеке, выполнение арифметических операций с указателями и разыменование указателей для доступа к данным.

Чтобы объявить указатель в C#, добавьте к типу данных звездочку (\*):

```
int* pointer; // указатель на целое число
```

Чтобы разыменовать указатель (то есть получить значение по адресу), также используйте звездочку перед именем переменной:

```
int number = *pointer; // разыменование указателя
```

Рассмотрим пример.

1. В редакторе кода создайте проект **Console App** (Консольное приложение) / **console**, присвойте ему имя **PointersApp** и добавьте в решение **Chapter04**.
2. В файле **PointersApp.csproj** добавьте код для статического импорта класса **System.Console** во все файлы; обработки предупреждений как ошибки и разрешения небезопасных блоков кода:

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

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

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

</Project>
```

3. В файле **Program.cs** удалите все содержимое и добавьте следующий код, который суммирует массив из пяти целых чисел с использованием указателей:

```
unsafe
{
    // Количество целых чисел
    const int size = 5;

    // Выделение блока памяти в стеке.
    // numbers – указатель на первый элемент блока
    int* numbers = stackalloc int[size];

    // Инициализация массива
    for (int i = 0; i < size; i++)
    {
        *(numbers + i) = i + 1;
    }

    // Оперирование указателями и разыменование
    int sum = 0;
    for (int* ptr = numbers; ptr < numbers + size; ptr++)
    {
        sum += *ptr; // Разыменование указателя для получения значения
    }

    WriteLine($"The sum is: {sum}");
}
```

Обратите внимание на следующие моменты, касающиеся показанного кода.

- Небезопасный блок (`unsafe`) позволяет использовать указатели внутри метода.
- `int* numbers` — объявление указателя на целое число.
- Ключевое слово `stackalloc` выделяет в стеке память под массив из пяти целых чисел. Переменная `numbers` указывает на первый элемент этого блока. Память, выделенная с помощью `stackalloc`, автоматически освобождается при выходе из области видимости, то есть при завершении метода `Main`, когда консольное приложение завершает работу.
- Цикл `for (int* ptr = numbers; ptr < numbers + size; ptr++)` перебирает значения с помощью указателя: переменная `ptr` последовательно указывает на следующий элемент массива.
- `*ptr` — операция разыменования: позволяет получить значение по адресу, на который указывает указатель, и использовать его в вычислениях (например, при суммировании).

4. Запустите проект без отладки и проанализируйте результат:

```
The sum is: 15
```

В этом примере мы рассмотрели простейший сценарий работы с указателями. Но в .NET указатели применяются и в более сложных сценариях.

## Дополнительные области применения указателей

Указатели также применяются при вызове внешних платформенных функций, что известно как P/Invoke. Благодаря этому можно обращаться к неуправляемым функциям, определенным в библиотеках DLL, например в функциях Win32 API. Указатели нередко используются для передачи параметров по ссылке или для работы с неуправляемыми структурами и буферами.

По тем же причинам указатели могут применяться при взаимодействии с COM-объектами через COM Interop — например, для обработки ссылок на COM-интерфейсы или для передачи данных между управляемым и неуправляемым кодом.



Component Object Model (COM) — технология Microsoft, предназначенная для взаимодействия программных компонентов независимо от языков, на которых они разработаны. Хотя COM считается устаревшей, такие объекты до сих пор применяются в различных сценариях, особенно в среде Windows. COM активно используется для автоматизации задач и интеграции с программами Microsoft Office — Word, Excel, Outlook и PowerPoint. Многие элементы пользовательского интерфейса Windows, включая обработчики контекстного меню, накладные значки и панели инструментов в обозревателе Windows, реализованы в виде COM-компонентов. Компонент Windows Script Host (WSH) и сценарные языки, такие как VBScript, тоже применяют COM для автоматизации действий, взаимодействия с файловой системой и других задач в среде Windows.

В критически важных для производительности разделах приложения указатели могут использоваться для непосредственного доступа к памяти, что помогает снизить накладные расходы и повысить скорость обработки данных или алгоритмов, где требуется точный контроль над управлением памятью.

Небезопасный код может оказаться более эффективным в задачах, связанных с обработкой больших массивов или выполнением сложных числовых расчетов, — в таких случаях отключение проверок границ массива может существенно повысить производительность.

Указатели также используются для прямого доступа к файлам, отображенными в память. Благодаря этому можно выполнять высокопроизводительные операции ввода-вывода, обрабатывая содержимое файла напрямую, без дополнительных копирований.

В сценариях, где необходимо прямое взаимодействие с аппаратным обеспечением или с устройствами, отображенными в память, указатели и небезопасный код обеспечивают требуемый уровень контроля и производительности.



**Внимание!** Небезопасный код может представлять угрозу для безопасности и вызывать проблемы со стабильностью — например, повреждение памяти, выход за пределы буфера и другие типичные для неуправляемого кода ошибки. Приложения, в которых используется небезопасный код, могут оказаться менее переносимыми между различными средами исполнения и платформами .NET, поскольку они зависят от прямого доступа к памяти и могут содержать специфические для платформы зависимости. Выполнение небезопасного кода требует полного уровня доверия в области безопасности, который может быть недоступен в некоторых сценариях — особенно в ограниченных или частично доверенных средах, таких как определенные веб-серверы или клиентские компьютеры. В связи с рисками и повышенной сложностью использование указателей и небезопасного кода в .NET, как правило, не рекомендуется, если только не возникает острой необходимости.

## Упаковывание значений

Боксинг (упаковывание) никак не связан с ударами по лицу — хотя разработчикам на Unity, пытающимся выжать максимум из ограниченной памяти, это может показаться именно так.

В C# *упаковывание* означает перемещение значения из стека в кучу и обрачивание его в экземпляр типа `System.Object`. Обратный процесс — распаковывание — возвращает значение обратно в стек и всегда выполняется явно. Упаковывание происходит в фоновом режиме, поэтому разработчик может даже не подозревать, что оно имеет место. При этом оно может работать до 20 раз медленнее, чем обычное перемещение значений без упаковки.

Например, значение типа `int` может быть как упаковано, так и распаковано:

```
int n = 3;
object o = n; // Упаковывание происходит неявно
n = (int)o; // Распаковывание выполняется только в явном виде
```

Один из самых распространенных сценариев, в которых упаковка происходит не-заметно для разработчика, — это передача значений в форматированные строки:

```
string name = "Hilda";
DateTime hired = new(2024, 2, 21);
int days = 5;

// hired и days — это типы значений, они будут упакованы
Console.WriteLine("{0} hired on {1} for {2} days.", name, hired, days);
```

Переменная `name` не упаковывается, так как `string` — ссылочный тип, а значит, уже размещен в куче.

Операции упаковывания и распаковывания могут отрицательно сказываться на производительности. Хотя разработчику на .NET полезно понимать, как работает упаковка, и стараться избегать ее, в большинстве типов проектов .NET и во многих сценариях об этом не стоит слишком переживать — накладные расходы от упаковки обычно меньше, чем, например, задержки при сетевых вызовах или обновлении пользовательского интерфейса.

Но в играх, разработанных на платформе Unity, ситуация иная. Сборщик мусора Unity не так быстро и автоматически освобождает упакованные значения, поэтому здесь особенно важно избегать упаковки, где только возможно. По этой причине JetBrains Rider с установленным плагином Unity Support будет ругаться на операции упаковки при их обнаружении в коде. К сожалению, такая проверка выполняется без учета контекста и не различает проекты Unity и .NET.



**В копилку знаний:** дополнительная информация об упаковывании значений доступна по адресу [learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/boxing-and-unboxing](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/boxing-and-unboxing).

## Сборка мусора

Память в стеке освобождается при выходе из метода, в котором она была выделена, поэтому момент ее освобождения легко предсказать. А вот память, выделенная в куче, освобождается не сразу. Этим управляет специальный компонент среды выполнения .NET — сборщик мусора (Garbage Collector, GC), который сам решает, когда и какие объекты больше не используются и могут быть удалены.



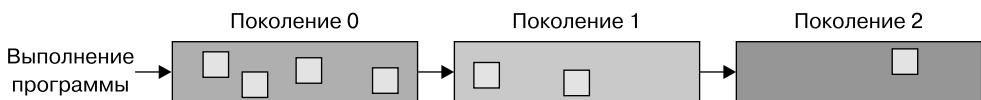
Сокращение GC в документации используется для обозначения как самого процесса сборки мусора, так и компонента среды выполнения и класса GC, управляющего этим процессом.

Когда код создает объект, среда выполнения .NET выделяет для него память в управляемой куче. По мере создания новых объектов куча постепенно увеличивается.

Периодически сборщик мусора (GC) запускает процесс очистки, чтобы освободить память, занятую объектами, которые больше недоступны из приложения. GC начинает с пометки всех активных объектов — тех, на которые все еще существуют прямые или косвенные ссылки от корневых элементов. Корневые элементы включают статические переменные, локальные переменные в стеке и регистры процессора, содержащие ссылки на объекты.

После того как GC определяет активные объекты, он уплотняет кучу, сдвигая эти объекты ближе друг к другу и устранив пустоты, образовавшиеся после удаления недоступных объектов. Таким образом можно уменьшить фрагментацию памяти. Если вы помните времена вращающихся жестких дисков — мы запускали утилиты дефрагментации, чтобы собрать файлы «в кучку». GC выполняет нечто подобное, но применительно к объектам в памяти. Объекты с финализаторами — специальными методами, противоположными конструктором, — получают шанс выполнить финализацию перед тем, как память, которую они занимали, будет освобождена.

В .NET используется система сборки мусора на основе поколений, в которой объекты распределяются по трем поколениям — 0, 1 и 2 — в зависимости от продолжительности их жизни. Новые объекты размещаются в поколении 0. Если объект переживает сборку мусора, он перемещается в следующее поколение. Объекты более высокого поколения очищаются реже, что помогает оптимизировать производительность: предполагается, что долгоживущие объекты будут оставаться в памяти надолго и нет смысла проверять их слишком часто (рис. 4.10).



**Рис. 4.10.** Поколения сборщиков мусора

Поколения GC кратко описаны ниже.

- **Выполнение программы.** Все новые объекты помещаются в поколение 0. Сборка мусора здесь происходит чаще всего.
- **Повышение.** Объекты, пережившие сборку в поколении 0, переходят в поколение 1. Эти объекты уже прожили хотя бы один цикл GC. Сборка мусора в этом поколении запускается реже.
- **Повышение.** Объекты, выжившие после сборки в поколении 1, переходят в поколение 2. Здесь размещаются долгоживущие объекты. Сборка мусора в этом поколении выполняется очень редко.

## Управление сборкой мусора

Хотя сборка мусора выполняется автоматически, в некоторых случаях можно вручную повлиять на ее поведение с помощью класса GC. Например, имеет смысл вызвать сборку мусора принудительно, если вы только что освободили

значительный объем памяти и хотите, чтобы эта память была возвращена немедленно:

```
GC.Collect(); // Принудительный запуск сборки мусора для всех поколений
```

Если у вас есть объект `obj` с финализатором, но его неуправляемые ресурсы уже освобождены вручную, можно запретить вызов финализатора — так вы сможете повысить производительность:

```
GC.SuppressFinalize(obj);
```

Вы можете отслеживать, к какому поколению относится объект, и при необходимости управлять сборкой мусора для конкретного поколения. Например, можно узнать поколение объекта или указать, что сборка мусора должна выполняться только для определенного поколения:

```
int objectGeneration = GC.GetGeneration(obj); // Получение номера поколения,
  // к которому относится объект obj
GC.Collect(0); // Сборка мусора только в поколении 0
```

## Управление ресурсами с помощью `IDisposable`

Неуправляемые ресурсы находятся вне прямого контроля среды выполнения .NET. К ним относятся, например, подключения к базам данных, файлы и другие внешние ресурсы. Хотя сборщик мусора может освободить связанные с ними память, это происходит не сразу. Для безопасной очистки требуются как минимум два прохода GC, если только вы не реализуете интерфейс `IDisposable` и не используете подход явного освобождения ресурсов:

```
public class ResourceWrapper : IDisposable
{
    private bool disposed = false;

    // Реализовать интерфейс IDisposable
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this); // Явно запретить вызов финализатора
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Освободить управляемые ресурсы
            }
        }
    }
}
```

```
// Освободить неуправляемые ресурсы
disposed = true;
}

~ResourceWrapper() // Реализовать финализатор (если он действительно нужен)
{
    Dispose(false);
}
}
```

В приведенном примере метод `Dispose` следует вызывать для освобождения неуправляемого ресурса вручную. В реализации `Dispose` обычно вызывается `GC.SuppressFinalize`, чтобы предотвратить запуск финализатора, — ресурсы уже были освобождены, и повторная очистка не требуется. Такой прием снижает нагрузку на сборщик мусора и повышает эффективность его работы. Если метод `Dispose` не вызывается, то освобождение неуправляемого ресурса произойдет через финализатор. Однако в этом случае потребуются как минимум два прохода GC: сначала для вызова финализатора, а затем — для освобождения выделенной памяти, связанной с объектом в управляемой куче.

Ручное управление сборкой мусора с помощью класса `GC` должно использоваться крайне осторожно. Неправильное применение может привести к снижению производительности. Принудительный вызов `GC` может нарушить оптимизированную автоматическую стратегию, реализованную в .NET. Как правило, такие вызовы уместны только в специфических сценариях — например, при больших и редких выделениях памяти или при управлении неуправляемыми ресурсами.



**Дельный совет:** сборщик мусора в среде выполнения .NET отлично оптимизирован. В большинстве случаев лучше довериться его автоматической работе, чем вмешиваться вручную.



**В копилку знаний:** подробности об улучшениях в GC в .NET 8 можно найти на сайте [devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/#gc](https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/#gc).

## Инструменты и приемы для устранения проблем с памятью

Анализ памяти помогает выявлять утечки, понимать, как расходуется память, и оптимизировать производительность приложения. Экосистема .NET предлагает для этого разнообразные инструменты и методы, подходящие для разных задач и этапов разработки.

## Распространенные инструменты и навыки работы с памятью

Ниже перечислены некоторые часто используемые приемы и инструменты для анализа памяти.

- **Анализ GC.** Анализ журналов сборки мусора помогает понять, как работает GC: с какой частотой он срабатывает, как управляется память и как это влияет на производительность приложения. Ведение журналов GC можно включить через настройки среды выполнения, а затем проанализировать их с помощью таких инструментов, как *PerfView* или *GCStats*.
- **Статический анализ кода.** Такой анализ выполняется без запуска приложения и позволяет выявить потенциальные утечки памяти и неэффективные решения. В .NET SDK уже встроены анализаторы, которые предупреждают о некорректном использовании API, способном привести к проблемам с памятью. Сторонние пакеты также могут включать собственные анализаторы. Например, *ReSharper* и *CodeRush* предоставляют инструменты статического анализа, помогающие находить «запахи кода» и потенциальные утечки на ранней стадии разработки.
- **Инструменты профилирования.** Это самый прямой способ анализа использования памяти: они позволяют измерять объем, потребляемый объектами, и отслеживать моменты выделения памяти. В Visual Studio встроены средства диагностики, включая профилировщик памяти, — с его помощью можно оценить использование памяти и выявить потенциальные утечки. Программа *JetBrains dotMemory* — мощный инструмент профилирования, предоставляющий подробную информацию об использовании памяти, возможных утечках и «трафике» объектов в куче.
- **Анализ кучи.** Такой анализ помогает понять, как распределяются объекты в управляемой куче и как долго они живут. Он особенно полезен для выявления утечек памяти и причин, по которым объекты удерживаются дольше, чем нужно. Инструмент *WinDbg* в сочетании с расширением *SOS* — низкоуровневый и мощный вариант для анализа сложных проблем с памятью, особенно в рабочих (production) средах и при разборе дампов памяти. *PerfView* — универсальный инструмент производительности, умеющий собирать и анализировать данные об использовании памяти, а также другие метрики. Он особенно эффективен при работе с большими объемами диагностических данных.
- **Отслеживание памяти и обнаружение утечек.** Это процесс мониторинга распределения и использования памяти со временем, направленный на выявление утечек. Такой подход может использоваться как на этапе тестирования, так и в производственной среде — с минимальными накладными расходами. В проектах на .NET Visual Studio предлагает встроенные инструменты для анализа дампов памяти и активного мониторинга, позволяющие обнаруживать аномалии в потреблении памяти.

- **Анализ замеров производительности.** Сочетание профилирования памяти с измерением производительности помогает оценить, как меняется потребление памяти при разных условиях нагрузки. Хотя пакет BenchmarkDotNet ориентирован прежде всего на оценки производительности, его можно настроить на сбор памяти — это дает целостную картину влияния изменений в коде на производительность и расход ресурсов.
- **Пользовательское журналирование и мониторинг.** Встраивание пользовательского журналирования в критические участки кода или для отслеживания конкретных объектов помогает выявлять неожиданные шаблоны использования памяти. Системы вроде OpenTelemetry или Serilog с пользовательскими метриками позволяют отслеживать использование памяти и записывать данные, что особенно ценно в производственной среде. Подробнее о метриках и OpenTelemetry рассказывается в главе 5.

Выбор инструмента и подхода зависит от конкретного сценария, стадии разработки и сложности проблем, связанных с памятью.



**Дельный совет:** на ранних этапах разработки наибольшую пользу приносят встроенные средства IDE и статический анализ. В производственной среде часто требуются более надежные инструменты профилирования и анализа кучи — они могут поставляться по платной лицензии.

Для разработчиков на .NET сочетание этих инструментов и техник позволяет всесторонне анализировать использование памяти, улучшать производительность приложений, снижать объем потребляемой памяти и повышать общую стабильность.

Начнем с инструментов, встроенных в самый популярный редактор кода — программу Visual Studio.

## Инструменты Visual Studio

Инструменты, встроенные в программу Visual Studio, помогают анализировать производительность приложений и использование памяти. С их помощью можно выявлять и устранять проблемы, влияющие на эффективность работы. Visual Studio содержит следующие средства для устранения неполадок, связанных с памятью.

- **Инструмент профилирования производительности.** Объединяет несколько видов профилирования — CPU Usage, Memory Usage и GPU Usage, — чтобы представить полную картину производительности приложения. Этот инструмент собирает подробные метрики, включая время работы процессора, распределение памяти и загрузку графического процессора. Благодаря этому разработчики могут точно определить узкие места в производительности. Все данные

собираются и анализируются прямо в Visual Studio, благодаря чему процесс диагностики и оптимизации становится особенно удобным.

- **Инструмент анализа использования памяти.** Позволяет делать моментальные снимки управляемой кучи .NET в разные моменты времени. Сравнивая эти снимки, можно выявить объемы выделения памяти и потенциальные утечки. Инструмент предоставляет информацию об экземплярах объектов, их размере и связях, что помогает понять общую картину использования памяти. Он также выделяет объекты, которые удерживаются в памяти дольше, чем ожидалось, — это облегчает поиск причин утечек памяти.
- **Профилирование процессора в режиме реального времени в отладчике.** Дает возможность отслеживать загрузку процессора прямо во время отладки. Инструмент выделяет строки кода и методы, наиболее нагружающие ЦП. Он встроен в панель инструментов отладки, поэтому профилирование можно включать и отключать по мере необходимости. Благодаря этому можно выявлять проблемы с производительностью еще на ранних этапах, не прибегая к полным сессиям профилирования.



Visual Studio доступна в нескольких редакциях, включая редакцию Community, которая предоставляется бесплатно для индивидуальных разработчиков, проектов с открытым исходным кодом, образовательных учреждений, академических исследований и небольших профессиональных команд.

Между редакцией Community и платными редакциями существуют различия — в первую очередь в доступности расширенных средств диагностики и профилирования. Тем не менее такие инструменты, как Performance Profiler (включая CPU Usage, Memory Usage, GPU Usage и др.) и Memory Usage (Использование памяти), как правило, доступны и в бесплатной редакции.

Сравнение всех редакций Visual Studio представлено на сайте [visualstudio.microsoft.com/vs/compare/](https://visualstudio.microsoft.com/vs/compare/).

## Анализ использования памяти в Visual Studio

Средства диагностики, встроенные в Visual Studio, позволяют измерять использование памяти во времени — это отличный способ понять, как приложение выделяет и расходует память. Такой анализ помогает выявить утечки, чрезмерное потребление памяти и другие проблемы, способные отрицательно сказаться на производительности приложения.

Создадим консольное приложение, которое будет генерировать список объектов в цикле, — это типичный сценарий, на котором удобно наблюдать за использованием памяти.

1. В редакторе кода создайте проект Console App (Консольное приложение) / console, присвойте ему имя `MemoryApp` и добавьте в решение `Chapter04`.

2. В файле `MemoryApp.csproj` добавьте статический импорт класса `System.Console` для всех файлов проекта и настройте обработку предупреждений как ошибок.
3. В файле `Program.cs` удалите все содержимое и напишите выражения, создающие список из десяти тысяч объектов типа `Person`, — такой объем данных даст заметную нагрузку на память:

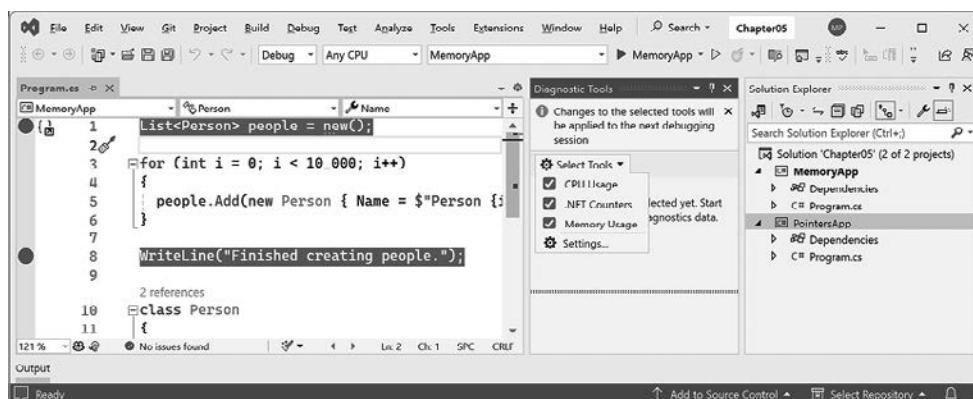
```
List<Person> people = new();

for (int i = 0; i < 10_000; i++)
{
    people.Add(new Person { Name = $"Person {i}", Age = i });
}

WriteLine("Finished creating people.");

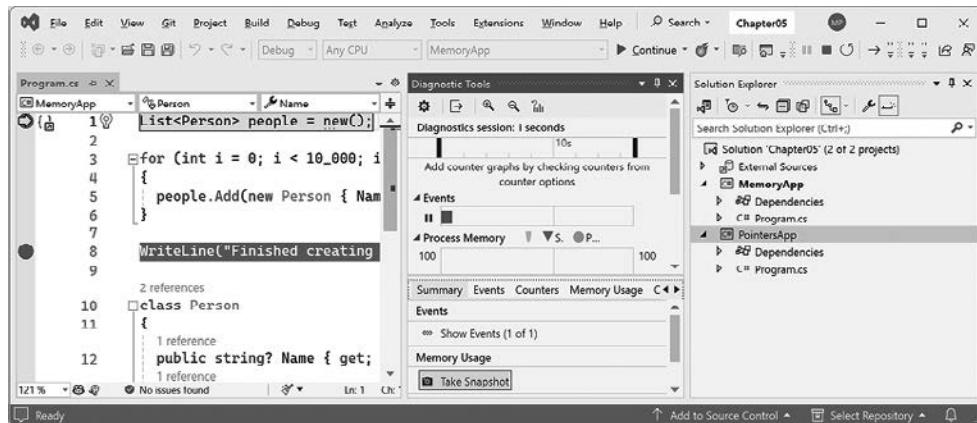
class Person
{
    public string? Name { get; set; }
    public int Age { get; set; }
}
```

4. Установите две точки останова:
  - на выражении, в котором создаются объекты;
  - на выражении, выводящем информацию в консоль.
5. Выберите команду меню `Debug > Windows > Show Diagnostics` (Отладка > Окна > Показать средства диагностики) и закрепите панель.
6. На панели `Diagnostics Tools` (Инструменты диагностики) нажмите кнопку `Select Tools` (Выбрать инструменты) и убедитесь, что выбраны параметры `CPU Usage` (Использование ЦП), `.NET Counters` (Счетчики .NET) и `Memory Usage` (Использование памяти) (рис. 4.11).



**Рис. 4.11.** Настройка инструментов диагностики

7. Выберите команду меню Debug ▶ Start Debugging (Отладка ▶ Начать отладку), нажмите клавишу F5 или нажмите кнопку запуска в окне MemoryApp.
8. На панели Diagnostics Tools (Инструменты диагностики) в разделе Process Memory (Память процесса) на вкладке Summary (Резюме) нажмите кнопку Take Snapshot (Сделать моментальный снимок) (�� Take Snapshot) (рис. 4.12).



**Рис. 4.12.** Создание моментального снимка

9. Обратите внимание: вкладка Memory Usage (Использование памяти) стала активной, и на ней появился первый моментальный снимок. Это – базовое состояние, до создания списка объектов.
10. Выберите команду меню Debug ▶ Continue (Отладка ▶ Продолжить) или нажмите клавишу F5.
11. На вкладке Memory Usage (Использование памяти) в мини-панели инструментов снова нажмите кнопку Take Snapshot (Сделать снимок) (��). Появится второй снимок, рядом с которым отобразятся красные стрелки, указывающие на увеличение затрат памяти (рис. 4.13).
12. Щелкните кнопкой мыши на второй записи в столбце Objects (Diff) (Объекты (Различия)) или Heap Size (Diff) (Размер кучи (Различия)), чтобы открыть представление кучи, отсортированное по размеру.
13. В верхней части окна Managed Memory (Управляемая память) установите флагок Show dead objects (Показать неиспользуемые объекты). В поле Filter types (Типы фильтров) введите Person. В раскрывающемся списке Compare With Baseline (Сравнить с базовым показателем) выберите значение Snapshot #1. В списке Types (Типы) выберите значение List<Person>. Затем перейдите на вкладку Referenced Types (Ссылочные типы) и разверните иерархию объектов (рис. 4.14).

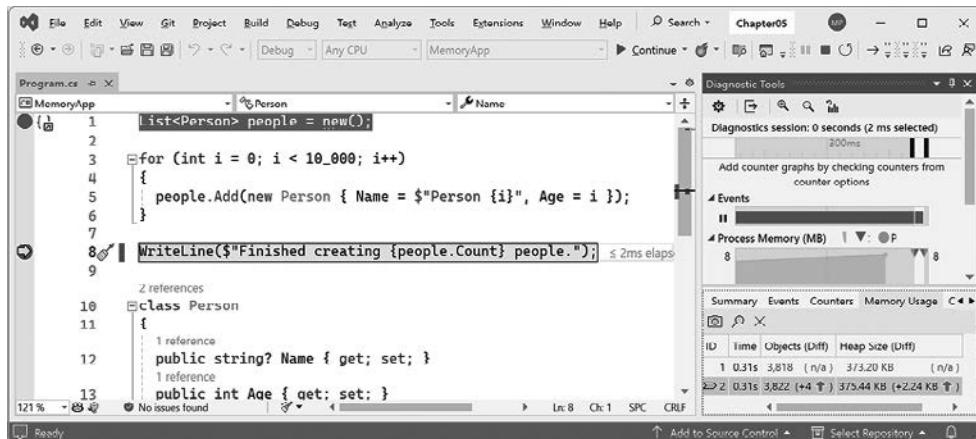


Рис. 4.13. Несколько снимков, отражающих рост затрат памяти

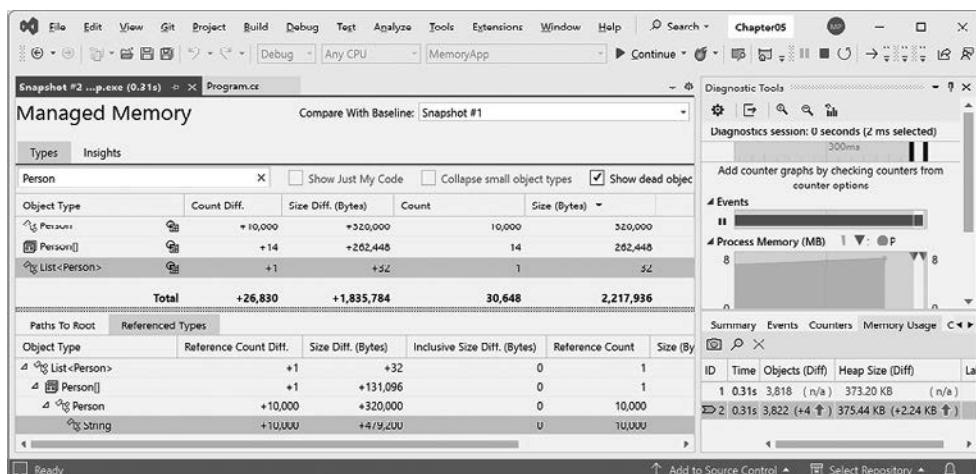


Рис. 4.14. Сравнение двух моментальных снимков

Обратите внимание на следующие детали.

- Коллекция `List<Person>` основана на массиве `Person[]`.
- В памяти находятся десять тысяч экземпляров класса `Person`, каждый из которых ссылается на строковое значение — всего десять тысяч строк.

В реальных сценариях оптимизация использования памяти может включать выбор более эффективных структур данных, сокращение области видимости и времени жизни объектов и использование стратегий кэширования, если это оправданно.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 4.1. Дополнительные онлайн-материалы

- Познакомьтесь с некоторыми из последних улучшений в отладчике Visual Studio на сайте [devblogs.microsoft.com/visualstudio/in-the-debuggers-spotlight-a-year-in-review/](http://devblogs.microsoft.com/visualstudio/in-the-debuggers-spotlight-a-year-in-review/).
- Полная документация по инструментам профилирования и методам работы с ними доступна по адресу [learn.microsoft.com/ru-ru/visualstudio/profiling/](http://learn.microsoft.com/ru-ru/visualstudio/profiling/).
- Обзор возможностей инструментов профилирования Visual Studio опубликован на странице [learn.microsoft.com/ru-ru/visualstudio/profiling/profiling-feature-tour](http://learn.microsoft.com/ru-ru/visualstudio/profiling/profiling-feature-tour).

### Упражнение 4.2. Практические задания

В главе 15 будет рассказано о контейнеризации. Подробнее о том, как выполнять отладку контейнеров, можно прочитать на сайте [devblogs.microsoft.com/dotnet/debugging-dotnet-containers-with-visual-studio-code-docker-tools/](http://devblogs.microsoft.com/dotnet/debugging-dotnet-containers-with-visual-studio-code-docker-tools/).

Для анализа использования памяти также можно воспользоваться инструментом .NET Object Allocation. Его описание представлено по адресу [learn.microsoft.com/ru-ru/visualstudio/profiling/dotnet-alloc-tool](http://learn.microsoft.com/ru-ru/visualstudio/profiling/dotnet-alloc-tool).

### Упражнение 4.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Какие приемы и стратегии считаются наиболее полезными при отладке?
2. Чем отличаются панели Locals (Локальные переменные) и Watch (Отслеживаемые значения) в процессе отладки?
3. Как управлять содержимым панелей отладки, например Locals (Локальные переменные)?
4. По умолчанию строковые значения на панелях отладки отображаются в кавычках. Как скрыть эти кавычки?
5. Какие инструменты в Visual Studio и решениях JetBrains позволяют анализировать производительность приложения и использование памяти?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 4.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-4---debugging-and-memory-troubleshooting](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-4---debugging-and-memory-troubleshooting).

## Резюме

В этой главе вы узнали:

- какие бывают стратегии отладки;
- как выполнять интерактивную отладку кода в Visual Studio;
- как определить собственные типы, чтобы управлять их отображением в окнах отладки;
- как отлаживать тестовые проекты;
- как устроено управление памятью, указателями и ресурсами в .NET;
- как анализировать использование памяти с помощью средств Visual Studio.

В следующей главе речь пойдет о трассировке, метриках и журнализации — инструментах, обеспечивающих наблюдаемость за приложением.

# 5

## Ведение журналов, трассировка и сбор метрик для обеспечения наблюдаемости

Эта глава посвящена журналированию, трассировке и мониторингу с использованием метрик — трем ключевым подходам, которые часто называют столпами наблюдаемости.

*Журналирование* — это запись событий, происходящих в системе. С помощью журналов разработчики получают детальную информацию о поведении системы и могут быстро понять, что именно пошло не так и как это исправить. Кроме того, журналы позволяют отслеживать действия пользователей, что помогает при диагностике и устранении проблем. Также они служат историческим источником данных, необходимым для аудита безопасности и выполнения нормативных требований.

*Трассировка* позволяет проследить путь запроса через компоненты системы. Благодаря ей проще понять, где конкретно и почему возникла проблема, что облегчает ее диагностику и исправление.

*Метрики* — числовые показатели, отражающие производительность и поведение системы. Регулярный сбор и анализ метрик позволяет выявлять тренды и закономерности, указывающие на проблемные участки или возможности для оптимизации. Метрики также дают возможность прогнозировать будущие нагрузки и заранее планировать ресурсы для масштабирования системы.

### Журналирование и трассировка в .NET

Эффективное журналирование и наблюдаемость в проектах на .NET помогают быстро диагностировать проблемы, лучше понимать поведение системы и поддерживать стабильность приложений.

Ниже перечислено несколько важных стратегий, которые помогут вам сделать журналирование и наблюдаемость в .NET-приложениях более эффективными.

- **Используйте структурированные журналы.** В отличие от обычных текстовых сообщений, структурированные журналы используют стандартный формат, например JSON. Благодаря этому их проще анализировать, фильтровать и от-

правлять в различные системы. Такой подход позволяет добавлять к каждому сообщению контекст: идентификатор пользователя, информацию о сеансе, данные об окружении и другие параметры. Один из популярных инструментов — Serilog, который предлагает гибкие возможности конфигурирования и поддержку различных приемников (*sinks*). Он позволяет направлять журналы в файлы, базы данных или облачные сервисы.

- **Внедрите централизованное журналирование.** Журналы из всех источников стоит собирать в единой системе. Так упрощаются поиск, анализ и выявление закономерностей. Serilog можно настроить так, чтобы он отправлял журналы в разные хранилища — например, в Elasticsearch, который подходит для хранения и поиска больших объемов данных в реальном времени. Для этого используется пакет `Serilog.Sinks.Elasticsearch`. Централизованный подход особенно полезен в распределенных системах, где важно видеть общую картину и отслеживать взаимосвязанные события между компонентами.
- **Используйте идентификатор корреляции для отслеживания запросов.** Назначьте уникальный идентификатор каждому запросу или транзакции, проходящей через систему. Этот идентификатор должен передаваться между компонентами или микросервисами и включаться во все связанные сообщения журнала. Благодаря этому можно проследить полный путь запроса и быстрее выявить проблемные участки в распределенной системе. Качественно настроенные механизмы журналирования автоматически добавляют идентификатор корреляции в каждую запись.
- **Используйте платформы наблюдаемости.** Такие решения, как OpenTelemetry, объединяют трассировки, метрики и журналы, предоставляя целостное представление о работе системы. С помощью этих платформ можно создавать информационные панели, отображающие текущее состояние системы, *ключевые показатели эффективности (KPI)* и важные журнальные события. Панели должны быть адаптированы под разные роли: разработчикам потребуется детальная техническая информация, а руководителям — обобщенные данные по основным метрикам и стабильности работы.
- **Настройте оповещения и мониторинг.** Используйте журналы и метрики для настройки оповещений, основанных на конкретных событиях, повышенной частоте ошибок или превышении пороговых значений производительности. Эффективная система оповещений помогает обнаружить проблему до того, как она затронет пользователей, и дает возможность принять меры заранее.
- **Обучайте команду.** Разработчики должны владеть базовыми принципами журналирования: что именно следует регистрировать, как использовать уровни журналов и почему важно сохранять единый формат сообщений. Следует избегать включения в журналы конфиденциальной информации и *персональных данных (РД)*. Формируйте культуру, в которой наблюдаемость воспринимается как неотъемлемая часть разработки. Так вы сможете интегрировать журналирование и мониторинг в процесс с самого начала, а не рассматривать их как временное решение «на потом».

Применение этих стратегий заметно повышает качество журналирования и наблюдаемости в проектах на .NET. Они способствуют более быстрому выявлению и устранению проблем, улучшают надежность системы и помогают глубже понимать поведение и производительность приложения.

## Интерфейс ILogger в .NET

Интерфейс `ILogger` — основа системы журналирования в .NET. Он входит в пространство имен `Microsoft.Extensions.Logging` и представляет собой обобщенный, расширяемый интерфейс, который можно использовать в приложениях разных типов.

Понимание принципов работы `ILogger` помогает внедрять эффективные практики журналирования в проектах на .NET.

Рассмотрим ключевые моменты, которые .NET-разработчику стоит знать о `ILogger`.

- **Абстракция над поставщиками журналирования.** Интерфейс `ILogger` обеспечивает единый уровень абстракции над различными системами журналирования. Разработчики могут писать код без привязки к конкретному инструменту, например Serilog. Фактическую реализацию можно заменить без изменения логики приложения, что повышает гибкость и упрощает сопровождение.
- **Пользовательские поставщики.** При необходимости возможности `ILogger` можно расширить, реализовав собственный поставщик журналов. Такой подход пригодится для записи данных в нестандартные хранилища, интеграции со сторонними сервисами или добавления специфической логики обработки сообщений.
- **Уровни важности сообщений.** Интерфейс `ILogger` поддерживает несколько уровней журналирования, позволяющих классифицировать сообщения по степени важности. Среди них: `Trace`, `Debug`, `Information`, `Warning`, `Error` и `Critical`. Такая градация помогает фильтровать сообщения в зависимости от нужного уровня детализации и играет важную роль при диагностике и анализе поведения приложения. Важно тщательно подбирать уровни, чтобы журналы оставались информативными, но не перегруженными лишней информацией.
- **Структурированное журналирование.** Интерфейс `ILogger` поддерживает запись сообщений с использованием заполнителей и передачей значений отдельно. При таком подходе структура сообщения остается постоянной, что упрощает анализ и поиск в системах журналирования. Структурированное журналирование более эффективно по сравнению с обычной интерполяцией строк: оно не только улучшает читаемость, но и повышает качество работы с журналами в сторонних сервисах.
- **Области журналирования.** Интерфейс `ILogger` поддерживает создание областей (scopes), позволяя группировать логически связанные операции. Это особенно полезно при привязке журналов к конкретному запросу, транзакции или рабочему процессу. С помощью областей можно легче отслеживать события, относящиеся к одной и той же операции в разных частях системы.

- **Интеграция с внедрением зависимостей (Dependency Injection).** Интерфейс `ILogger` легко встраивается в систему внедрения зависимостей .NET. Можно внедрить  `ILogger<T>` в любой класс, где `T` — это тип самого класса. Такая интеграция обеспечивает контекстный журнал, автоматически привязанный к исходному типу.
- **Настройка во время эксплуатации.** Поведение журналирования можно настраивать в конфигурационных файлах приложения. Благодаря этой возможности разработчики и системные администраторы могут управлять уровнями журналирования и другими параметрами `ILogger` без необходимости вносить изменения в исходный код. Ниже приведен пример такой конфигурации:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.AspNetCore": "Error",
      "System": "Warning"
    },
    "Console": {
      "IncludeScopes": true,
      "FormatterName": "simple",
      "FormatterOptions": {
        "TimestampFormat": "yyyy-MM-dd HH:mm:ss.fff",
        "UseUtcTimestamp": true
      }
    }
  }
}
```



При приведенной выше конфигурации система `ILogger` будет работать следующим образом.

- По умолчанию будут записываться сообщения уровня `Information` и выше. Для отдельных пространств имен заданы более строгие уровни: `Warning` для `Microsoft` и `System`, а также `Error` для `Microsoft.AspNetCore`.
- Вывод в консоль будет включать сообщения с активными областями (`scopes`), отформатированные с помощью простого форматера. Области позволяют добавлять контекст — например, идентификаторы запросов или сведения о пользователе.
- Временные метки будут включены в форматированном виде: `yyyy-MM-dd HH:mm:ss.fff`, с указанием времени в UTC. Такой формат содержит полную дату, время и миллисекунды.

Такая настройка обеспечивает разумный компромисс между получением необходимой информации для устранения неполадок (путем записи в журнал сообщений уровня `Information` и выше) и избеганием избыточных данных (ограничив журналирование для `Microsoft` и `System` уровнями `Warning` или `Error`): вы получаете достаточно данных для отладки и анализа, при этом избегая излишней детализации. Форматированный вывод в консоль улучшает читаемость журналов и обеспечивает согласованность временных меток.

- Влияние на производительность.** Несмотря на то что `ILogger` спроектирован с учетом эффективности, важно учитывать нагрузку от журналирования в чувствительных к производительности участках. Избыточная детализация или чрезмерная частота сообщений могут негативно повлиять на работу приложения.

Хорошее знание принципов работы `ILogger` — основа для создания надежных, поддерживаемых и наблюдаемых .NET-приложений. Интерфейс не только играет важную роль в отладке и мониторинге, но и обеспечивает гибкий, масштабируемый подход к реализации требований журналирования и диагностики.

## Как вести журнал с помощью `ILogger`

Абстракции и реализации `ILogger` разделены на отдельные пакеты, как описано в табл. 5.1.

**Таблица 5.1.** Пакеты журналирования и их назначение

Пакет	Описание
<code>Microsoft.Extensions.Logging.Abstractions</code>	Предоставляет абстракции для журналирования. Интерфейсы, определенные в этом пакете, реализуются классами других пакетов журналирования
<code>Microsoft.Extensions.Logging</code>	Содержит абстрактную реализацию, используемую основными встроенными реализациями, такими как консоль, журнал событий и отладочный журнал
<code>Microsoft.Extensions.Logging.Console</code>	Реализация поставщика ведения журналов для консоли. Зависит от двух предыдущих пакетов, поэтому вам не нужно ссылаться на них напрямую
<code>Microsoft.Extensions.Logging.Debug</code>	Реализация поставщика ведения журналов для вывода в <code>Debug</code> . Записывает сообщения с помощью метода <code>System.Diagnostics.Debug.WriteLine()</code>

Рассмотрим базовый пример журналирования в консольном приложении.

- В редакторе кода создайте проект `Console App` (Консольное приложение) / `console`, присвойте ему имя `LoggerApp` и добавьте в решение `Chapter05`.
- В файле `LoggerApp.csproj` проекта настройте обработку предупреждений как ошибок и добавьте ссылку на пакет, реализующий поставщика журналирования для консоли:

```
<ItemGroup>
  <PackageReference Include="Microsoft.Extensions.Logging.Console"
    Version="8.0.0" />
</ItemGroup>
```



Проверить наличие последней версии пакета `Microsoft.Extensions.Hosting` можно по адресу [www.nuget.org/packages/Microsoft.Extensions.Hosting/](http://www.nuget.org/packages/Microsoft.Extensions.Hosting/).

3. В файле `Program.cs` удалите все содержимое, создайте фабрику компонентов для ведения журнала с реализацией поставщика для консоли, затем создайте компонент для журналирования и выведите сообщения в консоль:

```
// Подключение ILoggerFactory, LoggerFactory, ILogger
using Microsoft.Extensions.Logging;

using ILoggerFactory factory =
    LoggerFactory.Create(builder =>
{
    // Этот метод расширения доступен только при подключении
    // пакета Microsoft.Extensions.Logging.Console
    builder.AddConsole();

    // По желанию можно добавить других поставщиков, например:
    // AddDebug, AddEventLog, AddSerilog, AddLog4Net.
    // Эти методы доступны только при подключении
    // соответствующих пакетов.

    // Также можно добавить конфигурацию, например:
    // AddFilter, AddConfiguration
});

// Параметр универсального типа используется для указания
// категорий сообщений журнала
ILogger logger = factory.CreateLogger<Program>();

string[] messageTemplates =
{
    "Logging is {Description}", // messageTemplates[0]
    "Product ID: {ProductId}, Product Name: {ProductName}"
};

logger.LogInformation(
    // Параметр message назван неудачно. Это НЕ сообщение.
    // Это строка формата шаблона, которая может содержать заполнители.
    message: messageTemplates[0],
    args: "cool!");

logger.LogWarning(messageTemplates[1],
    // Параметр args – это массив объектов, передаваемых для параметров.
    1, "Chai");
```



**Дельный совет:** хотя параметр называется `message`, на самом деле это шаблон сообщения. Если обращаться с ним как с обычным сообщением и каждый раз передавать разные строковые значения, то при каждом вызове метода будет происходить выделение памяти для хранения строки. Такой подход может привести к лишним затратам памяти и снижению производительности. Чтобы избежать указанных проблем, определите ограниченное количество строковых шаблонов сообщений и передавайте различные значения через параметр `args`.

4. Запустите проект `LoggerApp` без отладки.
5. Результат приведен ниже и показан на рис. 5.1:

```
info: Program[0]
      Logging is cool!
warn: Program[0]
      Product ID: 1, Product Name: Chai
```



The screenshot shows the Microsoft Visual Studio Debug Console window. It displays the following log entries:

```
Microsoft Visual Studio Debug Console
info: Program[0]
      Logging is cool!
warn: Program[0]
      Product ID: 1, Product Name: Chai
C:\tools-skills-net8\Chapter04\LoggerApp\bin\Debug\net8.0\LoggerApp.exe (process 17120) exited with code 0.
```

**Рис. 5.1.** Запись информации и предупреждений в консоль



**Дельный совет:** если в приложении используется внедрение зависимостей (DI), получайте объекты `ILoggerFactory` и `ILogger` из DI-контейнера, а не создавайте их вручную.



**В копилку знаний:** подробнее о журналировании с помощью `ILogger` можно узнать на сайте [learn.microsoft.com/ru-ru/dotnet/core/extensions/logging](https://learn.microsoft.com/ru-ru/dotnet/core/extensions/logging).

## Создание веб-сервиса для ведения журнала

Создадим минимальный проект веб-сервиса, который позже можно будет использовать для записи журналов.

1. В редакторе кода создайте проект веб-API и добавьте его в решение `Chapter05`, указав следующие настройки:
  - выберите шаблон проекта ASP.NET Core Web API (Веб-API ASP.NET Core) / `webapi`;
  - в поле `Solution name` (Имя решения) укажите значение `Chapter05`;
  - в поле `Project name` (Имя проекта) укажите значение `Northwind.WebApi`;
  - в раскрывающемся списке `Authentication type` (Тип проверки подлинности) выберите пункт `None` (Отсутствует);
  - установите флажок `Configure for HTTPS` (Настроить для HTTPS);
  - снимите флажок `Enable Docker` (Включить поддержку контейнера);
  - установите флажок `Enable OpenAPI support` (Включить поддержку OpenAPI);
  - снимите флажок `Do not use top-level statements` (Не использовать операторы верхнего уровня);

- снимите флажок Use controllers (Использовать контроллеры);
  - снимите флажок Enlist in .NET Aspire orchestration (Включить в список проект оркестрации .NET Aspire).
2. В файле проекта настройте обработку предупреждений как ошибок и добавьте ссылку на проект контекста данных, созданный в главе 1:
- ```
<Project Sdk="Microsoft.NET.Sdk.Web">

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

<ItemGroup>
    <ProjectReference Include=
        "..\..\Chapter01\Northwind.DataContext\Northwind.DataContext.csproj" />
</ItemGroup>

<ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="8.0.4" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.6.1" />
</ItemGroup>

</Project>
```
3. В терминале выполните сборку проекта Northwind.WebApi, чтобы восстановить пакеты и скомпилировать библиотеку классов:

```
dotnet build
```



**Внимание!** Если вы используете Visual Studio и ссылаетесь на проект, находящийся за пределами текущего решения, при сборке через меню Build (Сборка) может возникнуть следующая ошибка:

NU1105 Unable to find project information for 'C:\tools-killsnets8\Chapter01\Northwind.DataContext\Northwind.DataContext.csproj'. If you are using Visual Studio, this may be because the project is unloaded or not part of the current solution.

В этом случае выполните сборку с помощью команды `dotnet build` в терминале. Чтобы открыть терминал, щелкните правой кнопкой мыши по проекту на панели Solution Explorer (Обозреватель решений) и выберите в контекстном меню пункт Open in Terminal (Открыть в терминале).

После того как проект хотя бы один раз собран в терминале, можно использовать встроенные средства сборки Visual Studio для последующих запусков.

4. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5051 для протокола HTTPS и 5052 — для HTTP:

```
"profiles": {  
    ...  
    "https": {  
        "commandName": "Project",  
        "dotnetRunMessages": true,  
        "launchBrowser": true,  
        "launchUrl": "swagger",  
        "applicationUrl": "https://localhost:50051;http://localhost:50052",  
        "environmentVariables": {  
            "ASPNETCORE_ENVIRONMENT": "Development"  
        }  
    }  
}
```



Visual Studio прочитает файл `launchSettings.json` и автоматически запустит браузер, если параметру `launchBrowser` присвоено значение `true`. После этого произойдет переход по адресу, указанному в `applicationUrl` и `launchUrl`.

- В среде Visual Studio Code и при использовании команды `dotnet run` автоматический запуск браузера не происходит, поэтому нужно вручную открыть браузер и перейти по адресу `http://localhost:50051/swagger`.
5. В файле `Program.cs` удалите выражения, связанные с сервисом погоды, и замените их на операторы импорта пространства имен и добавления класса `NorthwindContext` в коллекцию настроенных сервисов, как показано в выделенном фрагменте кода ниже:

```
using Northwind.EntityModels; // Подключение метода AddNorthwindContext  
  
var builder = WebApplication.CreateBuilder(args);  
  
// Добавление сервисов в контейнер зависимостей.  
// Подробнее о настройке Swagger и OpenAPI – на сайте  
// aka.ms/aspnetcore/swashbuckle.  
builder.Services.AddEndpointsApiExplorer();  
builder.Services.AddSwaggerGen();  
  
builder.Services.AddNorthwindContext();  
  
var app = builder.Build();  
  
// Настройка конвейера обработки HTTP-запросов  
if (app.Environment.IsDevelopment())  
{  
    app.UseSwagger();  
    app.UseSwaggerUI();  
}  
  
app.UseHttpsRedirection();  
  
app.Run();
```

6. Создайте файл класса `WebApplicationExtensions.cs`.



**Дельный совет:** чтобы не перегружать файл `Program.cs` сотнями строк кода, определяйте методы расширения для часто используемых типов в минимальных API, таких как `WebApplication` и `IServiceCollection`.

7. В файле `WebApplicationExtensions.cs` импортируйте пространства имен, необходимые для управления результатами HTTP-запросов, привязки параметров к зависимым сервисам и работы с моделями сущностей `Northwind`. Затем определите метод расширения для класса `WebApplication`, чтобы настроить обработку всех HTTP-запросов `GET`, перечисленных в таблице нашего API:

```
using Microsoft.AspNetCore.Http.HttpResults; // Подключение класса Results
using Microsoft.AspNetCore.Mvc; // Подключение атрибута [FromServices]
                                // и других связанных возможностей.
using Northwind.EntityModels; // Подключение классов NorthwindContext и Product
namespace Packt.Extensions;

public static class WebApplicationExtensions
{
    public static WebApplication MapGets(this WebApplication app,
        int pageSize = 10)
    {
        app.MapGet("/", () => "Hello World!")
            .ExcludeFromDescription();

        app.MapGet("api/products",
            [FromServices] NorthwindContext db,
            [FromQuery] int? page) =>
        db.Products?
            .Where(p => p.UnitsInStock > 0 && !p.Discontinued)
            .OrderBy(product => product.ProductId)
            .Skip((page ?? 1) - 1) * pageSize)
            .Take(pageSize)
        )
        .WithName("GetProducts")
        .WithOpenApi(operation =>
        {
            operation.Description =
                "Get products with UnitsInStock > 0 and Discontinued = false.";
            operation.Summary = "Get in-stock products that are not discontinued.";
            return operation;
        })
        .Produces<Product[]>(StatusCodes.Status200OK);

        app.MapGet("api/products/outofstock",
            [FromServices] NorthwindContext db) => db.Products?
                .Where(p => p.UnitsInStock == 0 && !p.Discontinued)
        )
        .WithName("GetProductsOutOfStock")
        .WithOpenApi()
        .Produces<Product[]>(StatusCodes.Status200OK);
    }
}
```

```

app.MapGet("api/products/discontinued",
    ([FromServices] NorthwindContext db) =>
        db.Products?.Where(product => product.Discontinued)
    )
    .WithName("GetProductsDiscontinued")
    .WithOpenApi()
    .Produces<Product[]>(StatusCodes.Status200OK);

app.MapGet("api/products/{id:int}",
    async Task<Results<Ok<Product>, NotFound>> (
        [FromServices] NorthwindContext db,
        [FromRoute] int id) =>
    {
        if (db.Products is null)
        {
            return TypedResults.NotFound();
        }
        else
        {
            return await db.Products.FindAsync(id) is Product product ?
                TypedResults.Ok(product) : TypedResults.NotFound();
        }
    })
    .WithName("GetProductById")
    .WithOpenApi()
    .Produces<Product>(StatusCodes.Status200OK)
    .Produces(StatusCodes.Status404NotFound);

app.MapGet("api/products/{name}", (
    [FromServices] NorthwindContext db,
    [FromRoute] string name) =>
    db.Products?.Where(p => p.ProductName.Contains(name)))
    .WithName("GetProductsByName")
    .WithOpenApi()
    .Produces<Product[]>(StatusCodes.Status200OK);

return app;
}
}

```

8. В файле `Program.cs` импортируйте пространство имен, чтобы использовать методы расширения, определенные ранее:

```
using Packt.Extensions; // Подключение метода расширения MapGets
```

9. В этом же файле перед вызовом метода `app.Run()` вызовите свой метод расширения для отображения GET-запросов. Обратите внимание: при запросе всех товаров можно переопределить стандартный размер страницы, равный десяти сущностям:

```
app.MapGets(); // Размер страницы по умолчанию: 10
```

```
app.Run();
```

## Тестирование базовой функциональности веб-сервиса

Теперь можно проверить, как работает веб-сервис на базовом уровне.

1. Запустите проект `Northwind.WebApi` без отладки.
2. В браузере откроется страница Swagger — она используется для документирования и тестирования API.
3. Разверните раздел `GET /api/products/outofstock`.
4. Нажмите кнопку `Try it out` (Попробовать).
5. Нажмите кнопку `Execute` (Выполнить) и обратите внимание на результат — должен отобразиться один товар, отсутствующий на складе, как показано в следующем выводе:

```
[  
 {  
     "productId": 31,  
     "productName": "Gorgonzola Telino",  
     "unitPrice": 12.5,  
     "unitsInStock": 0,  
     "discontinued": false,  
     "categoryId": 4,  
     "category": null  
 }  
]
```

6. Закройте браузер и остановите веб-сервер.

Теперь можно переходить к добавлению мониторинга в эти проекты.

## Мониторинг с использованием метрик в .NET

Метрики — это фундаментальная концепция мониторинга приложений и сервисов в .NET. Они тесно связаны с оповещениями, поскольку именно на основе метрик формируются сигналы, которые указывают на необходимость внимания к системе.

Оповещения — это уведомления о событиях, выходящих за пределы ожидаемого поведения. Правильно настроенные оповещения помогают обнаруживать и устранять проблемы еще до того, как они перерастут в серьезные сбои. Такой подход снижает время простоев и повышает надежность решений.

### Метрики и оповещения: основные понятия

Перед тем как углубиться в практику, кратко рассмотрим, что представляют собой метрики и оповещения, а также как они помогают повышать устойчивость и наблюдаемость .NET-приложений.

## Метрики

Метрики — это количественные показатели, которые используются для отслеживания и анализа производительности, состояния и поведения приложения или системы. В .NET часто применяются следующие категории метрик.

- Производительность — использование процессора, потребление памяти, время обработки запросов и другие характеристики на уровне системы.
- Приложение — количество запросов в секунду, частота ошибок, время отклика и другие показатели поведения приложения.
- Пользовательские — любые специфические для предметной области измерения, например количество обработанных товаров, оформленных заказов или вошедших в систему пользователей.

Метрики собираются на протяжении определенного времени и дают представление о том, как обычно работает система. Они помогают выявлять закономерности, находить проблемные места и улучшать производительность.

В .NET-приложениях можно использовать различные инструменты для сбора и анализа метрик.

- *System.Diagnostics* — встроенная библиотека .NET для работы со счетчиками производительности.
- *Prometheus* — система мониторинга с открытым исходным кодом, часто используется в .NET через экспортёры.
- *Grafana* — инструмент визуализации, поддерживающий разные источники данных, включая Prometheus.
- *Application Insights* — облачный сервис мониторинга Azure, способный собирать и анализировать метрики .NET-приложений. В книге он не рассматривается, поскольку требует инфраструктуру Azure.

## Оповещения

Оповещения — это уведомления, которые срабатывают при достижении определенных условий или пороговых значений, заданных на основе метрик. Они информируют, когда в приложении или системе происходит что-то необычное или потенциально критичное. Благодаря оповещениям можно быстрее реагировать на проблемы, сокращая время простоя и снижая влияние на пользователей.

Вот распространенные сценарии для оповещений.

- **Нарушение пороговых значений** — например, использование процессора превышает 80 % в течение более 5 минут.
- **Частота ошибок** — если количество ответов с кодом HTTP 500 превышает заданный процент.
- **Пользовательские условия** — любые специфические для приложения метрики, например резкое снижение количества заказов или рост времени обработки задач.

Распространенные инструменты для настройки оповещений.

- **Prometheus Alertmanager** — расширяет Prometheus возможностью определять правила оповещений и отправлять уведомления.
- **Azure Monitor и Application Insights** — службы Azure, предлагающие широкие возможности для оповещений на основе собранных метрик приложений .NET. Поддерживается создание оповещений на основе собранных метрик.

## Сценарии для метрик и предупреждений

Работа метрик и оповещений строится на тесной взаимосвязи.

1. Метрики непрерывно собираются из приложения.
2. Собранные данные анализируются, чтобы отслеживать поведение и производительность системы.
3. На основе этих метрик настраиваются условия оповещений — например, пороговые значения.
4. Когда заданное условие выполняется (например, превышен лимит загрузки процессора), срабатывает оповещение, которое доставляется через электронную почту, SMS или другие каналы уведомлений.

Представим, что у вас есть веб-приложение на платформе .NET. В таком случае можно настроить следующие метрики и оповещения.

- **Метрика:** среднее время ответа на HTTP-запросы.
- **Оповещение:** если среднее время ответа превышает 2 секунды в течение 5 минут.
- **Метрика:** количество активных пользователей.
- **Оповещение:** если количество активных пользователей опускается ниже заданного порога — это может свидетельствовать о сбоях в доступности.
- **Метрика:** частота ошибок при выполнении HTTP-запросов.
- **Оповещение:** если уровень ошибок превышает 5 % — возможен сбой в работе приложения.

Постоянный мониторинг этих показателей вместе с грамотно настроенными оповещениями помогает поддерживать бесперебойную работу приложения и своевременно реагировать на возникающие проблемы. Метрики дают представление о текущем состоянии, а оповещения обеспечивают мгновенную реакцию в случае отклонений от нормы.

## Реализация метрик

В .NET для сбора и записи метрик используются классы `Meter`, `Counter<T>` и `Histogram<T>` из пространства имен `System.Diagnostics.Metrics`. Эти типы предназначены для мониторинга и диагностики в современных .NET-приложениях.



**Внимание!** Избегайте использования API System.Diagnostics.PerformanceCounter. Этот интерфейс считается устаревшим: он работает только в Windows и представляет собой управляемую оболочку вокруг системы счетчиков производительности операционной системы. Современный и рекомендуемый подход — System.Diagnostics.Metrics. Эти API кросс-платформенные, совместимы с проектом OpenTelemetry и поддерживают инструмент командной строки .NET Counters, который будет рассмотрен в этом разделе.

Класс Meter представляет собой именованную группу инструментов метрик (например, счетчиков и гистограмм), логически объединенных между собой. Он выступает в роли фабрики, создающей эти инструменты, и предоставляет для них область видимости. Каждый экземпляр Meter имеет имя и версию, по которым его можно идентифицировать.

Класс Counter<T> используется для отслеживания количества событий или операций. Это накапливаемая метрика, предназначенная для ситуаций, когда значения только увеличиваются — например, общее число полученных запросов или количество ошибок. Такие счетчики полезны при анализе активности системы за определенный период.

Создание счетчика начинается с вызова соответствующего метода у экземпляра Meter, а увеличение значения выполняется через метод Add:

```
// Создание экземпляра Meter
Meter meter = new("Northwind.WebApi.Metrics", "1.0");

// Создание счетчика с помощью Meter
Counter<int> counter = meter.CreateCounter<int>(
    "request_count", description: "Number of requests received.");

// Увеличение значения счетчика
counter.Add(1);
```

Класс Histogram<T> используется для записи распределения значений — например, длительности обработки запросов или размера полезной нагрузки. Такой инструмент помогает анализировать диапазон и характер колебаний в наборе измерений. Экземпляр Histogram<T> создается через объект Meter, а значения записываются с помощью метода Record.



**Дельный совет:** в предыдущем примере был создан новый экземпляр класса Meter. Однако в проектах, где используется внедрение зависимостей (DI), таких как ASP.NET Core, рекомендуется использовать зарегистрированный экземпляр IMeterFactory.

Подробные причины этого подхода обсуждаются в главах 10 и 17. В проектах на .NET 8 и новее IMeterFactory регистрируется автоматически. В других случаях его можно добавить вручную в коллекцию IServiceCollection, вызвав метод AddMetrics.

Теперь рассмотрим практический пример добавления счетчика и гистограммы в проект веб-сервиса ASP.NET Core.

## Добавление метрик в проект ASP.NET Core

Мы добавим в веб-сервис несколько конечных точек для управления и обновления метрик. В этом примере будут использоваться счетчик и гистограмма для отслеживания общего количества запросов и времени их обработки.

1. В файле `Northwind.WebApi.csproj` проекта добавьте ссылку на пакет, необходимый для работы со счетчиками метрик.

```
<PackageReference Version="8.0.0"
  Include="System.Diagnostics.PerformanceCounter" />
```

2. Создайте файл класса `MetricsService.cs`.
3. В созданном файле импортируйте пространство имен `System.Diagnostics.Metrics` и определите класс, отвечающий за настройку счетчиков и увеличение значения количества полученных запросов.

```
using System.Diagnostics.Metrics; // Подключение классов Meter,
                                // Counter и Histogram

namespace Northwind.WebApi;

public class MetricsService
{
    private readonly Meter _meter;
    private readonly Counter<int> _requestCounter;
    private readonly Histogram<double> _requestDuration;

    public int RequestCount { get; private set; }
    public List<double> RequestDurations { get; private set; } = [];

    public MetricsService(IMeterFactory meterFactory)
    {
        _meter = meterFactory.Create("Northwind.WebApi.Metrics", "1.0");

        _requestCounter = _meter.CreateCounter<int>(name: "request_count",
            description: "Number of requests received.");

        _requestDuration = _meter.CreateHistogram<double>(
            name: "request_duration", unit: "ms",
            description: "Request duration in milliseconds.");
    }

    public void IncrementRequestCount()
    {
        _requestCounter.Add(1);
        RequestCount++;
    }
}
```

```
public void RecordRequestDuration(double duration)
{
    _requestDuration.Record(duration);
    RequestDurations.Add(duration);
}
```

4. В файле `Program.cs` импортируйте пространство имен, содержащее определение `MetricsService`.

```
using Northwind.WebApi; // Подключение класса MetricsService
```

5. В том же файле сразу после строки регистрации контекста данных `NorthwindContext` добавьте регистрацию сервиса `MetricsService` в контейнер зависимостей.

```
builder.Services.AddNorthwindContext();
builder.Services.AddSingleton<MetricsService>();
```

6. Создайте файл класса `MetricsMiddleware.cs`.

7. В этом файле определите промежуточный компонент, который:

- получает экземпляр зарегистрированного сервиса метрик;
- измеряет длительность обработки текущего HTTP-запроса;
- увеличивает значение счетчика количества запросов (за исключением обращений к маршруту `/api/metrics`).

```
using System.Diagnostics; // Для работы с классом Stopwatch

namespace Northwind.WebApi;

public class MetricsMiddleware
{
    private readonly RequestDelegate _next;
    private readonly MetricsService _metricsService;

    public MetricsMiddleware(RequestDelegate next, MetricsService metricsService)
    {
        _next = next;
        _metricsService = metricsService;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        Stopwatch timer = Stopwatch.StartNew();

        // Переход к следующему промежуточному модулю в конвейере
        await _next(context);

        timer.Stop();

        if (!context.Request.Path.StartsWithSegments("/api/metrics"))
        {
            _metricsService.IncrementRequestCount();
        }
    }
}
```

```
        _metricsService.RecordRequestDuration(timer.ElapsedMilliseconds);
    }
}

public static class MetricsMiddlewareExtensions
{
    public static IApplicationBuilder UseMetricsMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MetricsMiddleware>();
    }
}
```

8. В файле `Program.cs` сразу после строки с вызовом перенаправления HTTPS-запросов добавьте строку, подключающую пользовательское промежуточное ПО `MetricsMiddleware` к конвейеру HTTP-запросов.

```
app.UseHttpsRedirection();
app.UseMetricsMiddleware();
```

9. В файле `WebApplicationExtensions.cs` добавьте код, определяющий новую конечную точку по адресу `/api/metrics`. Эта точка будет возвращать значения счетчиков и гистограмм, собранных в сервисе метрик.

```
public static WebApplication MapGets(this WebApplication app,
    int pageSize = 10)
{
    app.MapGet("api/metrics", (
        [FromServices] MetricsService metricsService) => new
    {
        metricsService.RequestCount, metricsService.RequestDurations
    });

    app.MapGet("/", () => "Hello World!")
        .ExcludeFromDescription();
}
```

10. Запустите проект `Northwind.WebApi` без отладки.
11. В браузере откроется страница Swagger — интерфейс для документирования и тестирования API.
12. Разверните раздел `GET /api/metrics`, нажмите кнопку `Try it out` (Попробовать), затем кнопку `Execute` (Выполнить). Обратите внимание, что на этом этапе метрики еще не отображаются, как показано в теле ответа ниже.

```
{
    "requestCount": 0,
    "requestDurations": []
}
```

13. Разверните раздел `GET /api/products`, нажмите кнопку `Try it out` (Попробовать), затем кнопку `Execute` (Выполнить). В ответе вернется первая страница, содержащая десять товаров.

14. В поле страницы введите 3, нажмите кнопку Execute (Выполнить) — в ответе вернется третья страница из десяти товаров.
15. Разверните раздел GET /api/products/discontinued, нажмите кнопку Try it out (Попробовать), затем кнопку Execute (Выполнить) — в ответе должны отобразиться три товара с истекшим сроком действия.
16. Вернитесь в раздел GET /api/metrics, нажмите кнопку Execute (Выполнить) — теперь в ответе отображаются значения метрик, как показано в теле ответа ниже.

```
{
  "requestCount": 3,
  "requestDurations": [
    783,
    21,
    57
  ]
}
```

17. Закройте браузер и остановите веб-сервер.

## Просмотр метрик

Для анализа производительности в реальном времени можно установить и использовать инструмент .NET Counters. Ниже приведена последовательность действий.

1. Установите инструмент .NET Counters глобально:

```
dotnet tool update -g dotnet-counters
```

2. Запустите проект Northwind.WebApi без отладки и оставьте его работать.
3. Откройте Task Manager (Диспетчер задач), найдите процесс northwind, щелкните правой кнопкой мыши на его записи и в контекстном меню выберите пункт Go to details (Подробное описание). Обратите внимание на идентификатор процесса (PID), например 9552 (рис. 5.2).



**Рис. 5.2.** Поиск PID для сервиса Northwind.WebApi

4. В терминале запустите мониторинг с помощью .NET Counters, указав PID приложения:

```
dotnet-counters monitor -p 9552 --counters Northwind.WebApi.Metrics
```

5. Перейдите в интерфейс Swagger и выполните несколько вызовов к веб-сервису.
6. В терминале просмотрите обновляющиеся метрики, отображаемые в режиме реального времени, как показано в следующем выводе.

```
Press p to pause, r to resume, q to quit.  
Status: Running  
  
Name  
Current Value  
[Northwind.WebApi.Metrics]  
    request_count (Count)  
    3  
    request_duration (ms)  
        Percentile=50  
    49  
        Percentile=95  
    49  
        Percentile=99  
    49
```

7. Закройте браузер и остановите веб-сервис.

Теперь переходим к следующему шагу — интеграции всех этих техник с помощью фреймворка OpenTelemetry.

## Знакомство с OpenTelemetry

Телеметрия помогает собирать метрики и журналы прямо из кода, а затем отслеживать и анализировать полученные данные. В отличие от отладки, она обычно не влияет на выполнение программы и почти не оказывается на производительности.

В журналах фиксируются отдельные операции с пометками, были ли они завершены успешно или сопровождались ошибками. Метрики отражают численные показатели — например, сколько запросов было обработано, сколько из них активны в данный момент или сколько возникло исключений.

*OpenTelemetry (OTel)* — это фреймворк, созданный для решения задач наблюдаемости. В него входят API, SDK и инструменты, которые совместно помогают организовать сбор телеметрических данных, таких как метрики и журналы, в вашем приложении.

Поскольку в .NET уже предусмотрен собственный API для журналирования, нет необходимости отдельно изучать API OpenTelemetry. Вместо этого OpenTelemetry может работать напрямую через интерфейс `ILogger`. Для работы с OpenTelemetry потребуется установленный .NET SDK версии 6 или новее.

OpenTelemetry имеет несколько важных преимуществ при работе с проектами на .NET. Среди них – единый механизм сбора, согласованные схемы и понятная семантика телеметрических данных. Также предусмотрены API-интерфейсы для подключения внешних систем анализа, таких как Prometheus (для метрик), Grafana (для визуализации) и Jaeger (для распределенной трассировки). Эти системы называются **экспортерами**.

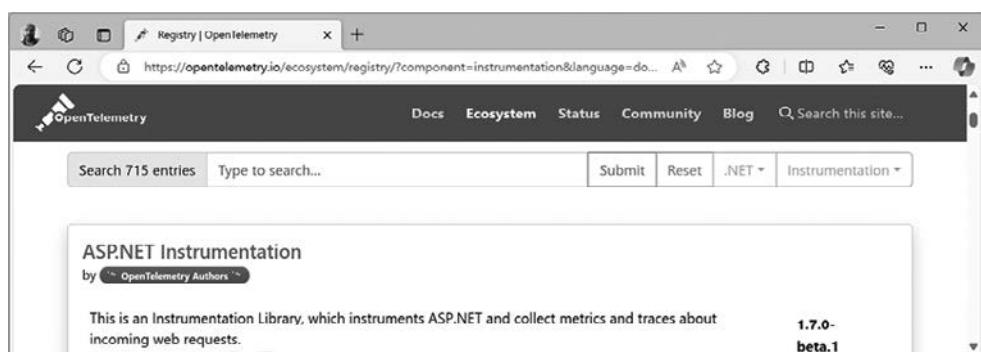
Хотя настройку можно выполнить вручную, в главе 16 мы рассмотрим, как подключить OpenTelemetry к проекту и сразу получить доступ к встроенным панелям инструментов для анализа.

*OpenTelemetry Protocol (OTLP)* – это независимый сетевой протокол, предназначенный для передачи телеметрических данных, собираемых с помощью OpenTelemetry.

## Поддерживаемые пакеты инструментов

В экосистеме OpenTelemetry доступно множество полезных компонентов — от библиотек и утилит до расширений для коллекторов. Найти их помогает *реестр OpenTelemetry Registry*, в котором удобно искать необходимые инструменты, подходящие под задачи проекта. Посмотрим, как устроен этот реестр.

1. Откройте реестр OpenTelemetry Registry для .NET-пакетов инструментов по адресу: [opentelemetry.io/ecosystem/registry/?component=instrumentation&language=dotnet](https://opentelemetry.io/ecosystem/registry/?component=instrumentation&language=dotnet).
2. В верхней части страницы указано общее количество пакетов — на момент написания книги их было 715. Однако отображаются только те, которые соответствуют выбранным фильтрам: поддерживают .NET и относятся к инструментам для трассировки (рис. 5.3).



**Рис. 5.3.** .NET-пакеты инструментов из реестра OpenTelemetry

3. В поле Type to search (Введите текст для поиска) введите `sql`, затем нажмите кнопку Submit (Подтвердить) или клавишу Enter. В результатах отобразится пакет `SqlClient Instrumentation for OpenTelemetry`.

4. В том же поле поиска введите запрос `entity`, затем нажмите кнопку Submit (Подтвердить) или клавишу Enter — отобразится пакет EntityFrameworkCore Instrumentation for OpenTelemetry .NET.
5. Теперь введите `aspnetcore`, снова нажмите кнопку Submit (Подтвердить) или клавишу Enter — отобразится пакет ASP.NET Core Instrumentation for OpenTelemetry .NET.
6. Введите запрос `grpc` и выполните поиск. Вы увидите два пакета: один предназначен только для клиентской стороны gRPC, а второй поддерживает как клиент, так и сервер.
7. При желании можно выполнить поиск по названиям инструментов и библиотек, которые вы используете в своих проектах. Например: `HttpClient`, `Redis`, `Hangfire`, `Quartz`, `MassTransit`, `Azure`, `AWS` и др.

## Сбор телеметрии в проекте ASP.NET Core

Один из самых распространенных сценариев сбора телеметрии — это проект ASP.NET Core, в котором данные сохраняются в SQL Server. В этом разделе мы продолжим работу с уже знакомыми примерами и шаг за шагом подключим OpenTelemetry.

1. Запустите проект `Northwind.WebApi` без отладки.
2. Выборочно проверьте работу конечных точек. Например, отправьте запрос на получение трех товаров, названия которых начинаются на `cha`. На рис. 5.4 показан пример результата.

```

Request URL
https://localhost:50051/api/products/cha

Server response
Code Details

200 Response body
[{"productId": 1, "productName": "Chai", "unitPrice": 2.99, "unitsInStock": 39, "discontinued": false, "categoryId": 1, "category": null}, {"productId": 2, "productName": "Chang", "unitPrice": 10, "unitsInStock": 17, "discontinued": false, "categoryId": 1, "category": null}, {"productId": 39, "productName": "Chartreuse verte", "unitPrice": 32, "unitsInStock": 69, "discontinued": false, "categoryId": 1, "category": null}]

```

**Рис. 5.4.** Результат запроса: три продукта с названиями, начинающимися на «cha»

3. Закройте браузер и завершите работу веб-сервера.
4. В проекте `Northwind.WebApi` добавьте ссылки на следующие пакеты для интеграции с OpenTelemetry:
  - пакет для хостинга .NET;
  - пакет для вывода данных в консоль;
  - пакет с инструментами для ASP.NET Core;
  - пакет клиента SQL Server;
  - пакет для Entity Framework Core.

```
<PackageReference Include="OpenTelemetry.Exporter.Console" Version="1.8.1" />
<PackageReference Include="OpenTelemetry.Extensions.Hosting" Version="1.8.1" />
<PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore"
    Version="1.8.1" />
<PackageReference Include="OpenTelemetry.Instrumentation.EntityFrameworkCore"
    Version="1.0.0-*" />
<PackageReference Include="OpenTelemetry.Instrumentation.SqlClient"
    Version="1.8.0-*" />
```



Во время работы над этой книгой пакеты инструментов для клиента SQL Server и EF Core находились на стадии бета-тестирования. Чтобы можно было использовать актуальные версии, в коде указаны формулировки с подстановочными знаками — в частности, `1.0.0-*` и `1.8.0-*`.

5. Выполните сборку проекта `Northwind.WebApi`, чтобы восстановить зависимости.
6. В файле `Program.cs` импортируйте пространства имен, необходимые для работы с OpenTelemetry.

```
using OpenTelemetry.Logs; // Подключение метода AddConsoleExporter
using OpenTelemetry.Metrics; // Подключение метода WithMetrics
using OpenTelemetry.Resources; // Подключение метода ResourceBuilder
using OpenTelemetry.Trace; // Подключение метода WithTracing
```

7. Найдите участок кода, где добавляется контекст данных Northwind, и перед вызовом `builder.Build()` добавьте инструкции, настраивающие OpenTelemetry — как для встроенной системы журналирования .NET, так и для сервисов, зарегистрированных в DI-контейнере.

```
// Добавьте и настройте OpenTelemetry для ведения журнала
// и использования сервисов.
const string serviceName = "Northwind.WebApi";

builder.Logging.AddOpenTelemetry(options =>
{
    options
        .SetResourceBuilder(ResourceBuilder.CreateDefault()
            .AddService(serviceName))
        .AddConsoleExporter();
});
```

```
builder.Services.AddOpenTelemetry()
    .ConfigureResource(resource => resource.AddService(serviceName))
    .WithTracing(tracing => tracing
        .AddAspNetCoreInstrumentation()
        .AddEntityFrameworkCoreInstrumentation()
        .AddSqlClientInstrumentation()
        .AddConsoleExporter())
    .WithMetrics(metrics => metrics
        .AddAspNetCoreInstrumentation()
        .AddConsoleExporter());
```



С такими настройками в консоль будут выводиться журналы трассировки веб-запросов, команд EF Core и SQL-клиента, а также метрики ASP.NET Core, включая данные от веб-сервера Kestrel.

## Просмотр телеметрии

Теперь можно проверить, как осуществляется сбор телеметрических данных.

1. Запустите проект `Northwind.WebApi` без отладки.
2. В терминале появится дополнительная информация, связанная с настройкой конечной точки HTTPS.

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:50051
LogRecord.Timestamp:          2024-02-06T13:53:15.800Z
LogRecord.CategoryName:       Microsoft.Hosting.Lifetime
LogRecord.Severity:           Info
LogRecord.SeverityText:       Information
LogRecord.Body:               Now listening on: {address}
LogRecord.Attributes (Key:Value):
    address: https://localhost:50051
    OriginalFormat (a.k.a Body): Now listening on: {address}
LogRecord.EventId:            14
LogRecord.EventName:          ListeningOnAddress

Resource associated with LogRecord:
service.name: Northwind.WebApi
service.instance.id: 8074ba98-0800-4db8-805b-5fd76b3affd3
telemetry.sdk.name: opentelemetry
telemetry.sdk.language: dotnet
telemetry.sdk.version: 1.7.0
```

3. При первом открытии домашней страницы Swagger в терминале будут отображаться записи о выполнении HTTP-запросов типа GET — например, к домашней странице документации Swagger.

Activity.TraceId:	d6efea25c5e02933af9c4f9cbfa42a8e
Activity.SpanId:	c20bd4cf4c41820d
Activity.TraceFlags:	Recorded

```

Activity.ActivitySourceName: Microsoft.AspNetCore
Activity.DisplayName: GET
Activity.Kind: Server
Activity.StartTime: 2024-02-06T13:53:16.8270755Z
Activity.Duration: 00:00:00.2663518
Activity.Tags:
    server.address: localhost
    server.port: 50051
    http.request.method: GET
    url.scheme: https
    url.path: /swagger/index.html
    network.protocol.version: 2
    user_agent.original: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 Safari/537.36
    http.response.status_code: 200
Resource associated with Activity:
    service.name: Northwind.WebApi
    service.instance.id: 8074ba98-0800-4db8-805b-5fd76b3affd3
    telemetry.sdk.name: opentelemetry
    telemetry.sdk.language: dotnet
    telemetry.sdk.version: 1.7.0

```

4. В терминале отображаются метрики веб-сервера *Kestrel* — в частности, количество активных HTTP-запросов. На этом этапе оно может быть равно нулю.

```

Metric Name: http.server.active_requests, Number of active HTTP server
requests., Unit: {request}, Meter: Microsoft.AspNetCore.Hosting
(2024-02-06T13:53:15.7220071Z, 2024-02-06T13:53:25.7214612Z] http.
request.method: GET url.scheme: https LongSumNonMonotonic
Value: 0

```

5. В браузере откройте Swagger и вызовите конечную точку, которая возвращает список товаров по названию.
6. В терминале выполните поиск по слову *cha*. Среди результатов будет HTTP-запрос *GET* — он соответствует вызову конечной точки, возвращающей товары с названиями, начинающимися на *cha*.

```

Activity.TraceId: dcf2799dd4b55a986bd1f899900d0fe2
Activity.SpanId: 7c799ce90ef0cc55
Activity.TraceFlags: Recorded
Activity.ActivitySourceName: Microsoft.AspNetCore
Activity.DisplayName: GET api/products/{name}
Activity.Kind: Server
Activity.StartTime: 2024-02-06T14:13:15.8030496Z
Activity.Duration: 00:00:00.7640944
Activity.Tags:
    server.address: localhost
    server.port: 50051
    http.request.method: GET
    url.scheme: https
    url.path: /api/products/cha
    network.protocol.version: 2

```

```

user_agent.original: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 Safari/537.36
http.route: api/products/{name}
http.response.status_code: 200
Resource associated with Activity:
service.name: Northwind.WebApi
service.instance.id: 53c0ed8b-a02c-4507-92ef-8bb118e614ac
telemetry.sdk.name: opentelemetry
telemetry.sdk.language: dotnet
telemetry.sdk.version: 1.7.0

```



Одна из ключевых частей в этих данных — свойство `Activity.TraceId`. В примере HTTP-запроса GET этому свойству присвоено значение `dcf2799dd4b55a986bd1f899900d0fe2`.

- Найдите в терминале сообщения, связанные с `SqlClient`. В одном из них указывается значение `Activity.TraceId: dcf2799dd4b55a986bd1f899900d0fe2`. Это означает, что трассировка охватывает сразу несколько уровней — от веб-сервиса до доступа к данным.

```

Activity.TraceId: dcf2799dd4b55a986bd1f899900d0fe2
Activity.SpanId: 5afdcdd6e4d6a297
Activity.TraceFlags: Recorded
Activity.ParentSpanId: f344a4a64bc72502
Activity.ActivitySourceName: OpenTelemetry.Instrumentation.SqlClient
Activity.DisplayName: Northwind
Activity.Kind: Client
Activity.StartTime: 2024-02-06T14:13:16.4738201Z
Activity.Duration: 00:00:00.0219486
Activity.Tags:
  db.system: mssql
  db.name: Northwind
  peer.service: .
  db.statement_type: Text
Resource associated with Activity:
service.name: Northwind.WebApi
service.instance.id: 53c0ed8b-a02c-4507-92ef-8bb118e614ac
telemetry.sdk.name: opentelemetry
telemetry.sdk.language: dotnet
telemetry.sdk.version: 1.7.0

```

- Выполните поиск по слову `Entity`. В найденных строках содержится свойство `LogRecord.TraceId` с тем же значением `dcf2799dd4b55a986bd1f899900d0fe2`. Это подтверждает, что трассировка прошла через все уровни, включая Entity Framework Core.

```

info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (33ms) [Parameters=@__name_0_rewritten='?' (Size =
40)], CommandType='Text', CommandTimeout='30']
      SELECT [p].[ProductId], [p].[CategoryId], [p].[Discontinued], [p].
      [ProductName], [p].[UnitPrice], [p].[UnitsInStock]

```

```

    FROM [Products] AS [p]
    WHERE [p].[ProductName] LIKE @_name_0_rewritten ESCAPE N'\''

LogRecord.Timestamp:      2024-02-06T14:13:16.5059590Z
LogRecord.TraceId:        dcf2799dd4b55a986bd1f899900d0fe2
LogRecord.SpanId:         5afdcdd6e4d6a297
LogRecord.TraceFlags:     Recorded
LogRecord.CategoryName:   Microsoft.EntityFrameworkCore.Database.Command
LogRecord.Severity:       Info
LogRecord.SeverityText:   Information
LogRecord.Body:           Executed DbCommand ({elapsed} ms)
[Parameters=[{parameters}], CommandType='{commandType}', CommandTimeout=
'{commandTimeout}']{newLine}{commandText}

LogRecord.Attributes (Key:Value):
elapsed:                 33
parameters:               @_name_0_rewritten='?' (Size = 40)
commandType:              Text
commandTimeout:           30
newLine:

commandText:              SELECT [p].[ProductId], [p].[CategoryId],
                           [p].[Discontinued], [p].[ProductName],
                           [p].[UnitPrice], [p].[UnitsInStock]
                           FROM [Products] AS [p]
                           WHERE [p].[ProductName] LIKE @_name_0_rewritten
                           ESCAPE N'\''

OriginalFormat (a.k.a Body): Executed DbCommand ({elapsed} ms)
[Parameters=[{parameters}], CommandType='{commandType}', CommandTimeout='{commandTimeout}']{newLine}{commandText}

LogRecord.EventId:         20101
LogRecord.EventName:       Microsoft.EntityFrameworkCore.Database.Command.
CommandExecutedResource associated with LogRecord:
service.name: Northwind.WebApi
service.instance.id: 53c0ed8b-a02c-4507-92ef-8bb118e614ac
telemetry.sdk.name: opentelemetry
telemetry.sdk.language: dotnet
telemetry.sdk.version: 1.7.0

```

## 9. Завершите работу веб-сервера.

Просматривать телеметрию прямо в консоли — не самое удобное занятие, но так вы увидите, какие именно данные перехватываются. В частности, `TraceId` помогает связать события, происходящие на различных уровнях и в разных компонентах полноценного .NET-решения. Чтобы работать с этими данными более удобно — фильтровать их, просматривать и анализировать, — лучше всего настроить экспорт в одну из специализированных систем, таких как Jaeger, Zipkin или Prometheus.



**В копилку знаний:** список систем, в которые можно экспортировать данные из OpenTelemetry, опубликован на сайте [opentelemetry.io/docs/languages/net/exporters/](https://opentelemetry.io/docs/languages/net/exporters/).

Разработчики на .NET могут также воспользоваться преимуществами нового фреймворка .NET Aspire. О том, как он помогает работать с телеметрией и не только, пойдет речь в главе 16.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 5.1. Дополнительные онлайн-материалы

Чтобы подробнее разобраться в возможностях OpenTelemetry для .NET, загляните в официальную документацию: [opentelemetry.io/docs/languages/net/](https://opentelemetry.io/docs/languages/net/).

### Упражнение 5.2. Практические задания

На GitHub-странице проекта по адресу [github.com/open-telemetry/opentelemetry-dotnet/tree/main/examples](https://github.com/open-telemetry/opentelemetry-dotnet/tree/main/examples) представлены готовые примеры — они демонстрируют, как реализуется поддержка OpenTelemetry в разных типах .NET-проектов.

Чтобы понять, как связать .NET и OpenTelemetry с такими системами, как Prometheus, Grafana и Jaeger, советуем ознакомиться со статьей на сайте Microsoft: [learn.microsoft.com/ru-ru/dotnet/core/diagnostics/observability-with-otel](https://learn.microsoft.com/ru-ru/dotnet/core/diagnostics/observability-with-otel).

### Упражнение 5.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Какие существуют подходы к оптимизации журналирования и наблюдаемости в .NET-приложениях?
2. Что важно знать .NET-разработчику об интерфейсе `ILogger` и принципах его использования?
3. Почему в методах `LogX` используют шаблоны сообщений, а не просто строки?
4. В чем разница между журналами, метриками и оповещениями?
5. Что такое OpenTelemetry и почему этот инструмент рекомендован Microsoft для проектов на платформе .NET?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 5.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-5---logging-tracing-and-metrics-for-observability](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-5---logging-tracing-and-metrics-for-observability).

## Резюме

В этой главе мы разобрались:

- как работает интерфейс `ILogger` и какую роль он играет в абстракции ведения журнала в .NET;
- как внедрять метрики и использовать их для мониторинга;
- что такое OpenTelemetry и как подключить его к проекту ASP.NET Core.

В следующей главе речь пойдет о том, как документировать код, в том числе публичные API и сервисы.

# 6

# Документирование кода, API и сервисов

В этой главе речь пойдет о том, как оформлять код, публичные *программные интерфейсы приложений* (Application Programming Interface, API) и сервисы так, чтобы в них легко разбирались не только коллеги, но и вы сами спустя некоторое время. Правильная документация упрощает сопровождение, ускоряет адаптацию новых членов команды, помогает выстраивать надежный дизайн API и существенно облегчает отладку и поддержку.

Мы также познакомимся с рядом инструментов, которые помогут создавать документацию быстрее и удобнее. Один из них — *DocFX*. Это мощный инструмент, разработанный компанией Microsoft, который умеет автоматически генерировать документацию по API, концептуальные статьи и многое другое на основе исходного кода и файлов в формате Markdown. Другой полезный инструмент — *Mermaid*. Это язык разметки для создания диаграмм и схем, таких как диаграммы последовательности и блок-схемы. В сочетании с Markdown он позволяет быстро добавлять визуальные элементы в документацию. Большинство редакторов кода имеют встроенную поддержку языков разметки, таких как Markdown и Mermaid, или же через устанавливаемые дополнительные расширения и плагины.

## Знакомство с документацией

Документирование поначалу может даваться непросто. Но со временем именно эта практика помогает команде лучше ориентироваться в коде, уверенно сопровождать проект и поддерживать его надежность в долгосрочной перспективе.

## Преимущества документации

Посмотрим, почему документация — это не просто полезное дополнение, а важная и необходимая часть разработки. Ниже перечислены ключевые причины.

- **Код становится понятнее.** Когда код документирован качественно, с ним проще работать — и вам, и другим. А вот недостаток пояснений может только запутать. Даже самые понятные на первый взгляд решения со временем теряют

ясность — и приходится заново разбираться. Документация помогает восстановить ход мыслей: почему решение было принято именно так, какую задачу решал тот или иной фрагмент. Особенно это важно в сложных проектах, где смысл отдельных частей не всегда очевиден.

- **Поддерживать документированный код легче.** Со временем код меняется — появляются новые функции, правятся ошибки, адаптируются технологии. Документация помогает вносить изменения с уверенностью, не опасаясь затронуть лишнего. Она подсказывает, как именно должен работать код и какие ожидания с ним связаны, — это помогает приниматьзвешенные решения при доработке.
- **Новые участники быстрее включаются в работу.** Когда в команду приходит новый человек, документация может серьезно упростить процесс адаптации. С ее помощью разработчик быстрее знакомится с проектом и может самостоятельно найти ответы на многие вопросы — не отвлекая коллег. Кроме того, новый участник быстрее вникает в проект, снижается вероятность ошибок при интеграции и стабилизируется процесс разработки.
- **Знания участников сохраняются в команде.** Всегда есть риск, что некоторые разработчики, особенно работавшие над важной частью системы, покинут команду. Если их знания нигде не зафиксированы, команда их теряет. Документация помогает сохранить важную информацию и передать ее другим — независимо от того, кто и когда присоединится к проекту. Это важный элемент устойчивой командной работы и надежной разработки.
- **Поддержка хороших практик.** Документирование помогает не только объяснить код, но и писать его понятнее с самого начала. Когда вы формулируете, как работает тот или иной фрагмент, легче заметить неучтенные условия, лишнюю сложность или логические неточности. Иногда сам процесс документирования подсказывает, как упростить или улучшить структуру кода, — и становится поводом для рефакторинга.
- **Переосмысление архитектуры и API.** Документация нередко включает не только комментарии в коде, но и внешние материалы — описание API, архитектурные схемы, примеры использования. Такой обзор всей системы помогает выявить дублирование, недоработки или области, которые можно упростить. Когда документирование становится частью процесса разработки, появляется привычка смотреть на решение как бы со стороны — и это часто приводит к более продуманному и удобному дизайну API.
- **Помощь при отладке и сопровождении.** Когда нужно разобраться, почему система ведет себя определенным образом, или найти источник ошибки, документация может стать хорошей отправной точкой. Она помогает понять, какие части системы могут быть связаны с проблемой. Это особенно важно в сложных решениях, где поведение системы зависит от взаимодействия множества компонентов.
- **Улучшение взаимопонимания между командами.** Документация важна не только внутри команды разработки. Ее могут создавать бизнес-аналитики,

чтобы пояснить техническим специалистам суть процессов, или технические авторы — для сотрудников, не связанных с ИТ. Качественно оформленные материалы помогают наладить взаимодействие между участниками с разными ролями и компетенциями.

Первоначальные усилия, вложенные в документирование кода и API, со временем обязательно окупаются. Благодаря этому кодовая база становится понятнее, ее проще сопровождать и развивать. Документация помогает сохранять знания в команде, упрощает отладку и снижает зависимость от конкретных людей. Это вклад не только в общее дело, но и в вашу собственную работу в будущем.



**Дельный совет:** предусмотрите возможность для участников команды и внешних разработчиков оставлять отзывы о документации. Так будет проще выявить участки, которые стоит уточнить или улучшить. Если предоставить нужные инструменты, сторонние разработчики нередко сами готовы доработать или дополнить документацию.

## Когда не стоит документировать

Документация (как правильно ее писать, мы обсудим далее в этой главе) — важная часть разработки, но не всегда она действительно необходима. В некоторых случаях документирование может оказаться избыточным и даже мешать. Ниже перечислены ситуации, когда документацию можно сократить или вовсе опустить.

- **Прототипирование и эксперименты.** При создании прототипа или проверке новых идей важна скорость. Подробная документация в таких случаях может только замедлить процесс. Прототип — это временное решение, задача которого скорее в том, чтобы прояснить подход или подтвердить гипотезу, а не стать основой для конечного продукта. В подобных ситуациях бывает достаточно кратких встроенных комментариев — особенно там, где логика может быть неочевидной из кода.
- **Самодокументирующийся код.** Если код написан понятно — с осмысленными именами переменных и методов, последовательной архитектурой и четкой структурой — он во многом объясняет себя сам. Такой подход опирается на идею, что лучший комментарий — это качественный код. Это не отменяет необходимости в документации более высокого уровня, описывающей архитектуру или ключевые процессы, но вполне может сократить количество подробных встроенных пояснений.
- **Крайне гибкие среды.** В условиях, где изменения происходят часто и быстро, поддерживать подробную документацию затруднительно. В таких случаях разумнее сосредоточить усилия на ключевых элементах — например, на API, открытых для внешних пользователей, или на сложных алгоритмах, критичных для работы системы. Документировать каждый внутренний сервис или метод при этом не всегда оправданно.

- **Недолговечный или временный код.** Если фрагмент кода предназначен для краткосрочного использования — например, как временное исправление или обходной путь, — подробно документировать его не всегда имеет смысл. В таких случаях достаточно краткого пояснения, чтобы было ясно, зачем этот код добавлен и когда его планируется заменить.
- **Внутренние инструменты для ограниченного использования.** Если сценарии или утилиты рассчитаны на работу внутри небольшой команды, где все знакомы с контекстом, подробная документация может оказаться избыточной. В таких случаях проще и эффективнее опираться на живое общение и быструю передачу знаний между участниками.
- **Ограниченные ресурсы.** В небольших командах или стартапах, где ресурсы и время сильно ограничены, приоритет часто отдается созданию рабочей функциональности, чтобы быстрее отвечать на потребности бизнеса. В таких условиях документация может временно отойти на второй план — особенно если задача требует немедленного результата.

Даже в ситуациях, когда документирование отходит на второй план, имеет смысл хотя бы зафиксировать общий замысел и назначение проекта, ключевые решения, которые неочевидны из кода, а также описать внешние API — особенно если с ними будут работать другие разработчики или системы.

Теперь перейдем к документированию исходного кода.

## Документирование исходного кода

Добавление комментариев к исходному коду — важная часть разработки ПО. Вам придется балансировать между необходимостью документирования и стремлением сохранить кодовую базу чистой и удобной для осуществления поддержки. Каждый комментарий означает дополнительную работу в будущем, когда что-то изменится, потому что вам придется обновлять комментарии в соответствии с изменениями кода.

### Когда следует документировать исходный код

Код может быть чрезмерно закомментирован, и не обязательно комментировать все подряд. Разберемся, в каких случаях разработчикам следует добавлять комментарии в код.

- **Сосредоточивайтесь на объяснении «почему», а не «что» и «как».** Сам код должен быть достаточно ясным, чтобы через названия переменных, функций и структуру показывать, *что* он делает и *как*. Такой подход называют самодокументирующемся кодом. Сегодня, когда есть инструменты вроде GitHub Copilot Chat, разобраться в «что» и «как» стало еще проще. Поэтому в комментариях особенно важно объяснять, *почему* были приняты те или иные решения, — на-

пример, почему выбран именно этот алгоритм, в чем суть сложной логики или зачем используется обходной путь.

- **Для пояснения сложных алгоритмов и неочевидной логики.** Если в коде используются математические расчеты, нестандартные алгоритмы или логика, которую трудно сразу понять, комментарии помогут восстановить ход рассуждений. Они позволяют пошагово объяснить, как работает алгоритм и как получен результат. Это особенно важно, когда упростить решение невозможно — например, из-за природы задачи.
- **Чтобы выделить важное поведение или побочные эффекты.** Если участок кода ведет себя нестандартно или вызывает побочные эффекты, которые неочевидны с первого взгляда, комментарии помогут обратить на это внимание. Такое поведение может быть связано как с особенностями используемых функций, так и с тем, что код написан не слишком прозрачно. В любом случае важно явно указать, что именно происходит — особенно если это влияет на стабильную работу приложения.
- **При работе с ограничениями или ошибками.** Такая ситуация встречается чаще, чем кажется. Иногда приходится использовать обходные решения из-за проблем в сторонних библиотеках, фреймворках или даже в языке. В таких ситуациях комментарии помогают сохранить контекст: почему был выбран именно этот путь, в чем суть ошибки и где можно найти подробности — например, ссылку на отчет или обсуждение. Такая информация особенно важна в будущем: она подскажет, когда обходной код уже не нужен и его можно безопасно удалить.
- **Для примеров использования.** В библиотеках, фреймворках и сложных функциях примеры прямо в комментариях могут быть очень полезны. Особенно это актуально для API и других общедоступных интерфейсов. Пример помогает понять, как вызвать функцию, какие параметры она принимает и что именно возвращает.
- **Для пометки задач и временных решений.** Комментарии с маркерами вроде TODO, UNDONE или HACK могут служить напоминанием о незавершенных задачах, временных решениях или участках, требующих пересмотра. Это удобно как для личных заметок, так и для координации внутри команды. Но такие пометки не должны захламлять код — если их слишком много, страдает читаемость. В идеале задачи лучше отслеживать через систему управления проектами или трекер проблем, а комментарий в коде использовать для пояснения контекста прямо на месте.



**Дельный совет:** метки TODO полезны, но не стоит превращать их в бессрочные напоминания. Если таких пометок становится слишком много, они теряют смысл. Постарайтесь возвращаться к ним как можно скорее — особенно если это важная функция или улучшение, а не просто идея «на потом».

- **Для пояснения внешних зависимостей.** Если код обращается к внешним библиотекам, API или источникам данных, имеет смысл добавить комментарии,

описывающие эти зависимости. Укажите, какие версии поддерживаются, как используется конкретная зависимость и есть ли возможные альтернативы. Такая информация помогает при обновлении компонентов и может значительно упростить поиск причин ошибок, связанных с внешними модулями.



**Дельный совет:** документируйте код осознанно. Комментарии не должны дублировать то, что уже ясно из кода. Лучше сосредоточиться на пояснении, почему принято то или иное решение, — вместо описания, что делает код или как он работает.

## Эффективные приемы комментирования кода

Ниже приведены несколько приемов, которые помогут писать полезные и уместные комментарии в исходном коде.

- **Следите за актуальностью комментариев.** Устаревшие пояснения могут сбивать с толку и вредить больше, чем их отсутствие. Комментарии должны изменяться вместе с кодом, к которому они относятся.
- **Избегайте избыточности.** Не описывайте очевидное. Комментарий должен дополнять код, а не повторять то, что уже и так понятно.
- **Используйте код как средство пояснения.** Стремитесь к читаемому коду: давайте понятные имена переменным и функциям, выносите сложные участки в отдельные методы, придерживайтесь логичной структуры. Такой подход называют самодокументирующемся кодом.
- **Оформляйте комментарии в формате документации.** Многие языки поддерживают специальные шаблоны комментариев, которые позволяют автоматически формировать документацию. Это особенно полезно при работе с публичными API и интерфейсами. Подробнее об этом речь пойдет в следующем разделе.

Комментарии стоит использовать для передачи контекста, пояснения решений и другой информации, которую сложно выразить только кодом. Помните, что основная аудитория таких комментариев — вы сами и ваша команда, имеющая доступ ко всему исходному коду проекта.

Теперь перейдем к документированию публичных API в библиотеках классов.

## Документирование API в библиотеках классов

Документирование исходного кода охватывает весь написанный вами код, включая типы с модификаторами `private` и `internal`, которые используются только внутри ваших проектов. Тем не менее особое внимание стоит уделить тем элементам, которые формируют публичный API — то есть всем типам и членам, доступным за пределами проекта. Такой API может использоваться как внутри вашей организации другими командами, так и за ее пределами, если вы публикуете NuGet-пакеты для широкой аудитории.

Документация API играет ключевую роль: она соединяет функциональность вашего ПО с его пользователями — будь то внутренние команды, партнеры или сторонние разработчики. Ниже приведены основные причины, почему стоит вложить усилия в ее создание.

- **Повышает удобство для разработчиков.** Полная, понятная и структурированная документация по API помогает разработчикам быстро находить нужную информацию. В нее важно включать примеры, возможные ошибки и рекомендации по устранению проблем. Недостатки или нехватка документации — одна из самых частых причин, по которым разработчики отказываются от использования API и переходят к альтернативным решениям.



В шестом издании одной из моих книг, в разделе о GraphQL, я сначала использовал библиотеку GraphQL.NET (в частности, пакет GraphQL.Server.Transports.AspNetCore и связанные с ним модули). Позже я перешел на платформу ChilliCream GraphQL — во многом потому, что ее API оказались лучше документированы и заметно приятнее в использовании. Качественная документация может быть решающим фактором при выборе технологии.

- **Снижает нагрузку на поддержку.** Качественно оформленная документация по API помогает разработчикам самостоятельно находить ответы на вопросы и устранять проблемы. Это уменьшает количество обращений в службу поддержки, экономит время команды и повышает доверенность пользователей. Освободившиеся ресурсы можно направить на развитие продукта, а не на разъяснение очевидного.
- **Способствует лучшему дизайну и единообразию.** Процесс документирования помогает взглянуть на API глазами тех, кто будет с ним работать. Так можно заранее заметить несогласованности, сложные места и отклонения от общепринятых подходов. Благодаря этому API получается более логичным, последовательным и понятным, с опорой на привычные стандарты — например, REST или GraphQL.
- **Облегчает тестирование и интеграцию.** Современная документация по API нередко включает интерактивные инструменты, такие как API Explorer или Sandbox, с помощью которых можно проверять работу конечных точек прямо в интерфейсе документации. Такая интерактивная документация ускоряет интеграцию, позволяет разработчикам экспериментировать с API в режиме реального времени, помогает понять структуру запросов и ответов, а также сразу выявить возможные ошибки.
- **Поддерживает устойчивость и развитие API.** По мере развития и изменения API поддержание актуальной документации помогает пользователям ориентироваться в новых возможностях и сохранять совместимость. Это особенно важно при введении новых версий API, объявлении устаревших методов или добавлении новых функций. Четко оформленные журналы изменений и правила работы с версиями помогают API оставаться надежным и масштабируемым инструментом.

- **Соответствие требованиям и стандартам безопасности.** В некоторых сферах — особенно там, где обрабатываются конфиденциальные данные или действуют отраслевые нормы, — документация может играть юридически значимую роль. В ней важно зафиксировать, каким стандартам безопасности соответствует API, как соблюдаются требования к защите данных и на каких условиях осуществляется доступ. Четкое описание этих аспектов помогает снизить юридические риски и повышает доверие со стороны пользователей.
- **Повышает конкурентоспособность и поддерживает сообщество.** Качественная документация — это еще и способ показать, что вы заботитесь о разработчиках. Она помогает продемонстрировать возможности API, привлечь новых пользователей и сформировать вокруг проекта сообщество. Активное сообщество может вносить обратную связь, сообщать об ошибках и даже участвовать в развитии API — особенно если проект с открытым исходным кодом.

Документация по API — это не просто техническая формальность. Она играет важную роль в удобстве использования, укрепляет доверие среди разработчиков, поддерживает безопасность и помогает сформировать сообщество вокруг вашего решения. Инвестиции в качественную документацию — это вклад в долгосрочный успех и востребованность вашего API.

А теперь перейдем к практической стороне: как именно оформлять такую документацию.

## Документирование с помощью XML-комментариев

В .NET предусмотрен стандартный способ оформления документации для общедоступных API — с помощью *XML-комментариев*. Такие комментарии начинаются с символов `///` и используются как для встроенной документации, так и для генерации внешней документации по кодовой базе.

XML-комментарии интегрируются в функционал *IntelliSense* в программе Visual Studio, помогая разработчикам прямо в процессе написания кода. Это повышает продуктивность: описания функций, параметры и возвращаемые значения становятся доступны сразу, без необходимости обращаться к внешним источникам. Благодаря этому проще разобраться в API и использовать его корректно.

Помимо всплывающих подсказок в редакторе, одно из главных достоинств XML-комментариев — возможность автоматически создавать полноценную документацию. Автоматизированные инструменты вроде *DocFX* или *Sandcastle* могут анализировать такие комментарии и формировать подробную, структурированную документацию в разных форматах — например, HTML или PDF. Поскольку она строится напрямую на основе кода, это снижает риск расхождений и помогает документации оставаться актуальной вместе с кодовой базой.

Использование XML-комментариев также способствует единообразному подходу к оформлению документации в рамках команды или проекта. Четкая структура — с краткими описаниями, параметрами, возвращаемыми значениями, исключениями

ми и другими элементами — упрощает навигацию и делает код более понятным для всех, кто с ним работает.

В отраслях с особыми требованиями к качеству и соответствуя стандартам XML-комментарии помогают удостовериться, что весь код документирован должным образом. Это важно для прохождения аудита, соблюдения нормативов и поддержки процессов контроля качества.

Такие комментарии полезны и при модульном тестировании, и в процессе отладки. Если метод сопровождается описанием ожидаемого поведения — включая граничные случаи и возможные исключения, — это помогает создавать более надежные тесты. А при отладке такие пояснения дают представление о том, как должен вести себя код, и помогают быстрее найти причину проблемы.



**Дельный совет:** XML-комментарии требуют совсем немного усилий, но при этом приносят большую пользу. Их поддержка — это разумное вложение в любой .NET-проект, особенно если вы хотите, чтобы код был понятен, удобен в сопровождении и пригоден для совместной работы.

## Как документировать код с помощью XML-комментариев

В .NET поддерживается набор стандартных XML-тегов, с помощью которых можно структурировать комментарии к коду. Эти теги распознаются инструментами создания документации и отображаются средством *IntelliSense*, помогая представить информацию ясно и последовательно.

Некоторые наиболее часто используемые теги перечислены в табл. 6.1.

**Таблица 6.1.** Часто используемые теги в XML-комментариях

Тег	Описание
<summary>	Описание типа или члена
<remarks>	Дополнительная информация, дополняющая основное описание
<param name="name">	Описание параметра метода
<paramref name="name" />	Ссылка на параметр в тексте комментария
<returns>	Описание возвращаемого значения метода
<exception cref="member">	Описание исключений, которые может выбросить элемент. Тег можно применять к методам, свойствам, событиям и индексаторам. Для создания ссылки на класс исключения используется атрибут <code>cref</code> со ссылкой на соответствующий элемент

*Продолжение ↗*

**Таблица 6.1 (окончание)**

Тег	Описание
<example>	Пример использования типа или члена, часто с блоком <code>
<see cref="member" />, <seealso />	Ссылка на другие элементы кода для предоставления дополнительного контекста
<para>	Формирует отдельный абзац (аналог <p> в HTML)
<c>, <code>	Оформление текста как кода — встроенного (<c>) или блочного (<code>)
<list type="bullet">	Создание списка. Тип может быть bullet, number или table
<item>	Элемент внутри списка
<b>, <i>, <u>	Оформление текста: полужирный, курсив, подчеркивание
<inheritdoc [ cref="" ] [path="" ]/>	Наследование XML-комментариев от базовых классов, интерфейсов или аналогичных методов для избежания дублирования



**Дельный совет:** тег <inheritdoc> особенно полезен, когда один и тот же метод реализуется в нескольких местах. Он позволяет автоматически наследовать XML-комментарии, снижая риск несоответствий и устаревшей информации в документации.

Хотя сложное форматирование в XML-комментариях недоступно, можно использовать теги <b>, <i> и <u> для выделения текста полужирным, курсивом и подчеркиванием. Для ссылок на код предусмотрены теги <c> (для строчных (встроенных) фрагментов) и <code> (для блоков кода). Это помогает визуально отделять имена переменных или участки кода от основного текста в документации, как показано на рис. 6.3.

```
/// <summary>
/// This method configures the console's culture.
/// For example, <c>ConfigureConsole()</c> sets culture to <i>en-US</i>.
/// The following code sets the console culture to <b>French in France</b>:
/// <code>
/// ConfigureConsole("fr-FR");
/// </code>
/// </summary>
```



**Внимание!** В программах Code и Rider поддерживаются только теги <b> и <i> для выделения текста полужирным и курсивом. Тег <u> (подчеркивание) корректно отображается только в Visual Studio.

Для оформления перечней в XML-комментариях используется тег <list type="list\_style">. С его помощью можно структурировать информацию в виде

маркированного или нумерованного списка — например, при описании шагов, требований или важных соображений. Такой формат помогает сделать текст более читаемым и упорядоченным:

```
/// <summary>
/// Consider the following points:
/// <list type="number">
///   <item><b>First</b> point.</item>
///   <item><u>Second</u> point.</item>
/// </list>
/// </summary>
```

Посмотрим на это в действии.

1. В редакторе кода создайте проект **Class Library** (Библиотека классов) / **classlib**,

присвойте ему имя **PacktLibrary** и добавьте в решение **Chapter06**.

2. В файле **PacktLibrary.csproj** выполните следующие настройки:

- включите статический импорт классов **System.Console** и **System.Text.Encoding**;
- добавьте глобальный импорт пространства имен **System.Globalization** для всех файлов кода;
- настройте проект так, чтобы предупреждения компилятора обрабатывались как ошибки:

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

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

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

</Project>
```

3. Переименуйте файл **Class1.cs** в **Utility.cs**.

4. В файле **Utility.cs** удалите все содержимое и определите класс с двумя методами:

```
namespace PacktLibrary;

public class Utility
{
  public static string CurrentConsoleCulture()
  {
    return $"CurrentCulture: {CultureInfo.CurrentCulture.DisplayName}";
}
```

```

public static void ConfigureConsole(string culture = "en-US",
    bool useComputerCulture = false)
{
    // Для включения в консоли символов Юникода, например символа евро
    OutputEncoding = UTF8;

    if (!useComputerCulture)
    {
        CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo(culture);
    }

    WriteLine(CurrentConsoleCulture());
}

public static void WriteLineInColor(string text, ConsoleColor color)
{
    ConsoleColor previousColor = ForegroundColor;
    ForegroundColor = color;
    WriteLine(text);
    ForegroundColor = previousColor;
}
}

```



Метод CurrentConsoleCulture можно было бы реализовать как свойство, но немного позже в этом разделе мы рассмотрим, как документировать возвращаемое значение метода.

5. Вставьте пустую строку над методом `ConfigureConsole`.
6. Введите три косые черты (///), либо выберите команду меню `Edit > IntelliSense > Insert Comment` (Редактирование > IntelliSense > Добавить комментарий), либо щелкните правой кнопкой мыши по методу и выберите в контекстном меню пункт `IntelliSense > Insert Comment` (IntelliSense > Добавить комментарий).
7. При добавлении комментария автоматически вставляются элементы, документирующие два параметра: `culture` и `useComputerCulture`.

```

/// <summary>
///
/// </summary>
/// <param name="culture"></param>
/// <param name="useComputerCulture"></param>
public static void ConfigureConsole(string culture = "en-US",
    bool useComputerCulture = false)

```

8. Дополните XML-комментарий следующими текстами:
  - `<summary>`: Configure the console to support Unicode characters and set the culture to en-US (by default), or to a specified culture, or to the local computer culture.
  - `<param name="culture">`: Set to an ISO culture code like fr-FR, en-GB, or es-AR.

```

■ <param name="useComputerCulture">: Set to true to change the culture to the
local computer's culture.

/// <summary>
/// Configure the console to support Unicode characters and set the culture to
en-US (by default), or to a specified culture, or to the local computer culture.
/// </summary>
/// <param name="culture">Set to an ISO culture code like fr-FR, en-GB,
or es-AR.</param>
/// <param name="useComputerCulture">Set to true to change the culture to the
local computer's culture.</param>
public static void ConfigureConsole(string culture = "en-US",
bool useComputerCulture = false)

```



**Дельный совет:** текст описания параметров отображается во всплывающих подсказках, поэтому лучше сформулировать его кратко и по сути. Укажите назначение параметра и при необходимости добавьте один-два примера возможных значений.

9. Вставьте пустую строку над методом `CurrentConsoleCulture`.
10. Введите три косые черты (///) либо воспользуйтесь той же командой `Insert ▶ Comment (IntelliSense ▶ Добавить комментарий)`, как и ранее.
11. Добавьте в XML-комментарий к методу `CurrentConsoleCulture` описание метода и возвращаемого значения:

```

/// <summary>
/// Gets the current console culture in its native language.
/// </summary>
/// <returns>The current console culture as a string.</returns>
public static string CurrentConsoleCulture()

```



Пока XML-комментарии будут добавлены только к первым двум методам. Третий метод мы оставим без описания — к нему вернемся позже.

12. В редакторе кода создайте проект `Console App` (Консольное приложение) / `console`, присвойте ему имя `DocsApp` и добавьте в решение `Chapter06`.
13. В файле `DocsApp.csproj` выполните следующие настройки:
  - подключите статический импорт `System.Console` для всех файлов кода;
  - добавьте ссылку на проект `PacktLibrary`;
  - настройте компилятор на обработку предупреждений как ошибок:

```

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

<PropertyGroup>
<OutputType>Exe</OutputType>

```

```

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

<ItemGroup>
    <ProjectReference Include="..\PacktLibrary\PacktLibrary.csproj" />
</ItemGroup>

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

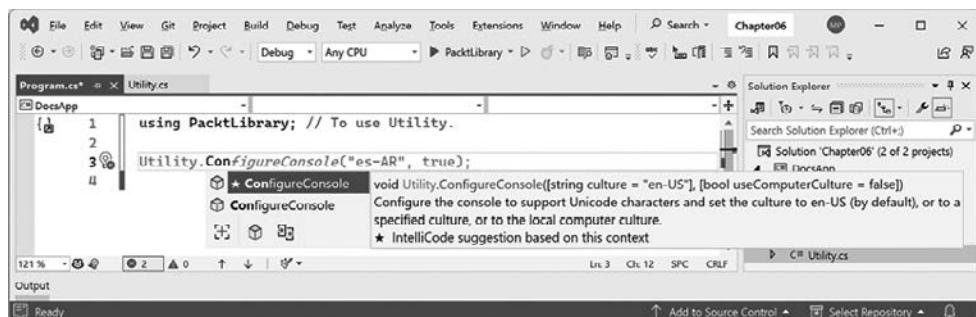
</Project>

```

- Выполните сборку проекта `DocsApp`. В результате будет скомпилирован проект `PacktLibrary`, и его сборка будет скопирована в папку `bin` консольного приложения.
- В файле `Program.cs` удалите все содержимое и импортируйте пространство имен `PacktLibrary`.

```
using PacktLibrary; // Подключение класса Utility
```

- На новой строке добавьте вызов метода `ConfigureConsole` и посмотрите на всплывающую подсказку IntelliSense (рис. 6.1).

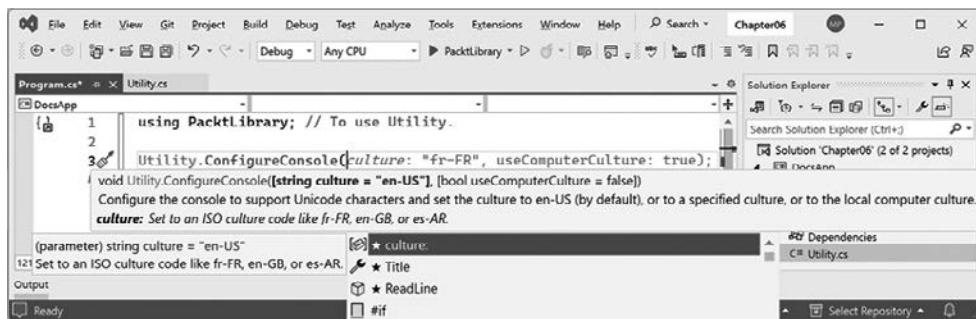


**Рис. 6.1.** Подсказка IntelliSense для метода, документированного с помощью XML-комментариев



На рис. 6.1 видно, что автозавершение кода в Visual Studio предлагает возможные аргументы для метода — например, строку "es-AR" и значение true. Однако такая комбинация не имеет смысла: если параметру `useComputerCulture` присвоено значение true, указанная культура будет проигнорирована. Искусственный интеллект в Visual Studio пока еще нуждается в доработке. :)

- Допишите оставшуюся часть имени метода, введите открывающую круглую скобку (и обратите внимание на появившуюся подсказку IntelliSense (рис. 6.2)).



**Рис. 6.2.** Подсказка Intellisense для параметра culture

18. Завершите вызов метода `ConfigureConsole`, затем добавьте выражение для вызова метода `WriteLineInColor`, позволяющее вывести текущую культуру консоли красным цветом:

```
Utility.ConfigureConsole();
Utility.WriteLineInColor(
    text: Utility.CurrentCulture(), color: ConsoleColor.Red);
```

19. Запустите проект. В выводе будет показано, что текущая культура консоли — английский (США) — сначала отображается черным, а затем красным цветом, как показано в следующем примере:

```
CurrentCulture: English (United States)
CurrentCulture: English (United States)
```

Хотя XML-комментарии в .NET не поддерживают стилизацию с помощью CSS или других механизмов оформления, читаемость и структура документации могут быть значительно улучшены за счет использования семантических тегов.

Теперь посмотрим, как можно сгенерировать документацию для этого публичного API с помощью инструмента DocFX.

## Создание документации с помощью DocFX

Инструмент DocFX создает статический HTML-сайт на основе исходного кода — для генерации документации по публичным API — и файлов Markdown, с помощью которых можно добавлять дополнительные страницы. Макет и оформление сайта можно настраивать с помощью шаблонов.

Когда XML-комментарии обрабатываются генераторами документации, такими как DocFX, появляется больше гибкости в оформлении итоговых страниц. Хотя сами комментарии не содержат стилей, выходная документация может быть дополнительно стилизована с помощью таблиц CSS — например, для адаптации к фирменному стилю или улучшения читаемости.

DocFX распространяется как глобальный инструмент CLI для платформы .NET. Далее мы установим его и настроим проект библиотеки классов на генерацию XML-файла документации из исходного кода.

1. В терминале установите DocFX:

```
dotnet tool install -g docfx
```

2. Если DocFX уже установлен, обновите его до последней версии:

```
dotnet tool update -g docfx
```

3. Проверьте, что DocFX установлен:

```
dotnet tool list -g
```

4. На момент написания книги (февраль 2024 года) актуальной была версия 2.75.2 (рис. 6.3):

Package Id	Version	Commands
docfx	2.75.2	docfx
dotnet-ef	8.0.1	dotnet-ef
microsoft.web.librarymanager.cli	2.1.175	libman



Актуальную версию DocFX можно найти на сайте [www.nuget.org/packages/docfx](http://www.nuget.org/packages/docfx).

5. В файле `PacktLibrary.csproj` проекта добавьте элементы, отвечающие за генерацию документации, а также за управление путем и именем создаваемого XML-файла:

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
  <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <DocumentationFile>
    bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml
  </DocumentationFile>
</PropertyGroup>
```



Значение `$(Configuration)` будет `Debug` или `Release` в зависимости от выбранной конфигурации сборки. Значение `$(TargetFramework)` для текущего проекта — `net8.0`. Значение `$(AssemblyName)` — `PacktLibrary`. Также можно использовать `$(OutputPath)`, который содержит полный путь к выходной папке со сборками.

6. Выполните сборку проекта `PacktLibrary`. Возникнут ошибки компиляции.
7. Посмотрите, какие именно ошибки возникли:

```
Error CS1591 Missing XML comment for publicly visible type or member
'Utility.WriteLineInColor()'
Error CS1591 Missing XML comment for publicly visible type or member
'Utility'
```



**Дельный совет:** после включения генерации документации анализатор кода начнет предупреждать о необходимости добавить XML-комментарии ко всем публичным типам и членам. Поскольку проект настроен на обработку предупреждений как ошибок, эти комментарии нужно будет обязательно добавить — иначе сборка завершится с ошибками.

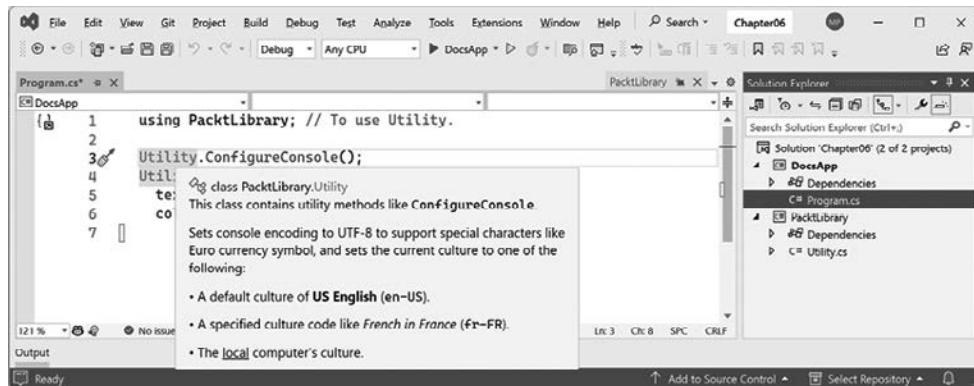
8. В файле `Utility.cs` вставьте пустую строку над объявлением класса `Utility`.
9. Введите три косые черты (///), либо выберите команду меню `Edit ▶ IntelliSense ▶ Insert Comment` (Редактирование ▶ IntelliSense ▶ Добавить комментарий), либо щелкните правой кнопкой мыши по методу и выберите в контекстном меню пункт `IntelliSense ▶ Insert Comment` (IntelliSense ▶ Добавить комментарий).
10. Добавьте форматированный текст с простыми стилями и маркированным списком в XML-комментарий, как показано в следующем фрагменте разметки:

```
/// <summary>
/// This class contains utility methods like <c>ConfigureConsole</c>.
/// <para>
///   Sets console encoding to UTF-8 to support special characters like
///   Euro currency symbol,
///   and sets the current culture to one of the following:
///   <list type="bullet">
///     <item>A default culture of <b>US English</b> (<c>en-US</c>).</item>
///     <item>A specified culture code like <i>French in France</i>
///       (<c>-fr-FR</c>).</item>
///     <item>The <u>local</u> computer's culture.</item>
///   </list>
/// </para>
/// </summary>
```

11. Вставьте пустую строку над методом `WriteLineInColor`.
12. Введите три косые черты (///) либо воспользуйтесь той же командой `Insert ▶ Comment` (IntelliSense ▶ Добавить комментарий), как и ранее.
13. Добавьте текст в XML-комментарий к методу `WriteLineInColor`:

```
/// <summary>
/// Write a message to the console in the specified color.
/// </summary>
/// <param name="text">The text of the message.</param>
/// <param name="color">The color of the text.</param>
public static void WriteLineInColor(string text, ConsoleColor color)
```

14. Выполните сборку проекта `PacktLibrary` — ошибки больше не должны появляться.
15. В проекте `DocsApp` в файле `Program.cs` наведите указатель мыши на имя `Utility` (рис. 6.3).



**Рис. 6.3.** Пользовательская форматированная всплывающая подсказка для класса Utility

16. В папке `bin\Debug\net8.0` откройте файл `PacktLibrary.xml`. В нем можно увидеть, что типы начинаются с префикса `T` (например, `T:PacktLibrary.Utility`), а члены — с `M` (например, `M:PacktLibrary.Utility.CurrentConsoleCulture`):

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>PacktLibrary</name>
  </assembly>
  <members>
    <member name="T:PacktLibrary.Utility">
      <summary>
        This class contains utility methods like <c>ConfigureConsole</c>.
      <para>
        ...
      </para>
      </summary>
    </member>
    <member name="M:PacktLibrary.Utility.CurrentConsoleCulture">
      <summary>
        Gets the current console culture in its native language.
      </summary>
      <returns>The current console culture as a string.</returns>
    </member>
    ...
  </members>
</doc>
```

Другие префиксы, используемые в XML-комментариях, включают `N` — для пространств имен, `F` — для полей, `P` — для свойств, `E` — для событий и др. Полезно знать

об этих префиксах, поскольку их можно использовать с атрибутом `cref` в тегах `<see>` и `<seealso>`, чтобы создавать ссылки на соответствующие элементы кода.

Например, из метода `ConfigureConsole` можно создать ссылку на метод `CurrentConsoleCulture`:

```
/// <para>This method calls the <seealso cref="M:PacktLibrary.Utility.CurrentCulture" /> method to output the current culture.</para>
```

В Visual Studio есть функция автодополнения кода, поэтому запоминать весь синтаксис необязательно.



**В копилку знаний:** подробнее о создании ссылок в документации API можно прочитать по адресу [learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/xmldoc/recommended-tags#generate-links-and-references](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/xmldoc/recommended-tags#generate-links-and-references).

Теперь перейдем к созданию проекта документации DocFX.

## Создание проекта DocFX

Сейчас мы с помощью DocFX создадим веб-страницу и PDF-файл для документирования библиотеки классов `PacktLibrary` и ее класса `Utility`.

1. В терминале перейдите в папку проекта `PacktLibrary` и инициализируйте проект DocFX.

```
docfx init
```

2. Во время инициализации инструмент задаст несколько вопросов. Ответьте на них по порядку — как показано ниже.

```
This utility will walk you through creating a docfx project.
It only covers the most common items, and tries to guess sensible defaults.
```

```
Name (mysite): PacktLibrary
Generate .NET API documentation? [y/n] (y): y
.NET projects location (src): PacktLibrary
Markdown docs location (docs): docs
Enable site search? [y/n] (y): y
Enable PDF? [y/n] (y): y
```

```
About to write to C:\tools-skills-net8\Chapter06\PacktLibrary\docfx.json:
```

```
{
  "metadata": [
    {
      "src": [
        {
          "src": "../PacktLibrary",
          "files": [
            "**/*.csproj"
```

```

        ]
    }
],
"dest": "api"
}
],
"build": {
    "content": [
        {
            "files": [
                "**/*.{md,yml}"
            ],
            "exclude": [
                "_site/**"
            ]
        }
    ],
    "resource": [
        {
            "files": [
                "images/**"
            ]
        }
    ],
    "output": "_site",
    "template": [
        "default",
        "modern"
    ],
    "globalMetadata": {
        "_appName": "PacktLibrary",
        "_appTitle": "PacktLibrary",
        "_enableSearch": true,
        "pdf": true
    }
}
}
Is this OK? [y/n] (y): y
Project created at C:\tools-skills-net8\Chapter06\PacktLibrary
Run docfx C:\tools-skills-net8\Chapter06\PacktLibrary\docfx.json --serve
to launch the site.

```

3. В той же папке запустите проект DocFX на встроенным веб-сервере.

```
docfx docfx.json --serve
```

4. После запуска будут созданы HTML-страницы и PDF-файл в папке проекта, а также в подпапках `api` и `docs`. Одновременно откроется веб-сервер, позволяющий сразу протестировать сайт документации.

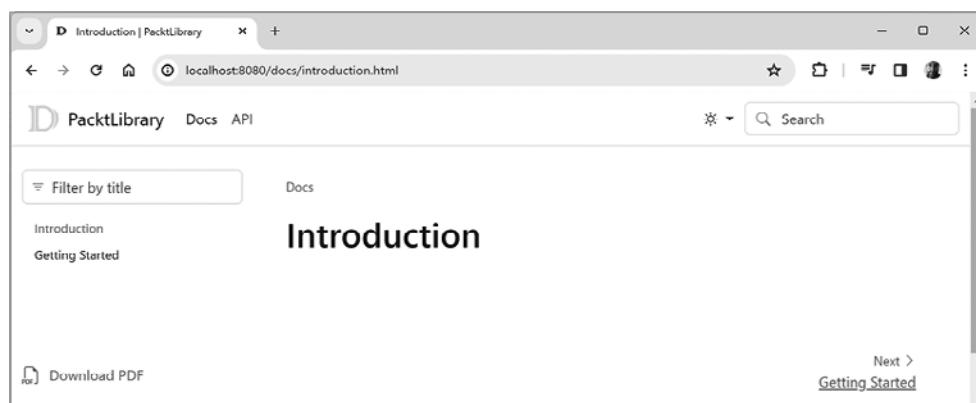
```

Using .NET Core SDK 8.0.200
Loading project C:/tools-skills-net8/Chapter06/PacktLibrary/PacktLibrary.csproj
Determining projects to restore...

```

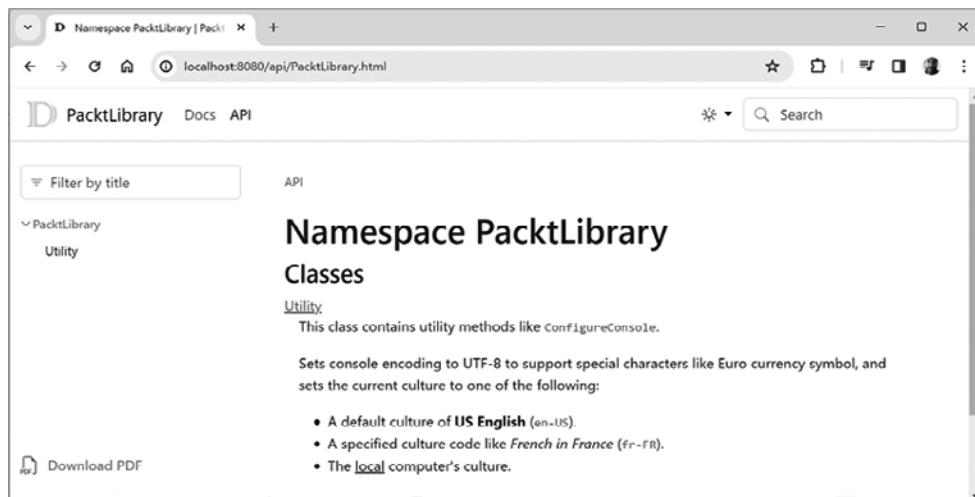
```
All projects are up-to-date for restore.  
Processing PacktLibrary  
Creating output...  
Searching custom plugins in directory C:\Users\markj\.dotnet\tools\  
store\docfx\2.75.2\docfx\2.75.2\tools\net8.0\any\...  
Post processor ExtractSearchIndex loaded.  
No files are found with glob pattern images/**, excluding <none>, under  
directory "C:\tools-skills-net8\Chapter06\PacktLibrary"  
7 plug-in(s) loaded.  
Building 3 file(s) in TocDocumentProcessor(BuildTocDocument)...  
Building 2 file(s) in ManagedReferenceDocumentProcessor  
(BuildManagedReferenceDocument=>  
SplitClassPageToMemberLevel=>ValidateManagedReference-  
DocumentMetadata=>ApplyOverwriteDocumentForMref=>FillReferenceInformation)...  
Building 3 file(s) in ConceptualDocumentProcessor(BuildConceptualDocument=>  
ValidateConceptualDocumentMetadata)...  
Applying templates to 8 model(s)...  
XRef map exported.  
Extracting index data from 5 html files  
  
api\toc.pdf ----- 100%  
docs\toc.pdf ----- 100%  
toc.pdf ----- 100%  
  
Serving "C:\tools-skills-net8\Chapter06\PacktLibrary\_site" on http://  
localhost: 8080. Press Ctrl+C to shut down.
```

5. Откройте браузер и перейдите по адресу <http://localhost:8080>. В операционной системе Windows можно щелкнуть на ссылке, нажав и удерживая клавишу Ctrl.
6. На главной странице в верхнем меню выберите пункт Docs (Документы). Первая страница называется Introduction (Введение) — она пока пуста и готова к наполнению. Здесь же уже есть навигационные ссылки, в том числе на страницу Getting Started (Начало работы) (рис. 6.4).



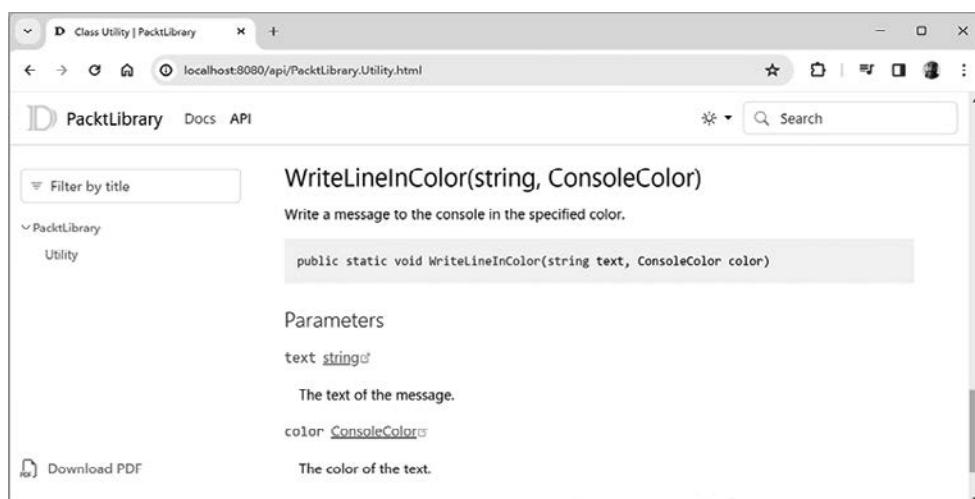
**Рис. 6.4.** Входная страница документации

7. На странице **Introduction** (Введение) в верхнем меню выберите пункт **API** — отобразится документация для пространства имен **PacktLibrary** и его класса **Utility** (рис. 6.5).



**Рис. 6.5.** Документация с пользовательским оформлением для пространства имен PacktLibrary

8. Щелкните на имени **Utility**, прокрутите страницу вниз и изучите документацию для метода **WriteLineInColor** (рис. 6.6).



**Рис. 6.6.** Сгенерированная документация для метода `WriteLineInColor`

9. Нажмите кнопку Download PDF (Загрузить PDF) — откроется PDF-файл, содержащий полную документацию по библиотеке классов на трех страницах.
10. Закройте браузер и остановите работу веб-сервера, нажав сочетание клавиш **Ctrl+C** в терминале.

Теперь вы знаете, что XML-комментарии можно использовать для управления содержимым раздела API на сайте документации. А как настраивается раздел Docs? Давайте разберемся.

## Добавление собственных страниц в документацию

Чтобы разместить на сайте документации дополнительные страницы, можно создать собственную структуру папок и добавить файлы в формате Markdown в папку `docs`. Сейчас в этой папке находятся примеры пустых страниц — `introduction.md` и `getting-started.md`, а также файл `toc.yml`, отвечающий за оглавление.

- name: Introduction  
  href: introduction.md
- name: Getting Started  
  href: getting-started.md

Когда вы добавляете новые файлы Markdown, необходимо обновить файл `toc.yml`, чтобы статическая версия сайта корректно отобразила новые разделы. Например, при добавлении файла `documenting-your-code.md` нужно внести соответствующую запись в документ `toc.yml`.

- name: Introduction  
  href: introduction.md
- name: Getting Started  
  href: getting-started.md
- name: Documenting Your Code  
  href: documenting-your-code.md

На сайте DocFX доступен выбор шаблонов, созданных сообществом. Их можно скачать и поместить в папку `templates` внутри вашего проекта DocFX. После этого нужно обновить массив шаблонов в файле `docfx.json`, чтобы он ссылался на новый шаблон.



**В копилку знаний:** подробнее о системе шаблонов DocFX вы можете узнать по адресу [dotnet.github.io/docfx/docs/template.html](https://dotnet.github.io/docfx/docs/template.html).

## Язык разметки Markdown

*Markdown* — это легкий язык разметки с простым синтаксисом, разработанный для форматирования обычного текста. Его можно легко преобразовать в HTML и другие форматы с помощью соответствующих инструментов. Язык был создан

Джоном Грубером и Аароном Шварцем в 2004 году и быстро завоевал популярность среди разработчиков и авторов технической документации благодаря своей простоте и удобству.

Markdown отличается тем, что даже в необработанном виде остается легко читаемым. Поэтому это идеальный выбор для написания структурированных текстов без необходимости изучать более сложные языки разметки, такие как HTML. Некоторые элементы синтаксиса показаны в табл. 6.2.

**Таблица 6.2.** Распространенный синтаксис Markdown

Синтаксис	Описание
# Заголовок 1	Заголовок верхнего уровня
## Заголовок 2	Заголовок второго уровня
### Заголовок 3	Заголовок третьего уровня и далее
*курсив* или _курсив_	Курсивный текст
**полужирный** или __полужирный__	Полужирный текст
[Текст ссылки](url-адрес)	Гиперссылка
![Замещающий текст](url-адрес)	Изображение с альтернативным текстом
> Цитата	Цитата (текст в кавычках)
- Пункт списка или * Пункт списка	Элемент маркированного списка
1. Пункт списка	Элемент нумерованного списка
`Код в тексте`	Встроенный (строчный) код
``` или ```cs	Многострочный блок кода, с подсветкой синтаксиса
--- или ***	Горизонтальная линия

А теперь рассмотрим несколько примеров.

## Заголовки

С помощью Markdown можно создавать заголовки, предваряя их знаком решетки #. Количество знаков решетки перед текстом заголовка определяет уровень заголовка:

```
# Заголовок 1
## Заголовок 2
### Заголовок 3
#### Заголовок 4
```

При отображении размер заголовков будет постепенно уменьшаться по мере увеличения уровня.

## Форматирование текста

Для акцента на тексте его можно выделить курсивом или полужирным начертанием:

*\*Этот текст будет выделен курсивом\**  
Этот текст также будет выделен курсивом  
**\*\*Этот текст будет полужирным\*\***  
Этот текст также будет полужирным  
**\*\*Вы можете комбинировать их\*\***

## Создание списков

В Markdown можно легко создавать как неупорядоченные (маркированные), так и упорядоченные (нумерованные) списки:

- Элемент 1
  - Элемент 2
    - Элемент 2а
    - Элемент 2б
1. Первый элемент
  2. Второй элемент

## Ссылки и изображения

Для добавления ссылок и изображений используется одинаковая структура: текст заключается в квадратные скобки, а адрес — в круглые. Чтобы вставить изображение, перед конструкцией нужно поставить восклицательный знак:

[Сайт OpenAI](<https://www.openai.com>)

![Логотип](<https://example.com/logo.png>)

## Блоки кода и подсветка синтаксиса

Чтобы вставить многострочный блок кода без подсветки синтаксиса, используйте тройные обратные апострофы (грависы):

```
```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25
}```
```

Если нужно подсветить синтаксис для конкретного языка программирования, укажите его после первых трех грависов:

```
```cs
void HelloWorld()
{
    Console.WriteLine("Hello, world!");
}```
```

```
```python
def hello_world():
    print("Hello, world!")
```

```

## Таблицы

В Markdown таблицы создаются с помощью вертикальных черт (|) и тире (-). Вертикальные черты задают границы столбцов, а тире отделяют заголовок от основного содержимого таблицы. Такой формат легко читается и позволяет аккуратно представить данные.

Чтобы создать таблицу в Markdown, сначала задайте заголовки столбцов, а сразу под ними — строку с тире, которая отделяет заголовки от содержимого. Каждый столбец разделяется вертикальной чертой (|). Ниже вы можете добавлять строки с данными, используя тот же формат.

Пример простой таблицы, в которой представлена информация о разных языках программирования, показан в следующем фрагменте Markdown-разметки:

| Язык программирования | Создатель        | Год  |
|-----------------------|------------------|------|
| C                     | Деннис Ритчи     | 1972 |
| Python                | Гвидо ван Россум | 1991 |
| Java                  | Джеймс Гослинг   | 1995 |

В результате получится таблица с тремя столбцами и заголовком. Столбцы аккуратно выровнены, а содержимое легко читается. Большинство движков рендеринга Markdown автоматически оформляют таблицы так, чтобы заголовки выделялись визуально — например, полужирным шрифтом.

Markdown также поддерживает выравнивание текста в ячейках таблицы. Для этого в строке-разделителе между заголовком и данными можно использовать двоеточия (:):

- :--- — выравнивание по левому краю;
- ---: — выравнивание по правому краю;
- :---: — выравнивание по центру.

Пример таблицы с тремя разными типами выравнивания:

| По левому краю | По центру | По правому краю |
|----------------|-----------|-----------------|
| текст          | текст     | текст           |
| еще текст      | еще текст | еще текст       |

Следуя этим простым правилам, вы сможете создавать таблицы в Markdown под самые разные задачи — от компактного представления данных до аккуратного и наглядного оформления информации.

Markdown популярен именно благодаря своей простоте. Он отлично подходит для написания технической документации: понятный синтаксис, быстрая разметка и качественно структурированный результат. Освоив Markdown, вы сможете заметно упростить себе работу.



Шпаргалки по Markdown доступны по следующим адресам: [www.markdownguide.org/cheat-sheet/](http://www.markdownguide.org/cheat-sheet/) и [github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet](https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet).

## Документирование сервисов

Документирование сервисов — особенно веб-API — помогает сделать их удобными для использования, легко сопровождаемыми и доступными для разработчиков. Сегодня существуют различные форматы и инструменты, которые стали стандартом для описания API и сервисов. У каждого из них — свои особенности и области применения.

### Важные моменты насчет документирования сервисов

Ниже перечислены ключевые моменты, которые стоит учитывать при документировании сервисов.

- **Учитывайте аудиторию.** Понимайте, для кого вы пишете документацию. Разным категориям пользователей может требоваться разный уровень технической глубины. Например, инструменты Microsoft Power BI и другие продукты Power Platform часто используются не разработчиками. Поэтому не стоит предполагать, что все пользователи вашего сервиса — опытные разработчики.
- **Приводите примеры и учебные сценарии.** Покажите, как выполнять типичные запросы или использовать сервис в распространенных сценариях. Так пользователи смогут быстрее освоиться и уменьшить количество вопросов.
- **Учитывайте изменения в версиях.** По мере развития сервиса важно, чтобы документация соответствовала актуальной версии. При этом желательно сохранить доступ к более ранним версиям, если они все еще поддерживаются. Устаревшие функции и конечные точки следует пометить соответствующим образом.
- **Обеспечьте доступность документации.** Структура, поиск и адаптивный интерфейс документации напрямую влияют на удобство ее использования. Продуманный пользовательский опыт делает документацию по-настоящему полезной.
- **Опишите требования к проверке подлинности.** Четко укажите, как происходит проверка подлинности, приведите примеры запросов, включая заголовки и токены.

- **Документируйте ошибки.** Предусмотрите отдельный раздел с описанием возможных ошибок, объяснением их причин и рекомендациями по устранению. Такой подход помогает быстрее решать проблемы и повышает надежность взаимодействия с сервисом.

## Инструменты для документирования сервисов

Выбор подходящего инструмента и метода зависит от специфики вашего проекта, уровня сложности сервиса и предпочтений команды разработчиков и пользователей API. Для полноценного документирования сложных или крупных сервисов может потребоваться использовать сразу несколько инструментов.

Среди популярных стандартов и решений для документирования сервисов — следующие.

- **Swagger и OpenAPI.** Спецификация OpenAPI (OAS) — это один из самых популярных форматов описания API. Она позволяет задокументировать все — от конечных точек и схем запросов/ответов до механизмов проверки подлинности. Описание сервиса создается в формате JSON или YAML. Несмотря на некоторую многословность, широкая поддержка инструментов и активное сообщество делают OAS надежной основой для документирования RESTful API.
- **Swashbuckle.** Это одно из самых популярных решений в экосистеме OpenAPI для проектов .NET. Оно умеет автоматически генерировать спецификацию OpenAPI на основе кода, добавляя визуальный интерфейс Swagger UI для интерактивной работы с API, а также создавать клиентские библиотеки. Альтернативы — NSwag и FastEndpoints, которые предлагают похожие функции и могут быть предпочтительнее в зависимости от задач проекта.



**Внимание!** Инструмент Swashbuckle использовался в шаблонах проектов .NET начиная с .NET 5, но в .NET 9 его планируется исключить. Подробности об этом решении доступны по адресу [github.com/dotnet/aspnetcore/issues/54599](https://github.com/dotnet/aspnetcore/issues/54599). Разработка Swashbuckle больше не ведется его автором из сообщества. Вместо него Microsoft планирует развивать функциональность, доступную в пакете Microsoft.AspNetCore.OpenApi, включая генерацию документов OpenAPI. Пока не решено, будет ли также реализован динамический веб-интерфейс для интерактивного тестирования API.

- **Postman.** Хотя этот инструмент чаще всего используют для тестирования API, его можно применять и для документирования. Документация создается в процессе тестирования, что удобно при активной разработке. Однако подход тесно привязан к экосистеме Postman, и для некоторых организаций использование Postman для работы с публичными веб-сервисами может быть нежелательным.
- **Read the Docs.** Платформа для размещения документации, автоматически синхронизируемой с системой контроля версий. Подходит для более развернутой

документации, включающей не только описание API, но и контекст, примеры, учебные материалы. Поддерживает ведение документации по нескольким версиям сервиса одновременно, что особенно полезно для API с длительным жизненным циклом.

Теперь разберемся, как применять OpenAPI для документирования сервисов.

## Основы спецификации OpenAPI (OAS)

OpenAPI (ранее известный как Swagger) — это открытый стандарт для описания, создания, использования и визуализации RESTful-веб-сервисов. Он помогает как людям, так и системам понимать возможности сервиса — без необходимости просматривать исходный код, читать дополнительную документацию или анализировать сетевой трафик. В этом разделе мы рассмотрим, что важно знать разработчику .NET для документирования сервисов с помощью OpenAPI.

Разработчику следует понимать структуру и синтаксис спецификации OpenAPI. Этот формат не привязан к какому-либо языку программирования. Он описывает конечные точки API, выполняемые операции, входные и выходные данные, способы проверки подлинности и другие важные аспекты. Описание может быть составлено в формате YAML или JSON. Подробности можно найти на сайте [spec.openapis.org/oas/latest.html](https://spec.openapis.org/oas/latest.html).

Познакомимся с экосистемой инструментов OpenAPI. В нее входят:

- **Swashbuckle** — включает такие компоненты, как Swagger UI, Swagger Editor и Swagger Codegen: [github.com/domaindrivendev/Swashbuckle.AspNetCore](https://github.com/domaindrivendev/Swashbuckle.AspNetCore);
- **NSwag** — [github.com/RicoSuter/NSwag](https://github.com/RicoSuter/NSwag);
- **FastEndpoints** — [github.com/FastEndpoints/FastEndpoints](https://github.com/FastEndpoints/FastEndpoints).

Фреймворк ASP.NET Core изначально поддерживает OpenAPI. Для проектов на .NET 5 — .NET 8 для генерации документации использовались пакеты Swashbuckle. В версии .NET 9 и новее эту задачу будет выполнять пакет `Microsoft.AspNetCore.OpenApi`.

Стоит также изучить лучшие практики разработки и документирования API с использованием OpenAPI. Речь идет о четком и кратком описании операций; продуманной структуре путей, параметров и схем; точном документировании моделей запросов и ответов, включая коды состояния и сообщения об ошибках; поддержке нескольких версий API и фиксировании изменений в спецификации. Подробные рекомендации по проектированию API опубликованы на сайте [swagger.io/solutions/api-design/](https://swagger.io/solutions/api-design/).

Следует разобраться, как описывать схемы безопасности в документе OpenAPI. Стандарт поддерживает разные механизмы проверки подлинности — в том числе ключи API, базовую HTTP-аутентификацию, OAuth2 и OpenID Connect. Точное описание таких схем помогает защитить API и ясно показать пользователям, как

отправлять запросы с аутентификацией. Подробнее — на сайте [learn.openapis.org/specification/security.html](http://learn.openapis.org/specification/security.html).

Полезно также изучить, как с помощью спецификации OpenAPI создавать клиентские библиотеки и серверные заглушки. Это ускоряет разработку, поддерживает согласованность между реализацией и документацией API и упрощает создание клиентских приложений на разных языках. Дополнительные сведения — на сайтах [github.com/OpenAPITools/openapi-generator](https://github.com/OpenAPITools/openapi-generator) и [swagger.io/tools/swagger-codegen](https://swagger.io/tools/swagger-codegen).

Для разработчика .NET освоение OpenAPI в контексте работы с сервисами — это не только умение документировать API. Это еще и знакомство с инструментами для проектирования, тестирования, настройки безопасности и генерации кода. Такой подход — основанный на спецификациях — помогает создавать веб-сервисы, которые проще в использовании, понятнее и надежнее в долгосрочной перспективе.

## Документирование сервисов с минимальными API с помощью OpenAPI

Наиболее распространенные методы документирования веб-сервисов, реализованных с помощью минимальных API в ASP.NET Core и OpenAPI, приведены в табл. 6.3.

**Таблица 6.3.** Методы настройки сервисов ASP.NET Core для OpenAPI

| Метод                   | Описание                                                                                                                                    |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| AddEndpointsApiExplorer | Добавляет сервисы API Explorer — они необходимы библиотеке Swashbuckle для генерации спецификации OpenAPI из конечных точек минимальных API |
| AddSwaggerGen           | Настраивает Swashbuckle: создает базовый документ Swagger с именем v1, задает название и версию API                                         |
| UseSwagger              | Включает Swagger UI в среде разработки и указывает путь к сгенерированной спецификации OpenAPI                                              |
| UseSwaggerUI            | Регистрирует промежуточное ПО Swagger UI, включая дополнительные настройки через внедрение зависимостей                                     |

Как и в случае с документированием публичного API библиотеки классов, нужно включить генерацию документации для проекта:

```
<GenerateDocumentationFile>true</GenerateDocumentationFile>
```

Чтобы обогатить сгенерированную документацию OpenAPI, нужно настроить ее так, чтобы она включала XML-комментарии. Для этого измените вызов метода AddSwaggerGen в файле `Program.cs` и укажите путь к XML-файлу с комментариями:

```
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "Northwind API", Version = "v1" });

    // Указание пути к комментариям для Swagger JSON и UI
    string xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
    string xmlPath = Path.Combine(ApplicationContext.BaseDirectory, xmlFile);
    c.IncludeXmlComments(xmlPath);
});
```

Теперь шаг за шагом разберемся, как документировать веб-сервис на основе минимальных API ASP.NET Core с помощью OpenAPI.

1. В редакторе кода создайте проект веб-API и поместите его в решение Chapter06, указав следующие настройки:

- выберите шаблон проекта ASP.NET Core Web API (Веб-API ASP.NET Core) / webapi;
- в поле Solution name (Имя решения) укажите значение Chapter06;
- в поле Project name (Имя проекта) укажите значение Northwind.WebApi;
- в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
- установите флажок Configure for HTTPS (Настроить для HTTPS);
- снимите флажок Enable Docker (Включить поддержку контейнера);
- установите флажок Enable OpenAPI support (Включить поддержку OpenAPI);
- снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
- снимите флажок Use controllers (Использовать контроллеры);
- снимите флажок Enlist in .NET Aspire orchestration (Включить в список проект оркестрации .NET Aspire).



**Внимание!** Если вы работаете в JetBrains Rider, настройку для создания проекта веб-API с использованием минимальных API можно найти в разделе Advanced Settings (Расширенные настройки). Она расположена рядом с полем UseMinimalAPIs и задается с помощью флажка.

2. В файле проекта настройте обработку предупреждений как ошибок. Затем проверьте ссылки на пакеты, в частности на пакет Microsoft.OpenApi и компоненты для интеграции Swashbuckle с ASP.NET:

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

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

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="8.0.4" />
  <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0" />
</ItemGroup>

</Project>
```



Актуальные версии пакетов доступны по адресам [www.nuget.org/packages/Swashbuckle.AspNetCore](http://www.nuget.org/packages/Swashbuckle.AspNetCore) и [www.nuget.org/packages/Microsoft.AspNetCore.OpenApi](http://www.nuget.org/packages/Microsoft.AspNetCore.OpenApi).

3. Выполните сборку проекта Northwind.WebApi, чтобы восстановить все зависимости.
4. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5061 для протокола HTTPS и 5062 — для HTTP:

```
"profiles": {
  ...
  "https": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "launchUrl": "swagger",
    "applicationUrl": "https://localhost:5061;http://localhost:5062",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
}
```



**Внимание!** Часто я использую номера портов, начинающиеся с 5000, где две средние цифры соответствуют номеру главы, а завершающая — 1 для HTTPS и 2 для HTTP. Однако имейте в виду: порты 5060 и 5061 (и TCP, и UDP) зарезервированы организацией IANA для установления сеанса (Session Initiation Protocol, SIP).

5. В файле `Program.cs` найдите выражения, которые настраивают OpenAPI, в частности `AddEndpointsApiExplorer` и `AddSwaggerGen`:

```
var builder = WebApplication.CreateBuilder(args);

// Добавление сервисов в контейнер.
// Подробнее о настройке Swagger и OpenAPI – на сайте
// aka.ms/aspnetcore/swashbuckle.
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

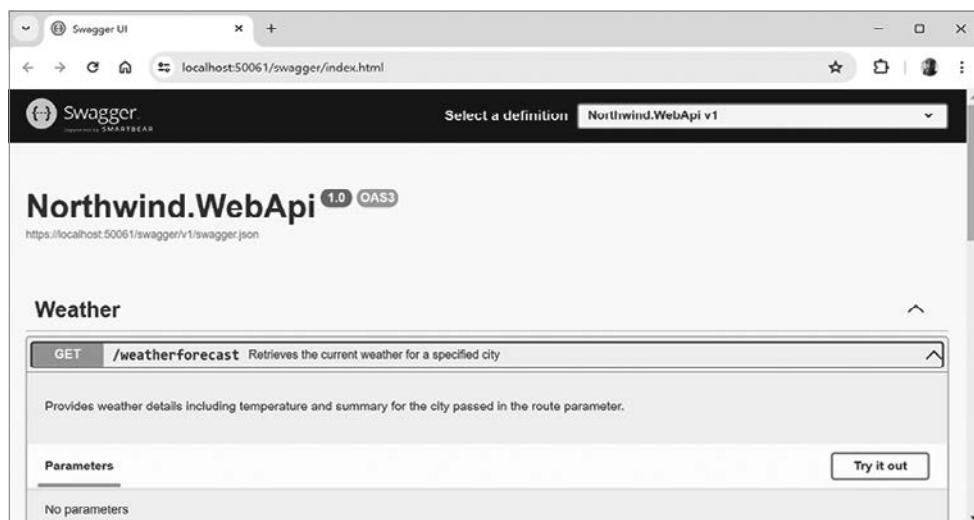
// Настройка конвейера обработки HTTP-запросов
if (app.Environment.IsDevelopment())
```

```
{  
    app.UseSwagger();  
    app.UseSwaggerUI();  
}  
  
app.UseHttpsRedirection();  
  
var summaries = new[]  
{  
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",  
    "Hot", "Sweltering", "Scorching"  
};  
  
app.MapGet("/weatherforecast", () =>  
{  
    var forecast = Enumerable.Range(1, 5).Select(index =>  
        new WeatherForecast  
        (  
            DateOnly.FromDateTime(DateTime.Now.AddDays(index)),  
            Random.Shared.Next(-20, 55),  
            summaries[Random.Shared.Next(summaries.Length)]  
        ))  
        .ToArray();  
    return forecast;  
})  
.WithName("GetWeatherForecast")  
.WithOpenApi();  
  
app.Run();  
  
internal record WeatherForecast(DateOnly Date, int TemperatureC, string?Summary)  
{  
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);  
}
```

6. Между вызовами методов `WithName` и `WithOpenApi` добавьте выражения для дополнительного документирования конечной точки прогноза погоды:

```
.WithName("GetWeatherForecast")  
.Produces (StatusCodes.Status200OK)  
.WithTags("Weather")  
.WithSummary("Retrieves the current weather for a specified city")  
.WithDescription("Provides weather details including temperature and summary  
for the city passed in the route parameter.")  
.WithOpenApi();
```

7. Запустите проект `Northwind.WebApi` без отладки.
8. На веб-странице `Swagger UI` щелкните кнопкой мыши на GET-запросе для сервиса прогноза погоды и изучите дополнительную документацию — краткие сведения и описание (рис. 6.7).
9. Закройте браузер и остановите веб-сервер.



**Рис. 6.7.** Документация Swagger UI

Теперь посмотрим, как можно создавать визуальную документацию с помощью диаграмм *Mermaid*.

## Визуальное документирование с помощью диаграмм Mermaid

Диаграмма — один из самых наглядных инструментов, позволяющих объяснить концепцию, особенно когда речь идет о людях без технического фона, например менеджерах. Умение создавать понятные визуализации помогает яснее излагать идеи на совещаниях и укрепляет вашу репутацию среди тех, кто принимает кадровые решения.

Благодаря инструменту вроде *Mermaid* не требуется владеть графическим редактором, чтобы подготовить наглядные и полезные схемы. Такие диаграммы помогают команде лучше понимать архитектуру решений или принятые проектировочные подходы. *Mermaid* особенно удобен для разработчиков на платформе .NET, так как система управления исходным кодом GitHub — как и другие решения компании Microsoft — поддерживает визуализацию диаграмм *Mermaid* прямо в репозиториях.

Чтобы вставить диаграмму в документацию, в файле с расширением .md определите блок кода и укажите язык *mermaid*:

```
```mermaid
```

```
```
```

Синтаксис **Mermaid** простой и интуитивно понятный, поэтому разработчики могут быстро добавлять диаграммы в техническую документацию, повышая ее наглядность и читаемость.

Инструмент Mermaid поддерживает множество типов диаграмм и графиков. Некоторые из них приведены в табл. 6.4.

**Таблица 6.4.** Распространенные типы диаграмм Mermaid

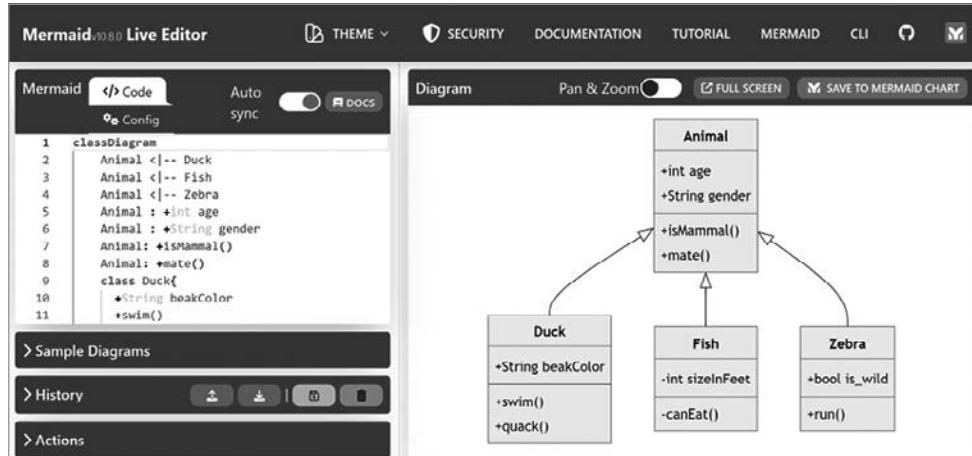
| Тип                           | Описание                                                                                                                                                                                        |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>flowchart, graph</code> | Блок-схемы и графы, помогающие визуализировать этапы и точки принятия решений в процессе или рабочем потоке. Благодаря таким диаграммам сложные процессы становятся понятными с первого взгляда |
| <code>sequenceDiagram</code>  | Показывают взаимодействие объектов в определенной последовательности. Хорошо подходят для отображения потока сообщений, событий или действий между частями системы                              |
| <code>classDiagram</code>     | Отражают структуру системы: классы, их атрибуты, методы и связи между объектами                                                                                                                 |
| <code>stateDiagram</code>     | Иллюстрируют возможные состояния сущности и переходы между ними. Используются для моделирования поведения системы или приложения                                                                |
| <code>erDiagram</code>        | Применяются при проектировании и моделировании БД для отображения связей между сущностями и данными                                                                                             |
| <code>gantt</code>            | Используются в управлении проектами. Визуализируют задачи, зависимости и длительность этапов проекта по временной шкале                                                                         |
| <code>journey</code>          | Отображают путь пользователя при взаимодействии с системой                                                                                                                                      |
| <code>gitGraph</code>         | Отображают ветвления в истории репозитория Git                                                                                                                                                  |
| <code>pie</code>              | Служат для наглядного представления распределения и пропорций данных                                                                                                                            |
| <code>mindmap</code>          | Используются для фиксации и связи идей при мозговом штурме                                                                                                                                      |

## Рендеринг диаграмм Mermaid

Среди основных инструментов для рендеринга диаграмм **Mermaid** можно выделить следующие.

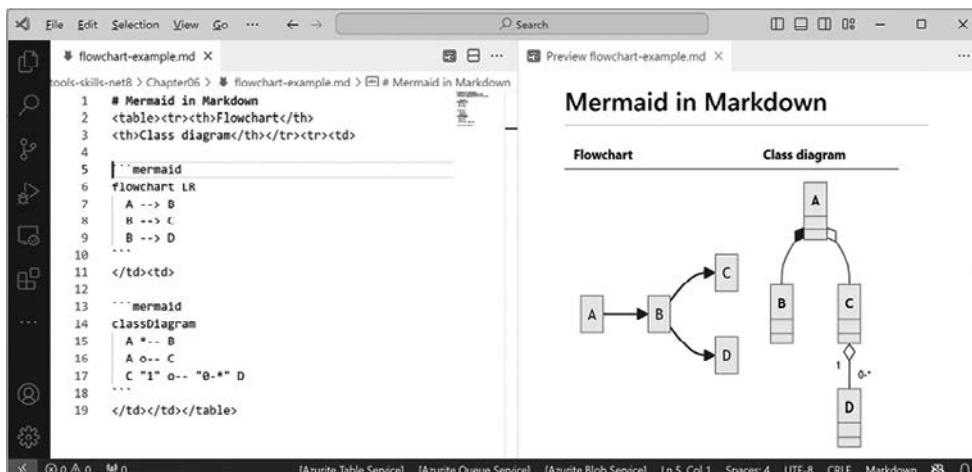
- **Mermaid Live** — сайт по адресу `mermaid.live`, на котором можно визуализировать диаграммы **Mermaid** в интерактивном режиме. Показан на рис. 6.8. Удобен для быстрых демонстраций на совещаниях, где присутствуют как технические, так

и нетехнические участники. Диаграмму можно изменять на лету, и все сразу видят результат. Это помогает оперативно выявлять и устранять недопонимания — без затяжной переписки по электронной почте.



**Рис. 6.8.** Сайт Mermaid Live с интерактивной диаграммой

- *Markdown Preview Mermaid Support* — расширение для Visual Studio Code с более чем миллионом загрузок. Добавляет поддержку диаграмм Mermaid в предварительный просмотр Markdown-файлов, а также в ячейки Markdown в блокнотах (рис. 6.9). Идентификатор расширения — `bierner.markdown-mermaid`.



**Рис. 6.9.** Расширение Markdown Preview Mermaid Support для Visual Studio Code

- *Mermaid CLI* — инструмент командной строки для преобразования файлов *Mermaid* в графические форматы, такие как PNG, SVG и PDF. Последнюю версию можно найти на сайте [github.com/mermaid-js/mermaid-cli/](https://github.com/mermaid-js/mermaid-cli/). Если установлен менеджер пакетов npm, используйте следующую команду для установки:

```
npm install -g @mermaid-js/mermaid-cli
```



Инструкции по загрузке и установке Node.js и npm приведены на сайте [docs.npmjs.com/downloading-and-installing-node-js-and-npm](https://docs.npmjs.com/downloading-and-installing-node-js-and-npm). Для Windows можно использовать менеджер версий nvm (Node Version Manager), доступный по адресу [github.com/coreybutler/nvm-windows](https://github.com/coreybutler/nvm-windows).

Теперь перейдем к нескольким простым примерам распространенных типов диаграмм и графиков.

## Создание блок-схемы с помощью Mermaid

Блок-схемы особенно удобны для объяснения логики даже тем, кто не имеет технического фона. Например, если вы решаете, какой электромобиль (EV) выбрать, можно построить схему принятия решений — как показано на рис. 6.10.

```
graph TD
    A((Start)) --> B{Budget?}
    B -- "Under $30k" --> C[Used EV]
    B -- "Between $30k-$50k" --> D[New Mid-Range EV]
    B -- "Over $50k" --> E[New Luxury EV]
    C --> F{Range Requirement?}
    D --> F
    E --> F
    F -- "Under 200 miles" --> G[City Car]
    F -- "200-300 miles" --> H[All-Purpose Vehicle]
    F -- "Over 300 miles" --> I[Long Range Vehicle]
    G --> J{Charging at Home?}
    H --> J
    I --> J
    J -- "Yes" --> K[Proceed with Purchase]
    J -- "No" --> L[Consider Charging Options]
    K --> M{Public Charging Available?}
    M -- "Yes" --> N[Proceed with Purchase]
    M -- "No" --> O[Reevaluate Requirements]
```

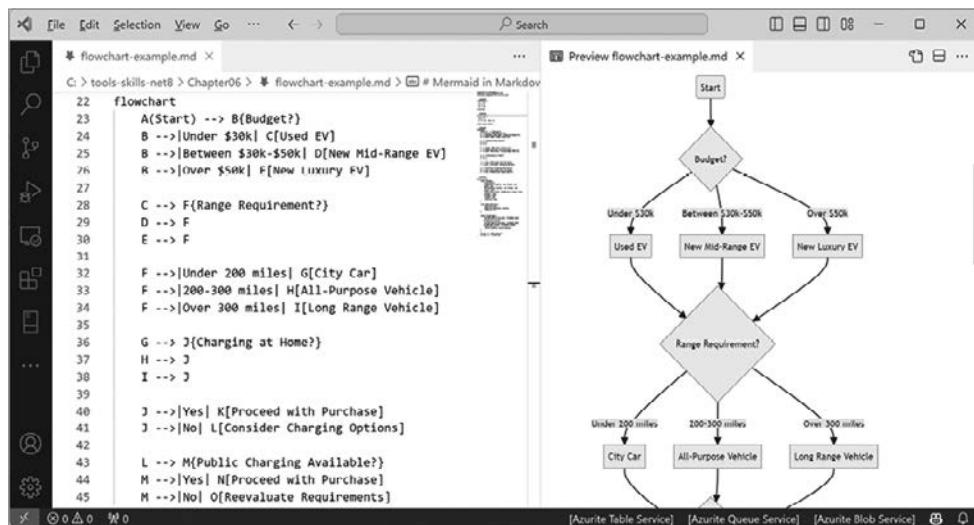


Рис. 6.10. Блок-схема для принятия решения о покупке EV

## Создание диаграммы классов с помощью Mermaid

Один из часто используемых типов диаграмм среди разработчиков — диаграмма классов. Например, можно наглядно представить отношения наследования между абстрактным классом `Stream` и производными классами `FileStream` и `MemoryStream`, как показано на рис. 6.11.

```

classDiagram
    class Stream {
        <<abstract>>
        +Read(byte[] buffer, int offset, int count) int
        +Write(byte[] buffer, int offset, int count) void
        +Seek(long offset, SeekOrigin origin) long
        +Flush() void
        +Close() void
        -Length long
        -Position long
    }

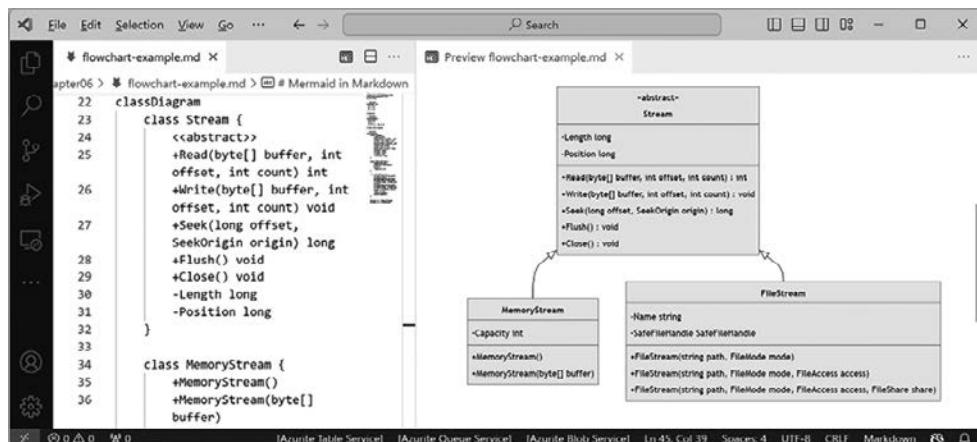
    class MemoryStream {
        +MemoryStream()
        +MemoryStream(byte[] buffer)
        -Capacity int
    }
  
```

```

class FileStream {
    +FileStream(string path, FileMode mode)
    +FileStream(string path, FileMode mode, FileAccess access)
    +FileStream(string path, FileMode mode, FileAccess access, FileShare share)
    -Name string
    -SafeFileHandle SafeFileHandle
}

Stream <|-- MemoryStream
Stream <|-- FileStream

```



**Рис. 6.11.** Диаграмма классов, отражающая наследование от класса Stream

На этой диаграмме можно выделить несколько ключевых моментов.

- Класс Stream обозначен как абстрактный. Он отражает общие методы (Read, Write, Seek, Close) и свойства (Length, Position), связанные с операциями над потоком.
- Классы MemoryStream и FileStream показаны как конкретные реализации. На диаграмме также представлены их конструкторы.
- Стрелки наследования указывают, что MemoryStream и FileStream наследуются от класса Stream.

Подробнее о создании диаграмм Mermaid речь пойдет в следующих главах — при изучении *унифицированного языка моделирования (UML)* и его распространенных диаграмм. В частности, в главе 18 будет показано, как документировать последовательности действий в системе и применять *предметно-ориентированное проектирование (Domain-Driven Design, DDD)* для построения диаграмм предметной области.

## Преобразование диаграмм Mermaid в формат SVG

В завершение преобразуем файл Markdown с несколькими диаграммами Mermaid в формат SVG. Такой файл можно просматривать в любом браузере, даже если он не поддерживает рендеринг Mermaid, как это делают, например, GitHub и GitLab.

1. В папке `Chapter06` создайте файл Markdown с несколькими диаграммами Mermaid. Пример доступен по адресу [github.com/markjprice/tools-skills-net8/blob/main/code/Chapter06/mermaid-examples.md](https://github.com/markjprice/tools-skills-net8/blob/main/code/Chapter06/mermaid-examples.md).
2. Откройте терминал в папке `Chapter06` и выполните преобразование Markdown-файла с блоками Mermaid в SVG-файл:

```
mmdc -i mermaid-examples.md -o output.md
```

3. Изучите результат выполнения команды — в выводе будет указано, что файл успешно создан и сохранен в формате SVG:

```
Found 4 mermaid charts in Markdown input
✓ ./output-1.svg
✓ ./output-2.svg
✓ ./output-4.svg
✓ ./output-3.svg
✓ output.md
```

4. Откройте файл `output.md`. Теперь он содержит ссылки на файлы SVG вместо исходных блоков кода Mermaid:

```
# Mermaid в Markdown
<table><tr><th>Flowchart</th>
<th>Class diagram</th></tr><tr><td>
![diagram](./output-1.svg)
</td><td>
![diagram](./output-2.svg)
</td></td></table>

![diagram](./output-3.svg)
![diagram](./output-4.svg)
```



Шпаргалку по Mermaid можно скачать по адресу [jojozhuang.github.io/tutorial/mermaid-cheat-sheet/](https://jojozhuang.github.io/tutorial/mermaid-cheat-sheet/).

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 6.1. Материалы, доступные только онлайн

Подробности о документации XML см. на сайте [learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/xml/doc/](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/xml/doc/).

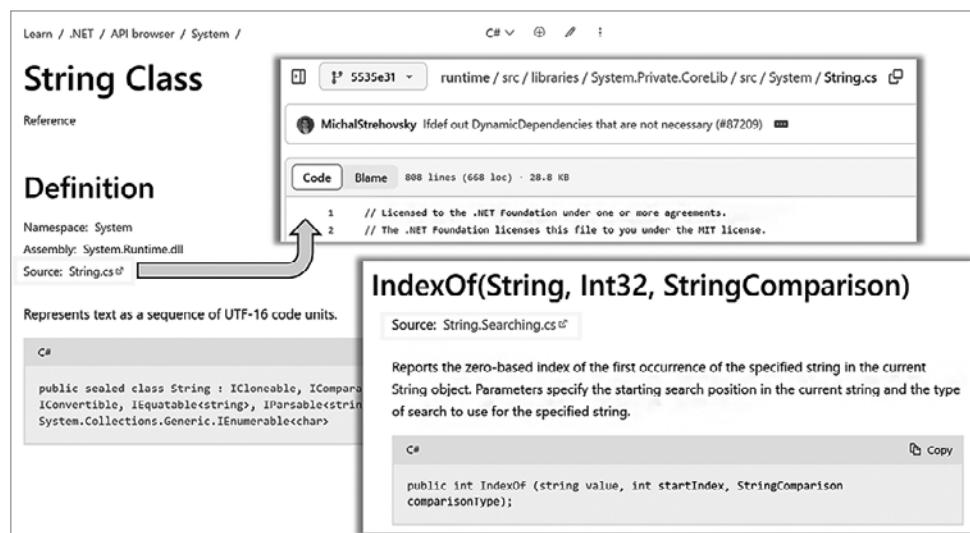
Список рекомендуемых тегов для XML-документации см. по адресу [learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/xml/doc/recommended-tags](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/xml/doc/recommended-tags).

Полное руководство по Markdown опубликовано на сайте [www.markdownguide.org](http://www.markdownguide.org).

Введение в Mermaid и примеры — ищите на сайте [mermaid.js.org/intro/getting-started.html](https://mermaid.js.org/intro/getting-started.html).

Дополнительные материалы по Mermaid доступны в репозитории [github.com/JakeSteam/Mermaid](https://github.com/JakeSteam/Mermaid).

При работе с документацией по API часто хочется заглянуть в исходный код. В проектах .NET с поддержкой Source Link в метаданные включаются ссылки на соответствующие участки кода в публичных репозиториях GitHub. Например, на странице документации класса `String` добавлена ссылка `Source`, а в описании метода `IndexOf` — отдельная ссылка на другой исходный файл (рис. 6.12).



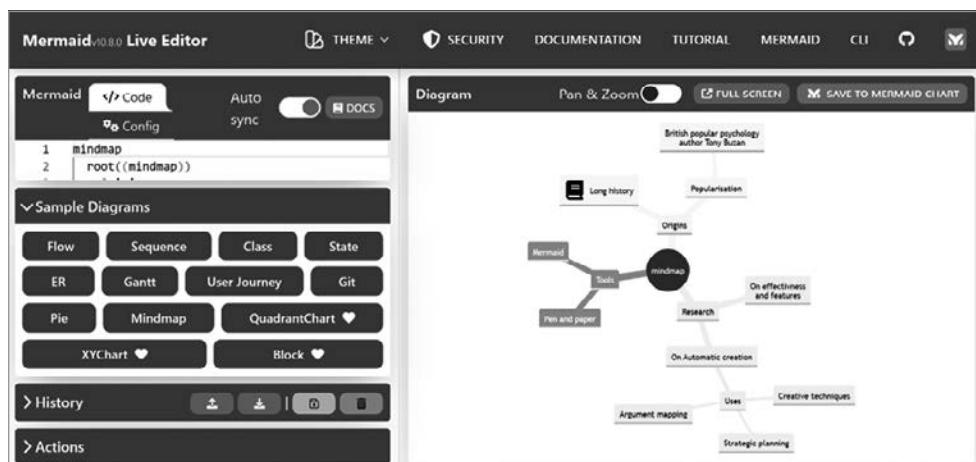
**Рис. 6.12.** Примеры ссылок на исходный код в официальной документации

Подробности об этой функции описаны в статье [devblogs.microsoft.com/dotnet/dotnet-docs-link-to-source-code/](https://devblogs.microsoft.com/dotnet/dotnet-docs-link-to-source-code/).

## Упражнение 6.2. Практические задания

Примеры XML-документации доступны по адресу [learn.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/examples](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/examples).

Изучите примеры диаграмм на сайте Mermaid Live — [mermaid.live/](https://mermaid.live/) (рис. 6.13).



**Рис. 6.13.** Примеры диаграмм Mermaid Live

## Упражнение 6.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Как правильно оформлять комментарии в исходном коде?
2. Как задокументировать параметр метода с помощью XML-комментария?
3. Можно ли применить стили ко всплывающей подсказке с XML-комментарием?
4. Что представляет собой инструмент DocFX и для чего он используется?
5. Как отобразить наследование на диаграмме классов, созданной с помощью инструмента Mermaid?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 6.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-6---documenting-your-code-apis-and-services](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-6---documenting-your-code-apis-and-services).

## Резюме

В этой главе вы узнали, как:

- документировать исходный код с помощью комментариев;
- оформлять документацию для публичных API в библиотеках классов;
- описывать сервисы с использованием OpenAPI;
- создавать наглядную документацию с помощью диаграмм Mermaid.

В следующей главе речь пойдет о некоторых распространенных типах из состава .NET, которые используются для анализа кода путем рефлексии, работы с атрибутами и чтения их значений. Вы также познакомитесь с деревьями выражений и узнаете, как создавать генераторы исходного кода.

# 7

## Наблюдение и динамическое изменение выполнения кода

В этой главе рассматриваются распространенные типы из состава .NET, которые используются для рефлексии кода, применения и чтения атрибутов, работы с деревьями выражений и создания генераторов исходного кода. Эти технологии открывают широкие возможности — от анализа и модификации кода до изменения поведения программы во время выполнения. Благодаря им можно повысить гибкость и адаптивность приложений.

*Рефлексия* — мощный механизм .NET, с помощью которого программа может анализировать и изменять собственную структуру. Она позволяет получать сведения о сборках, модулях и типах, а также динамически создавать и вызывать типы и методы.

*Деревья выражений* представляют код в виде структуры данных, где каждый узел соответствует отдельному выражению — например, вызову метода или арифметической операции. Такой подход позволяет динамически анализировать, изменять и выполнять код.

*Генераторы исходного кода*, впервые представленные в .NET 5, создают дополнительные файлы на этапе компиляции. Они запускаются во время сборки, анализируют структуру программы и автоматически добавляют новые исходные файлы, которые компилируются вместе с остальным кодом.

Все темы, затронутые в этой главе, обычно относят к продвинутому уровню, поэтому не стоит переживать, если вы не вникнете во все детали с первого раза. Для таких инструментов, как рефлексия, большинству разработчиков достаточно понимать общие принципы и иметь представление о том, в каких ситуациях они могут быть полезны.

После прочтения этой главы вы сможете, например, на собеседовании сказать: «Я не эксперт по рефлексии, деревьям выражений или генераторам исходного кода, но я запускал код и посмотрел, как это работает. Если потребуется реализовать более сложный сценарий — разберусь и справлюсь».

## Работа с рефлексией и атрибутами

*Рефлексия* — это механизм, позволяющий программе анализировать и взаимодействовать со своей структурой во время выполнения. Вот основные преимущества ее использования.

- **Динамический доступ к типам.** С помощью рефлексии можно обращаться к типам во время выполнения, что особенно полезно в архитектурах с плагинами, где функциональность добавляется без повторной компиляции основного приложения.
- **Позднее связывание.** Методы можно вызывать, а свойства — считывать и изменять динамически, даже если их тип неизвестен на этапе компиляции. Такой подход используется, например, при работе с API, структура которых заранее неизвестна.
- **Проверка метаданных.** Рефлексия позволяет получить сведения о типах и их членах. Это особенно полезно для универсальных библиотек и инструментов — таких как системы сериализации или ORM-фреймворки, которым нужно обрабатывать произвольные типы.

Начнем с изучения проверки метаданных.

### Метаданные в сборках .NET

Сборка .NET может содержать до четырех основных компонентов.

- **Метаданные и манифест сборки** — включают имя, версию, информацию о файле и сборках, на которые есть ссылки.
- **Метаданные о типах** — содержат сведения о типах, их членах и структуре.
- **Код в промежуточном представлении (IL, Intermediate Language)** — включает реализации методов, свойств, конструкторов и других элементов. Исходный код (например, на C# или F#) компилируется в IL-команды с помощью компилятора Roslyn и сохраняется в сборке.
- **Встроенные ресурсы** (необязательно) — изображения, строки, JavaScript-сценарии и другие данные.

Метаданные описывают содержимое сборки — они либо автоматически формируются из кода (например, информация о типах и их членах), либо добавляются вручную с помощью атрибутов.

Атрибуты можно применять на разных уровнях:

```
// Атрибут уровня сборки
[assembly: AssemblyTitle("Working with reflection and attributes")]

// Атрибут уровня типа
[Serializable]
public class Person
```

```
{  
// Атрибут уровня члена  
[Obsolete("Deprecated: use Run instead.")]  
public void Walk()  
{  
...  
}
```

Программирование с использованием атрибутов часто применяется в таких технологиях, как ASP.NET Core, — например, для настройки маршрутизации, проверки подлинности, авторизации и кэширования.

## Версионирование сборок

Знание номера версии сборки в .NET играет важную роль при разработке, сопровождении и развертывании приложений.

Приложения .NET для корректной работы часто используют несколько библиотек (сборок), каждая из которых может существовать в нескольких версиях — с различными возможностями, исправлениями и поведением. Отслеживание версий позволяет убедиться, что вы работаете с актуальной сборкой и учитываете ее особенности.

Разные версии одной и той же сборки могут содержать ошибки или влиять на производительность. Возможность определить, какая версия загружена, помогает быстрее выявить проблему и проверить, была ли она решена в более новой версии — например, по примечаниям к релизу.

Точное указание версии также важно для обеспечения совместимости, особенно при развертывании приложений в разных средах или при обновлении существующих систем.

В .NET номер версии используется при связывании и загрузке сборок. *Общезыкожая исполняющая среда (CLR)* ориентируется на номера версий, чтобы правильно разрешать ссылки и загружать нужные сборки во время выполнения. Это часть механизма идентификации сборки, который помогает избежать конфликтов между различными версиями компонентов — так называемого DLL-кошмара.

В .NET версии сборок обычно записываются в виде комбинации из трех чисел, например 8.0.4. Иногда добавляются дополнительные значения, образуя более длинный формат, например 4.0.5.7628.345.

Если придерживаться правил семантического версионирования, эти числа означают следующее.

- *Мажорная версия* — коренные, потенциально несовместимые изменения.
- *Минорная версия* — новые функции и улучшения, которые не нарушают обратную совместимость.
- *Патч-версия* — небольшие исправления и доработки без изменения поведения API.

Дополнительно могут использоваться:

- *предварительный релиз* — пометка о нестабильной версии, не предназначенный для коммерческого использования;
- *номер сборки* — используется для промежуточных и «ночных» сборок (nightly builds).



Иногда вместо классического семантического формата используется версионирование по году. Например, среда разработки Rider на момент написания книги имеет версию 2024.1.2, что означает первый функциональный релиз 2024 года и второй патч. Также у нее есть номер сборки — #RD241.15989.179.



**Дельный совет:** по умолчанию сборки .NET имеют версию 1.0.0.0, если не указано иное. При каждом новом релизе версию следует обновлять и придерживаться семантического подхода, описанного на сайте semver.org.

## Чтение метаданных сборки

Теперь, когда вы знаете, какие метаданные могут храниться в сборке .NET — например, информация о версии, — давайте научимся читать эти данные динамически во время выполнения. Для получения доступа к части метаданных, содержащихся в сборке, используются *атрибуты*.

Рассмотрим, как работать с атрибутами.

1. В редакторе кода создайте проект Console App (Консольное приложение) / console, присвойте ему имя WorkingWithReflection и добавьте в решение Chapter07.
2. В файле проекта статически и глобально импортируйте класс `Console`:

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

3. В файле `Program.cs` удалите все содержимое и добавьте следующее:
  - импортируйте пространство имен для работы с рефлексией;
  - получите сборку консольного приложения и выведите ее имя и местоположение;
  - получите все атрибуты уровня сборки и выведите их типы:

```
using System.Reflection; // Для работы с классом Assembly

WriteLine("Assembly metadata:");
Assembly? assembly = Assembly.GetEntryAssembly();

if (assembly is null)
{
  WriteLine("Failed to get entry assembly.");
```

```

        return; // Выход из приложения
    }

    WriteLine($" Full name: {assembly.FullName}");
    WriteLine($" Location: {assembly.Location}");
    WriteLine($" Entry point: {assembly.EntryPoint?.Name}");

    IEnumerable<Attribute> attributes = assembly.GetCustomAttributes();
    WriteLine($" Assembly-level attributes:");
    foreach (Attribute a in attributes)
    {
        WriteLine($" {a.GetType()}");
    }
}

```

4. Выполните код. В результате вы увидите следующий вывод:

```

Assembly metadata:
  Full name: WorkingWithReflection, Version=1.0.0.0, Culture=neutral,
  PublicKeyToken=null
  Location: C:\tools-skills-net8\Chapter07\WorkingWithReflection\bin\Debug\
net8.0\WorkingWithReflection.dll
  Entry point: <Main>$

  Assembly-level attributes:
    System.Runtime.CompilerServices.CompilationRelaxationsAttribute
    System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
    System.Diagnostics.DebuggableAttribute
    System.Runtime.Versioning.TargetFrameworkAttribute
    System.Reflection.AssemblyCompanyAttribute
    System.Reflection.AssemblyConfigurationAttribute
    System.Reflection.AssemblyFileVersionAttribute
    System.Reflection.AssemblyInformationalVersionAttribute
    System.Reflection.AssemblyProductAttribute
    System.Reflection.AssemblyTitleAttribute

```

По соображениям безопасности и доверия полное имя сборки должно однозначно идентифицировать ее. Оно состоит из следующих значений:

- имя, например `WorkingWithReflection`;
- версия, например `1.0.0.0`;
- культура, например `neutral`;
- токен открытого ключа (может быть `null`).

Теперь, когда мы разобрались с атрибутами для пометки сборки, можно запросить их значения.

5. В нижней части файла `Program.cs` добавьте выражения для получения атрибутов `AssemblyInformationalVersionAttribute` и `AssemblyCompanyAttribute`, а затем выведите их значения:

```

AssemblyInformationalVersionAttribute? version = assembly
    .GetCustomAttribute<AssemblyInformationalVersionAttribute>();

```

```
WriteLine($" Version: {version?.InformationalVersion}");  
AssemblyCompanyAttribute? company = assembly  
.GetCustomAttribute<AssemblyCompanyAttribute>();  
WriteLine($" Company: {company?.Company}");
```

6. Запустите код. Результат будет следующим:

```
Version: 1.0.0  
Company: WorkingWithReflection
```



Если не указать версию, по умолчанию будет установлена версия 1.0.0. Если не указать компанию, по умолчанию будет использовано имя сборки. Теперь давайте явно зададим эту информацию.

7. В файл `WorkingWithReflection.csproj` проекта добавьте свойства версии и компании:

```
<Project Sdk="Microsoft.NET.Sdk">  
  
<PropertyGroup>  
  <OutputType>Exe</OutputType>  
  <TargetFramework>net8.0</TargetFramework>  
  <ImplicitUsings>enable</ImplicitUsings>  
  <Nullable>enable</Nullable>  
  <Version>8.0.1</Version>  
  <Company>Packt Publishing</Company>  
</PropertyGroup>
```



Ранее в .NET Framework эти значения указывались с помощью атрибутов в исходном коде C#.

```
[assembly: AssemblyCompany("Packt Publishing")]  
[assembly: AssemblyInformationalVersion("1.3.0")]
```

Компилятор Roslyn, используемый в .NET, автоматически устанавливает эти атрибуты, поэтому старый способ уже не применим. Вместо этого они должны быть заданы в файле проекта.

8. Выполните код. В результате вы увидите следующий вывод:

```
Assembly metadata:  
  Full name: WorkingWithReflection, Version=8.0.1.0, Culture=neutral,  
  PublicKeyToken=null  
  ...  
  Version: 8.0.1  
  Company: Packt Publishing
```

## Создание пользовательских атрибутов

Вы можете определять собственные атрибуты, унаследовав их от класса `Attribute`.

1. В папке проекта `WorkingWithReflection` создайте файл класса `CoderAttribute.cs`.
2. В этом файле удалите все содержимое и определите класс-атрибут, который можно использовать для пометки классов или методов. Атрибут должен содержать два свойства: имя программиста и дату последнего изменения кода:

```
namespace Packt.Shared;

[AttributeUsage(
    AttributeTargets.Class | AttributeTargets.Method,
    AllowMultiple = true)]
public class CoderAttribute(
    string coder, string lastModified) : Attribute
{
    public string Coder { get; set; } = coder;

    public DateTime LastModified { get; set; }
        = DateTime.Parse(lastModified);
}
```

3. В папке проекта `WorkingWithReflection` создайте файл класса `Animal.cs`.
4. В файле `Animal.cs` удалите все содержимое, затем определите класс с методом и пометьте метод атрибутом `Coder`, указав данные о двух программастах:

```
namespace Packt.Shared;

public class Animal
{
    [Coder("Mark Price", "22 June 2024")]
    [Coder("Johnni Rasmussen", "13 July 2024")]
    public void Speak()
    {
        WriteLine("Woof...");
    }
}
```

5. В файле `Program.cs` импортируйте пространство имен для работы с пользовательским атрибутом:

```
using Packt.Shared; // Подключение класса CoderAttribute
```

6. В нижней части файла `Program.cs` добавьте операторы для получения типов в текущей сборке, перечисления их членов, поиска атрибутов `Coder` для этих членов и вывода информации о каждом найденном атрибуте:

```
WriteLine();
WriteLine("* Types:");
Type[] types = assembly.GetTypes();
```

```
foreach (Type type in types)
{
    WriteLine();
    WriteLine($"Type: {type.FullName}");
    MemberInfo[] members = type.GetMembers();

    foreach (MemberInfo member in members)
    {
        WriteLine($"{member.MemberType}: {member.Name} ({member.DeclaringType?.Name})");

        IOrderedEnumerable<CoderAttribute> coders =
            member.GetCustomAttributes<CoderAttribute>()
                .OrderByDescending(c => c.LastModified);

        foreach (CoderAttribute coder in coders)
        {
            WriteLine($"-> Modified by {coder.Coder} on {coder.LastModified.ToShortDateString()}");
        }
    }
}
```

7. Выполните код. В выводе отобразится информация о програмистах, участвовавших в работе над методом, к примеру:

```
* Types:

Type: Program
Method: GetType (Object)
Method: ToString (Object)
Method: Equals (Object)
Method: GetHashCode (Object)
Constructor: .ctor (Program)

Type: Packt.Shared.Animal
Method: Speak (Animal)
-> Modified by Jhnni Rasmussen on 13/07/2024
-> Modified by Mark Price on 22/06/2024
Method: GetType (Object)
Method: ToString (Object)
Method: Equals (Object)
Method: GetHashCode (Object)
Constructor: .ctor (Animal)

Type: Packt.Shared.CoderAttribute
Method: get_Coder (CoderAttribute)
Method: set_Coder (CoderAttribute)
Method: get_LastModified (CoderAttribute)
Method: set_LastModified (CoderAttribute)
Method: Equals (Attribute)
Method: GetHashCode (Attribute)
Method: get_TypeId (Attribute)
Method: Match (Attribute)
Method: IsDefaultAttribute (Attribute)
```

```

Method: GetType (Object)
Method: ToString (Object)
Constructor: .ctor (CoderAttribute)
Property: Coder (CoderAttribute)
Property: LastModified (CoderAttribute)
Property: TypeId (Attribute)

Type: Program+<>c
Method: GetType (Object)
Method: ToString (Object)
Method: Equals (Object)
Method: GetHashCode (Object)
Constructor: .ctor (<>c)
Field: <>9 (<>c)
Field: <>9_0_0 (<>c)

```

## Фильтрация автоматически сгенерированных типов

Во время выполнения вы могли заметить странно названные типы, такие как `Program+<>c`. Это внутренние недокументированные конструкции, создаваемые компилятором. Символы `<>` указывают на то, что это вспомогательный тип, а `c` обозначает класс отображения. Такие типы используются, например, для лямбда-выражений или анонимных методов. Эти детали не являются частью пользовательского кода, и их можно спокойно исключить из вывода. Чтобы скрыть такие элементы, добавьте в приложение фильтрацию по атрибуту `CompilerGeneratedAttribute`.



**Подсказка:** импортируйте пространство имен `System.Runtime.CompilerServices`, чтобы использовать этот атрибут:

```
using System.Runtime.CompilerServices; // Подключение класса
// CompilerGeneratedAttribute
```

## Пометка устаревших членов

С временем может возникнуть необходимость изменить структуру типов и их членов — например, в рамках рефакторинга. При этом важно сохранить обратную совместимость. Чтобы аккуратно подсказать разработчикам, что следует перейти на более новые реализации, можно использовать атрибут `[Obsolete]` и пометить им устаревшие элементы.

Рассмотрим пример.

1. В файле `Animal.cs` добавьте новый метод `SpeakBetter`, а метод `Speak` отметьте как устаревший:

```
public void SpeakBetter()
{
    WriteLine("Wooooooooof...");
}
```

```
[Coder("Mark Price", "22 August 2024")]
[Coder("Johnni Rasmussen", "13 September 2024")]
[Obsolete($"use {nameof(SpeakBetter)} instead.")]
public void Speak()
{
    WriteLine("Woof... ");
}
```

2. В файле `Program.cs` измените код так, чтобы он находил методы, помеченные как устаревшие:

```
foreach (MemberInfo member in members)
{
    ObsoleteAttribute? obsolete =
        member.GetCustomAttribute<ObsoleteAttribute>();

    WriteLine($"{member.MemberType}: {member.Name} ({{
        member.DeclaringType?.Name})} {
        obsolete is null ? "" : "Obsolete! " + obsolete.Message}}");
```

3. Выполните код. Вывод будет примерно следующим:

```
Type: Packt.Shared.Animal
Method: Speak (Animal) Obsolete! use SpeakBetter instead.
-> Modified by Johnni Rasmussen on 13/07/2024
-> Modified by Mark Price on 22/06/2024
Method: SpeakBetter (Animal)
Method: GetType (Object)
Method: ToString (Object)
Method: Equals (Object)
Method: GetHashCode (Object)
Constructor: .ctor (Animal)
```

## Динамическая загрузка сборок и выполнение методов

В большинстве случаев, если проект .NET должен обращаться к другой сборке, это указывается заранее — через ссылку на проект или пакет. Благодаря этому компилятор точно знает, какие сборки необходимо загрузить в память во время выполнения.

Но бывают ситуации, когда заранее определить нужную сборку невозможно. Например, в текстовом редакторе функция слияния данных с шаблоном (Mail merge) может быть вынесена в отдельную сборку, которая загружается только тогда, когда пользователь действительно запускает эту функцию. Нет смысла держать ее в памяти постоянно.

Другой пример — приложение с поддержкой подключаемых модулей (плагинов), которые могут быть разработаны сторонними авторами. Такое расширяемое поведение реализовано, например, в программах Visual Studio, Visual Studio Code и Rider.

Для поддержки такого подхода в .NET можно использовать класс `AssemblyLoadContext`. Он позволяет загружать сборки динамически, выполнять в них методы, а затем выгружать их из памяти, освобождая ресурсы. Это особенно полезно, когда нужно уменьшить объем используемой памяти.

Начиная с версии .NET 7 динамический доступ к членам типов, таким как методы или свойства, стал работать заметно быстрее, особенно если обращение повторяется к одному и тому же члену.

Теперь посмотрим, как загрузить сборку динамически, создать экземпляр класса и обратиться к его членам.

1. В редакторе кода создайте проект **Class Library** (Библиотека классов) / `classlib`, присвойте ему имя `DynamicLoadAndExecute.Library` и добавьте в решение `Chapter07`.
2. В файле `DynamicLoadAndExecute.Library.csproj` настройте сборку так, чтобы все предупреждения обрабатывались как ошибки. Затем выполните два глобальных импорта:
  - статический импорт класса `Console`;
  - импорт пространства имен для работы с рефлексией.

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

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

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

</Project>
```

3. Переименуйте файл `Class1.cs` в `Dog.cs`.
4. В файле `Dog.cs` определите класс `Dog` с методом `Speak`, который выводит сообщение в консоль. Сообщение должно формироваться на основе переданного параметра типа `string`:

```
namespace DynamicLoadAndExecute.Library;

public class Dog
{
    public void Speak(string? name)
    {
        WriteLine($"{name} says Woof!");
    }
}
```

5. В редакторе кода создайте проект консольного приложения с именем `DynamicLoadAndExecute.Console` и добавьте его в решение `Chapter07`.
6. В файле `DynamicLoadAndExecute.Console.csproj` проекта включите обработку предупреждений как ошибок, а также выполните два глобальных импорта:
  - статический импорт класса `Console`;
  - импорт пространства имен для работы с рефлексией.

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

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

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

</Project>
```

7. Отобразите структуру файлов обоих проектов `DynamicLoadAndExecute`, чтобы можно было видеть полную иерархию, включая скрытые папки (рис. 7.1).
8. Постройте проект `DynamicLoadAndExecute.Library`, чтобы получить скомпилированную сборку библиотеки классов в папке `bin` (см. рис. 7.1).
9. Соберите проект `DynamicLoadAndExecute.Console`, чтобы сформировалась соответствующая папка `bin`.
10. Скопируйте следующие три файла из папки `bin\Debug\net8.0` проекта `DynamicLoadAndExecute.Library` в аналогичную папку проекта `DynamicLoadAndExecute.Console` (см. рис. 7.1):
  - `DynamicLoadAndExecute.Library.deps.json`
  - `DynamicLoadAndExecute.Library.dll`
  - `DynamicLoadAndExecute.Library.pdb`



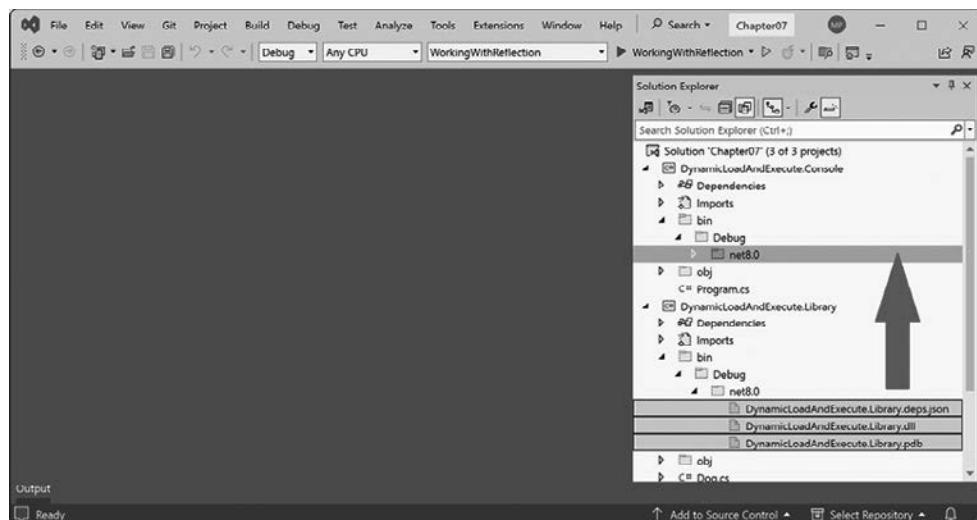
Вместо копирования файлов вручную можно указать одинаковый выходной каталог для обоих проектов — `DynamicLoadAndExecute.Library` и `DynamicLoadAndExecute.Console`.

11. В папке проекта `DynamicLoadAndExecute.Console` создайте файл класса `Program.Helpers.cs`. Удалите в нем все содержимое и добавьте метод, который будет выводить информацию о сборке и ее типах:

```
// Нет явного пространства имен!
```

```
partial class Program
```

```
{
    private static void OutputAssemblyInfo(Assembly a)
    {
        WriteLine($"FullName: {a.FullName}");
        WriteLine($"Location: {Path.GetDirectoryName(a.Location)}");
        WriteLine($"IsCollectible: {a.IsCollectible}");
        WriteLine("Defined types:");
        foreach (TypeInfo info in a.DefinedTypes)
        {
            if (!info.Name.EndsWith("Attribute"))
            {
                WriteLine($" Name: {info.Name}, Members: {info.GetMembers().Count()}");
            }
        }
        WriteLine();
    }
}
```



**Рис. 7.1.** Копирование сборки библиотеки классов в проект консольного приложения

12. В том же проекте создайте файл `DemoAssemblyLoadContext.cs`. Удалите в нем все содержимое и добавьте определения для класса, который загружает указанную сборку в текущий контекст выполнения с использованием распознавателя зависимостей:

```
using System.Runtime.Loader; // Подключение класса AssemblyDependencyResolver

internal class DemoAssemblyLoadContext : AssemblyLoadContext
{
    private readonly AssemblyDependencyResolver _resolver;
```

```
public DemoAssemblyLoadContext(string mainAssemblyToLoadPath)
    : base(isCollectible: true)
{
    _resolver = new AssemblyDependencyResolver(mainAssemblyToLoadPath);
}
```



**Внимание!** Частичная загрузка или выгрузка сборки невозможна, если на нее существуют активные ссылки. Это касается в том числе сборок, на которые явно ссылается ваш проект, а также самой сборки узла — консольного приложения.

13. В файле `Program.cs` удалите все содержимое. Затем определите класс `Program` с методом `Main`. Внутри метода создайте экземпляр пользовательского контекста загрузки, загрузите сборку `DynamicLoadAndExecute.Library`, выведите сведения о ней, создайте объект типа `Dog` и вызовите его метод `Speak`. Все это должно происходить динамически во время выполнения программы:

```
Assembly? thisAssembly = Assembly.GetEntryAssembly();

if (thisAssembly is null)
{
    WriteLine("Could not get the entry assembly.");
    return; // Выход из приложения
}

OutputAssemblyInfo(thisAssembly);

WriteLine($"Creating load context for:\n {
    Path.GetFileName(thisAssembly.Location)}\n");

DemoAssemblyLoadContext loadContext = new(thisAssembly.Location);

string assemblyPath = Path.Combine(
    Path.GetDirectoryName(thisAssembly.Location) ?? "",
    "DynamicLoadAndExecute.Library.dll");

WriteLine($"Loading:\n {Path.GetFileName(assemblyPath)}\n");

Assembly dogAssembly = loadContext.LoadFromAssemblyPath(assemblyPath);

OutputAssemblyInfo(dogAssembly);

Type? dogType = dogAssembly.GetType("DynamicLoadAndExecute.Library.Dog");

if (dogType is null)
{
    WriteLine("Could not get the Dog type.");
    return;
}
```

```
MethodInfo? method = dogType.GetMethod("Speak");

if (method != null)
{
    object? dog = Activator.CreateInstance(dogType);

    for (int i = 0; i < 10; i++)
    {
        method.Invoke(dog, new object[] { "Fido" });
    }
}

WriteLine();
WriteLine("Unloading context and assemblies.");
loadContext.Unload();
```

14. Запустите проект `DynamicLoadAndExecute.Console`. В результате появится следующий вывод:



Сборку, содержащую основную программу (то есть консольное приложение), удалить из памяти нельзя — она остается загруженной до завершения процесса. А вот динамически загруженные библиотеки могут быть выгружены, если для этого создать отдельный контекст загрузки и освободить его.

Теперь у вас есть представление о том, как с помощью рефлексии читать метаданные сборки, загружать сборки во время выполнения, создавать экземпляры типов и вызывать методы. Это только основные возможности. Рефлексия открывает гораздо больше — ей можно посвятить отдельную книгу. Но перед этим стоит рассмотреть и ограничения, с которыми придется столкнуться.

## Предупреждение о рефлексии и Native AOT

Одной из важных тем в .NET в последние годы стала активная поддержка *навигационной компиляции Ahead-of-Time (AOT)* — особенно в виде технологии Native AOT.

Приложения и сервисы, созданные с помощью Native AOT:

- *самодостаточны* — не требуют установленной среды выполнения .NET на целевом компьютере;
- *компилируются в нативный код* еще до запуска — благодаря этому сокращается время запуска и снижаются затраты памяти. Такой подход особенно полезен при масштабировании микросервисов (со множеством экземпляров), которые часто запускаются и останавливаются.

Native AOT преобразует промежуточный код (IL) в нативный во время публикации, в отличие от JIT-компиляции, которая выполняется во время выполнения. Однако приложения с Native AOT всегда ориентированы на конкретную среду, например Windows x64 или Linux ARM.

При всех преимуществах у технологии Native AOT есть и важные ограничения.

- Нельзя загружать сборки динамически во время выполнения.
- Нельзя генерировать код в процессе выполнения, например, с помощью пространства имён `System.Reflection.Emit`.
- Требуется сокращение кода (*trimming*), что может влиять на структуру сборки и ее поведение.
- Сборка должна быть самодостаточной — в нее должны быть включены все вызываемые библиотеки, что увеличивает итоговый размер.



**Внимание!** Если вы активно используете возможности рефлексии, такие как динамическая загрузка типов и вызов методов, то воспользоваться преимуществами Native AOT не получится. Эти сценарии с ней несовместимы.

Хотя ваши собственные приложения и сервисы могут не включать перечисленные выше возможности, многие ключевые компоненты платформы .NET активно

применяют их. Например, ASP.NET Core MVC (в том числе сервисы веб-API, построенные на контроллерах) и Entity Framework Core используют генерацию кода во время выполнения для реализации своей функциональности.

Команды разработчиков .NET продолжают активно работать над тем, чтобы расширить поддержку Native AOT. На данный момент .NET 8 предоставляет только начальную совместимость с ASP.NET Core при использовании минимальных API и не поддерживает EF Core.

Судя по текущему ходу работ, полноценная поддержка Native AOT для ASP.NET Core MVC и EF Core может появиться только в .NET 10.

В Native AOT допускается использование некоторых возможностей рефлексии. Однако важное ограничение связано с процессом сокращения кода (trimming). Он анализирует сборку на этапе компиляции и удаляет все члены, к которым, по его мнению, нет доступа.

Если к какому-либо методу или свойству можно обратиться только через рефлексию и это не указано явно, такой член может быть удален. Это приведет к выбросу исключений во время выполнения.



**Дельный совет:** если вы используете рефлексию, аннотируйте типы и члены, к которым должен быть обеспечен доступ, с помощью атрибута [DynamicallyAccessedMembers]. Благодаря этому компилятор учитывает, что соответствующий член требуется сохранить даже при сокращении кода.

## Изменения рефлексии в .NET 9

В версиях .NET 5 – .NET 8 с помощью класса `AssemblyBuilder` можно было создавать сборки и задавать метаданные для динамически определенных типов. Однако сохранить такие сборки на диск было невозможно. Это ограничение мешало миграции проектов с .NET Framework на современный .NET.

В .NET 9 добавлен новый класс `PersistedAssemblyBuilder`, который позволяет сохранять созданные сборки. На момент написания книги .NET 9 находился в предварительной версии, и API класса может измениться до окончательного релиза, запланированного на ноябрь 2025 года. Документация по классу `PersistedAssemblyBuilder` опубликована на странице [learn.microsoft.com/ru-ru/dotnet/core/whats-new/dotnet-9/libraries#reflection](https://learn.microsoft.com/ru-ru/dotnet/core/whats-new/dotnet-9/libraries#reflection).

## Дополнительные возможности рефлексии

Рефлексия дает гораздо больше, чем просто загрузку сборок и вызов методов. Вот лишь несколько дополнительных сценариев.

- Проверка содержимого сборки с помощью API `MetadataLoadContext`: [docs.microsoft.com/ru-ru/dotnet/standard/assembly/inspect-contents-using-metadataloadcontext](https://docs.microsoft.com/ru-ru/dotnet/standard/assembly/inspect-contents-using-metadataloadcontext).
- Динамическое создание нового кода и сборок: [docs.microsoft.com/ru-ru/dotnet/api/system.reflection.emit.assemblybuilder](https://docs.microsoft.com/ru-ru/dotnet/api/system.reflection.emit.assemblybuilder).

## Работа с деревьями выражений

*Деревья выражений* представляют код в виде структуры данных, которую можно исследовать или выполнить. Они неизменяемы — изменить существующее дерево нельзя, но можно создать его копию с нужными изменениями.

Если сравнивать деревья выражений с обычными функциями, то функции позволяют варьировать входные значения, но сама их структура — то, как обрабатываются эти значения, — остается неизменной. Деревья выражений, напротив, дают доступ к структуре самой функции, которую можно анализировать и даже перестроить во время выполнения. Это позволяет гибко управлять логикой выполнения кода в зависимости от условий.

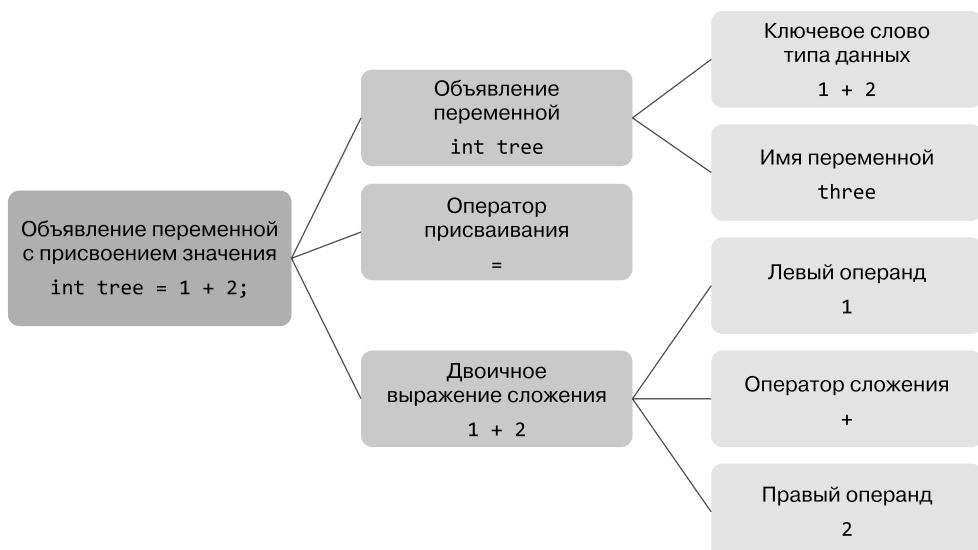
Кроме того, деревья выражений позволяют представить выражение в абстрактной форме — не как C#-код, а как структуру в памяти. Это открывает возможность преобразовывать такое представление в другие формы, включая другие языки или форматы.

Пример из практики — когда вы пишете LINQ-запрос к базе данных с помощью EF Core. Такой запрос компилируется в дерево выражений, которое затем преобразуется в SQL. При этом даже самое простое выражение на C# можно представить в виде дерева выражений.

Рассмотрим простейший пример — сложение двух чисел:

```
int three = 1 + 2;
```

Это выражение можно представить в виде дерева выражений, как показано на рис. 7.2.



**Рис. 7.2.** Дерево выражений для простого выражения сложения двух чисел

Как показывает схема, даже такое простое выражение, как `int three = 1 + 2;`, может быть представлено в виде дерева. Компиляторы используют древовидную структуру кода по нескольким причинам.

- **Иерархическая природа кода.** Большинство конструкций в коде вложены друг в друга. Деревья естественным образом отражают такую иерархию: каждый узел соответствует определенной конструкции, а его дочерние элементы — вложенным частям. Например, оператор `if` имеет ветви для условия и альтернативного блока `else`. Благодаря этому упрощаются навигация по структуре кода и его анализ во время компиляции.
- **Удобство для преобразований.** Древовидное представление облегчает преобразование, оптимизацию и генерацию кода. Если нужно изменить или оптимизировать определенное выражение, компилятор может работать с поддеревом, не затрагивая остальной код. Благодаря этому весь процесс становится более управляемым и модульным.
- **Упрощенный синтаксический анализ (парсинг).** Синтаксический анализ — это этап, на котором компилятор читает и «понимает» структуру кода. Представляя исходный код в виде дерева, компилятор может проще сопоставлять конструкции языка (например, условия, циклы, выражения) с правилами грамматики. Каждый узел в дереве соответствует конкретной конструкции, а связи между узлами отображают вложенность и зависимость.
- **Семантический анализ.** После синтаксической проверки выполняется семантический анализ: проверяются типы, области видимости, корректность использования идентификаторов и другие аспекты. Благодаря дереву компилятор может систематически проверять каждый элемент кода — как в отдельности, так и в контексте его родителей и дочерних узлов. Это помогает выявить ошибки и гарантировать соответствие языковым правилам.
- **Оптимизация кода.** Древовидная структура упрощает реализацию различных оптимизаций. Компилятор может локально анализировать поддеревья и выполнять операции вроде свертки констант (предварительного вычисления значений), удаления неиспользуемого кода и преобразования циклов — без нарушения общей логики программы.
- **Генерация целевого кода.** На финальном этапе дерево используется для трансляции в целевой код — будь то машинный код при Native AOT или промежуточный код IL. Дерево обеспечивает упорядоченное и логичное преобразование каждой конструкции исходного языка в ее эквивалент в целевом представлении, сохраняя структуру и порядок выполнения.
- **Отладка и сообщения об ошибках.** Когда компилятор обнаруживает ошибку, дерево позволяет точно указать ее местоположение. Каждый узел связан с конкретным участком исходного кода, благодаря чему можно вывести понятное сообщение с точным указанием строки и контекста ошибки. Это облегчает исправление и делает процесс разработки более прозрачным.

В целом древовидная структура помогает компилятору эффективно обрабатывать исходный код на всех ключевых этапах — от синтаксического и семантического анализа до оптимизации и генерации исполняемого кода.

## Обзор компонентов деревьев выражений

Пространство имен `System.Linq.Expressions` содержит типы, предназначенные для описания различных элементов дерева выражений. В табл. 7.1 перечислены наиболее часто используемые типы.

**Таблица 7.1.** Компоненты, применяемые при построении деревьев выражений

Тип	Описание
<code>BinaryExpression</code>	Выражение с двоичным оператором
<code>BlockExpression</code>	Блок, содержащий последовательность выражений и определение переменных
<code>CatchBlock</code>	Оператор <code>catch</code> в блоке <code>try</code>
<code>ConditionalExpression</code>	Выражение с условным оператором
<code>LambdaExpression</code>	Лямбда-выражение
<code>MemberAssignment</code>	Присвоение значения полю или свойству
<code>MemberExpression</code>	Доступ к полю или свойству
<code>MethodCallExpression</code>	Вызов метода
<code>NewExpression</code>	Вызов конструктора

Выполняться могут только те деревья выражений, которые представляют собой лямбда-выражения.

## Выполнение простейшего дерева выражений

Рассмотрим, как создать, скомпилировать и выполнить дерево выражений, представляющее простое присваивание:

```
int three = 1 + 2;
```

Выполните следующие действия.

1. В редакторе кода создайте проект `Console App` (Консольное приложение) / `console`, присвойте ему имя `WorkingWithExpressionTrees` и добавьте в решение `Chapter07`.
2. В файле `WorkingWithExpressionTrees.csproj` проекта статически и глобально импортируйте класс `Console`:

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

3. В файле `Program.cs` удалите все содержимое. Затем определите дерево выражений, скомпилируйте его и выполните:

```
using System.Linq.Expressions; // Подключение класса Expression и т.п.

ConstantExpression one = Expression.Constant(1, typeof(int));
ConstantExpression two = Expression.Constant(2, typeof(int));
BinaryExpression add = Expression.Add(one, two);

Expression<Func<int>> expressionTree = Expression.Lambda<Func<int>>(add);

Func<int> compiledTree = expressionTree.Compile();

WriteLine($"Result: {compiledTree()}");
```

4. Выполните консольное приложение. В результате вы увидите следующий вывод:

Result: 3

Теперь, построив и выполнив простейшее дерево выражений, вы получили общее представление о том, как они работают и что собой представляют.

Дальше мы перейдем к одной из самых интересных возможностей современного .NET — генераторам исходного кода.

## Создание генераторов исходного кода

*Генераторы исходного кода* были представлены в C# 9 и .NET 5. С их помощью можно получить доступ к объекту компиляции, который представляет весь текущий код проекта, а затем на его основе сгенерировать дополнительные файлы исходного кода. Эти файлы автоматически включаются в компиляцию.

Генератор исходного кода работает схожим образом с анализатором, но вместо выявления проблем он добавляет новый код в процессе компиляции. Это особенно полезно в ситуациях, когда нельзя воспользоваться рефлексией, — например, если планируется использовать компиляцию Native AOT.

Хороший пример — генератор исходного кода для `System.Text.Json`. При обычной сериализации объектов в JSON используется рефлексия во время выполнения, что может быть медленно. Альтернатива — генерация кода, который сериализует объекты без рефлексии. Такой подход работает быстрее, поскольку весь код генерируется заранее и компилируется вместе с остальной программой.



Подробнее о генераторе исходного кода `System.Text.Json` вы можете узнать по адресу [devblogs.microsoft.com/dotnet/try-the-new-system-text-json-source-generator/](https://devblogs.microsoft.com/dotnet/try-the-new-system-text-json-source-generator/).

## Реализация простейшего генератора исходного кода

Создадим простой генератор исходного кода, который программно добавит в класс `Program` новый метод. Этот метод будет сгенерирован во время компиляции и добавлен как отдельный файл:

```
// Сгенерированный исходный код
partial class Program
{
    static partial void Message(string message)
    {
        System.Console.WriteLine($"Generator says: '{message}'");
    }
}
```

Затем метод, сгенерированный генератором исходного кода, можно вызвать в файле `Program.cs` проекта, в котором этот генератор используется. Давайте разберемся, как это сделать.

1. В редакторе кода создайте проект `Console App` (Консольное приложение) / `console`, присвойте ему имя `GeneratingCodeApp` и добавьте в решение `Chapter07`.
2. В файле `GeneratingCodeApp.csproj` проекта выполните статический и глобальный импорт класса `Console`:

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

3. В папке проекта `GeneratingCodeApp` создайте файл класса `Program.Methods.cs`.
4. В файле `Program.Methods.cs` определите частичный класс `Program` с частичным методом, принимающим параметр типа `string`:

```
partial class Program
{
    static partial void Message(string message);
}
```

5. В файле `Program.cs` удалите содержимое и вызовите частичный метод:
- ```
Message("Hello from some source generator code.");
```
6. В редакторе кода создайте проект `Class Library` (Библиотека классов) / `classlib`, присвойте ему имя `GeneratingCodeLib`, связанное с платформой .NET Standard 2.0, и добавьте его в решение `Chapter07`.



На текущий момент генераторы исходного кода должны быть ориентированы на платформу .NET Standard 2.0. Это указано в официальной документации: [learn.microsoft.com/ru-ru/dotnet/csharp/roslyn-sdk/source-generators-overview](https://learn.microsoft.com/ru-ru/dotnet/csharp/roslyn-sdk/source-generators-overview).

По умолчанию библиотеки классов для .NET Standard 2.0 используют версию C# 7.3: [docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/configure-language-version#defaults](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/configure-language-version#defaults).

7. В файле проекта `GeneratingCodeLib.csproj` укажите версию языка C# 12 или новее (для поддержки глобальных операторов `using` и необработанных строковых литералов), выполните статический и глобальный импорт класса `Console`, а также установите NuGet-пакеты `Microsoft.CodeAnalysis.Analyzers` и `Microsoft.CodeAnalysis.CSharp`:

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

<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
  <LangVersion>12</LangVersion>
  <EnforceExtendedAnalyzerRules>true</EnforceExtendedAnalyzerRules>
</PropertyGroup>

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

<ItemGroup>
  <PackageReference Include="Microsoft.CodeAnalysis.Analyzers" Version="3.3.4">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
      buildtransitive</IncludeAssets>
  </PackageReference>
  <PackageReference Include="Microsoft.CodeAnalysis.CSharp" Version="4.9.2" />
</ItemGroup>

</Project>
```



**Внимание!** Поскольку в файле проекта отсутствует элемент `<Nullable>enable</Nullable>`, компилятор не будет выдавать предупреждения, связанные с null-значениями. Если вы добавите этот элемент, в дальнейшем могут появиться такие предупреждения — это поведение ожидаемое.



Последние версии необходимых пакетов можно найти по ссылкам [www.nuget.org/packages/Microsoft.CodeAnalysis.Analyzers](http://www.nuget.org/packages/Microsoft.CodeAnalysis.Analyzers) и [www.nuget.org/packages/Microsoft.CodeAnalysis.CSharp](http://www.nuget.org/packages/Microsoft.CodeAnalysis.CSharp).

8. Выполните сборку проекта `GeneratingCodeLib`.
9. Переименуйте файл `Class1.cs` в `MessageSourceGenerator.cs`.
10. В файле `MessageSourceGenerator.cs` проекта `GeneratingCodeLib` определите класс, реализующий интерфейс `ISourceGenerator`, и пометьте его атрибутом `[Generator]`:

```
// Подключение [Generator], ISourceGenerator и т. п.
using Microsoft.CodeAnalysis;

namespace Packt.Shared;
```

```
[Generator]
public class MessageSourceGenerator : ISourceGenerator
{
    public void Execute(GeneratorExecutionContext execContext)
    {
        IMethodSymbol mainMethod = execContext.Compilation
            .GetEntryPoint(execContext.CancellationToken);

        string sourceCode = $$"""

        // Сгенерированный исходный код

        partial class {{mainMethod.ContainingType.Name}}
        {
            static partial void Message(string message)
            {
                System.Console.WriteLine($"Generator says: '{message}'");
            }
        }
        """;

        string typeName = mainMethod.ContainingType.Name;

        execContext.AddSource($"{typeName}.Methods.g.cs", sourceCode);
    }

    public void Initialize(GeneratorInitializationContext initContext)
    {
        // Этот генератор исходного кода не требует инициализации
    }
}
```



**Дельный совет:** добавьте в имя сгенерированного файла суффикс .g. или .generated. — так будет проще отличить его от обычного кода.

11. В файле GeneratingCodeApp.csproj проекта добавьте ссылку на проект библиотеки GeneratingCodeLib:

```
<ItemGroup>
    <ProjectReference Include="..\GeneratingCodeLib\GeneratingCodeLib.csproj"
        OutputItemType="Analyzer" ReferenceOutputAssembly="false" />
</ItemGroup>
```



**Дельный совет:** иногда результаты работы генераторов исходного кода не отображаются сразу. В этом случае может понадобиться перезапустить программу Visual Studio.

12. Выполните сборку проекта GeneratingCodeApp и обратите внимание на автоматически сгенерированный файл класса Program.Methods.g.cs.

- В Visual Studio его можно найти на панели Solution Explorer (Обозреватель решений), развернув узлы Dependencies > Analyzers > GeneratingCodeLib Packt.Shared.MessageSourceGenerator (рис. 7.3).

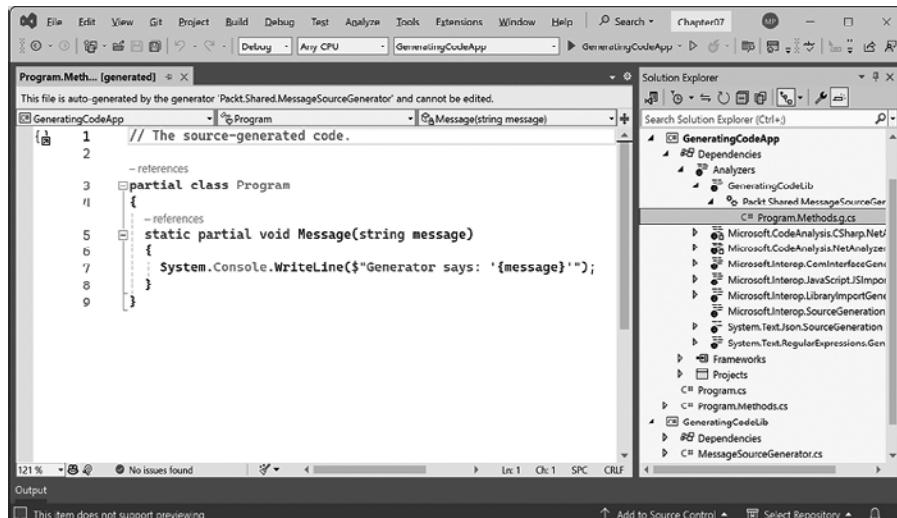


Рис. 7.3. Сгенерированный файл класса



По умолчанию код анализаторов не запускается автоматически. Чтобы включить генерацию кода во время компиляции, необходимо добавить специальную настройку в файл проекта.

- В проекте GeneratingCodeApp в файле проекта, внутри элемента `<PropertyGroup>` добавьте строку:

```
<EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
```

- В терминале создайте проект GeneratingCodeApp.
- В папке obj/Debug/net8.0 появится структура сгенерированных файлов.

Обратите внимание на вложенную папку GeneratingCodeLib/Packt.Shared.MessageSourceGenerator и на автоматически созданный файл с именем Program.Methods.g.cs.

### 13. Откройте файл Program.Methods.g.cs и изучите его содержимое:

```
// Сгенерированный исходный код

partial class Program
{
    static partial void Message(string message)
    {
        System.Console.WriteLine($"Generator says: '{message}'");
    }
}
```

14. Запустите консольное приложение. На экране появится сообщение, показанное ниже:

```
Generator says: 'Hello from some source generator code.'
```



Путь для размещения автоматически сгенерированных файлов можно задать с помощью элемента `<CompilerGeneratedFilesOutputPath>`.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 7.1. Материалы, доступные только онлайн

Чтобы рассмотреть еще один пример генератора исходного кода, можно ознакомиться с дополнительным онлайн-разделом к этой книге. Он размещен по адресу [github.com/markjprice/cs12dotnet8/blob/main/docs/ch07-source-generators.md](https://github.com/markjprice/cs12dotnet8/blob/main/docs/ch07-source-generators.md).

### Упражнение 7.2. Практические задания

Генераторы исходного кода — это широкая и многогранная тема. Если вы хотите углубиться в нее, посетите несколько полезных ресурсов.

- Спецификация дизайна генераторов исходного кода: [github.com/dotnet/roslyn/blob/main/docs/features/source-generators.md](https://github.com/dotnet/roslyn/blob/main/docs/features/source-generators.md).
- Примеры генераторов исходного кода: [github.com/dotnet/roslyn-sdk/tree/main/samples/CSharp/SourceGenerators](https://github.com/dotnet/roslyn-sdk/tree/main/samples/CSharp/SourceGenerators).
- Source Generators Cookbook: [github.com/dotnet/roslyn/blob/main/docs/features/source-generators.cookbook.md](https://github.com/dotnet/roslyn/blob/main/docs/features/source-generators.cookbook.md).

### Упражнение 7.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Из каких частей состоит сборка .NET и какие из них обязательны?
2. К каким элементам кода можно применять атрибут?
3. Какие части входят в номер версии и что они обозначают при семантическом версионировании?
4. Как получить объект сборки текущего консольного приложения?

5. Как получить все атрибуты, примененные к сборке?
6. Как определить и использовать собственный атрибут?
7. От какого класса нужно наследоваться, чтобы загружать сборки динамически?
8. Что такое дерево выражений и где оно используется в .NET?
9. Что такое генератор исходного кода и для чего он нужен?
10. Какой интерфейс должен реализовывать генератор исходного кода и какие методы в него входят?

## Упражнение 7.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-7---observing-and-modifying-code-execution-dynamically](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-7---observing-and-modifying-code-execution-dynamically).

## Резюме

В этой главе вы:

- познакомились с рефлексией и атрибутами;
- построили, скомпилировали и выполнили простое дерево выражений;
- создали генератор исходного кода и применили его в консольном приложении.

В следующей главе вы узнаете, как защищать данные с помощью средств криптографии.

# 8

# Защита данных и приложений с помощью криптографии

Эта глава посвящена защите данных от несанкционированного доступа и изменений. Вы узнаете, как применять шифрование, хеширование и цифровую подпись для обеспечения безопасности информации. Мы также разберем, как правильно генерировать случайные значения для криптографических операций и реализовать базовую проверку подлинности и авторизацию пользователей.



**Внимание!** Примеры кода в этой главе предназначены исключительно для базового знакомства с криптографическими примитивами. Не используйте их в производственных приложениях или библиотеках — это может поставить под угрозу безопасность ваших данных и проектов.

Рекомендуется опираться на библиотеки, разработанные специалистами по информационной безопасности. Эти решения основаны на тех же примитивах, но реализованы с учетом современных требований и лучших практик.

В завершение главы приведены ссылки на рекомендации Microsoft — на них следует опираться, если вы планируете реализовать аналогичную функциональность самостоятельно.

## Основные понятия в сфере безопасности

Зашита данных играет ключевую роль во многих сферах — от соблюдения правовых требований до обеспечения безопасности пользователей и сохранения репутации организаций.

В большинстве стран действуют законы и нормативные акты, обязывающие компании защищать персональные данные. Например, в Европейском союзе принят Общий регламент по защите данных (GDPR), в США — Калифорнийский закон о конфиденциальности потребителей (CCPA). В России действует Федеральный закон № 152-ФЗ «О персональных данных», который устанавливает требования к сбору, хранению, обработке и передаче персональной информации. Нарушение

таких нормативов может привести к штрафам, блокировке ресурсов, приостановке деятельности и другим последствиям, в зависимости от законодательства конкретной страны.

Отдельные отрасли, такие как здравоохранение и финансы, регулируются дополнительно. Например, в США применяются стандарты HIPAA (защита медицинских данных) и PCI-DSS (безопасность платежной информации). (В России подобные сферы также регулируются на уровне законодательства и отраслевых требований. В здравоохранении используется приказ Минздрава № 125н, в котором прописаны требования к защите персональных данных пациентов. Финансовая отрасль регулируется нормативами Центрального банка РФ, а также стандартами ФСТЭК и ФСБ в зависимости от уровня защищаемой информации. Кроме того, в России применяется ГОСТ Р 57580 – национальный стандарт по обеспечению безопасности персональных данных в финансовом секторе. – *Примеч. пер.*) Эти требования созданы для защиты чувствительной информации, характерной для конкретной сферы. Нарушение таких стандартов может повлечь за собой серьезные последствия – как юридические, так и финансовые.

Надежная защита данных важна и с точки зрения прав личности. Если личная информация уязвима, это может привести к утечке конфиденциальных данных и вторжению в частную жизнь. Недостаточная защита открывает путь к краже личности: злоумышленники могут получить доступ к данным, таким как номера социального страхования или реквизиты банковских карт, и использовать их для мошенничества. Утрата доверия пользователей после подобных инцидентов способна нанести серьезный урон репутации компании и привести к потере клиентской базы.

Организации, которые работают с интеллектуальной собственностью или запатентованными данными, также подвержены риску. Утечка таких данных может привести к их несанкционированному распространению, что поставит под угрозу конкурентные преимущества бизнеса.

## Методы защиты данных

Существует множество способов защиты данных. Ниже кратко описаны наиболее распространенные методы, а далее в главе мы рассмотрим их подробнее и приведем практические примеры.

- **Шифрование и расшифровка.** Это двусторонний процесс: открытый текст преобразуется в шифротекст, а затем обратно. *Открытый текст* – это исходные данные, которые нужно защитить. *Шифротекст* – результат шифрования, нечитаемый без соответствующего ключа.
- **Хеширование.** Односторонний процесс, в результате которого из данных создается дайджест фиксированной длины. Например, хеш этого абзаца может

выглядеть так: DoBFtDhKeN0aaaLVdErtrZ3mpZSvpWDQ9TXDoSTq0sQ=. Хеш-функции применяются для безопасного хранения паролей и проверки целостности данных. Однако для паролей не следует использовать простые хеш-алгоритмы. Вместо них рекомендуется применять алгоритмы PBKDF2, bcrypt или scrypt — они обеспечивают надежную защиту при правильной настройке, сводя к минимуму вероятность, что разные входные данные дадут один и тот же дайджест.

- **Цифровая подпись.** Применяется для подтверждения происхождения данных. Получатель может проверить подпись с помощью открытого ключа и убедиться, что сообщение действительно пришло от заявленного источника и не было изменено.
- **Проверка подлинности.** Используется для подтверждения личности пользователя путем проверки его учетных данных.
- **Авторизация.** Позволяет удостовериться, что пользователь имеет право выполнять конкретные действия или получать доступ к определенным данным — обычно на основе ролей или группы.



**Дельный совет:** если вы работаете с данными, где важна безопасность (а безопасность важна всегда), привлеките опытного специалиста по информационной безопасности. Даже незначительные ошибки в реализации могут привести к уязвимостям, которые станут очевидны слишком поздно!

## Ключи и их размеры

Во многих алгоритмах безопасности используются криптографические ключи. *Ключ* — это байтовый массив определенной длины, который служит основой для выполнения операций защиты данных. В зависимости от задачи применяются ключи разного типа и размера. Ниже приведены основные направления их использования.

- Шифрование и расшифровка: AES, 3DES, RC2, Rijndael и RSA.
- Цифровая подпись и проверка подписи: RSA, ECDSA и DSA.
- Проверка подлинности и целостности сообщений: HMAC.
- Безопасный обмен ключами (совместная выработка ключей): алгоритмы Диффи — Хеллмана и эллиптическая кривая Диффи — Хеллмана.



**Дельный совет:** выбирайте больший размер ключа, если хотите повысить стойкость защиты. Однако не переусердствуйте: хотя существуют реализации RSA с ключами до 16 384 бит, их генерация может занять дни, и в большинстве случаев такая длина избыточна. На сегодняшний день 2048-битного ключа RSA достаточно для большинства сценариев — вплоть до 2030 года. После этого стоит перейти на ключи длиной не менее 3192 бит.

Ключи могут быть двух видов.

- **Симметричные ключи.** Один и тот же ключ используется как для шифрования, так и для расшифровки. Его необходимо надежно хранить, поскольку знание ключа дает полный доступ к зашифрованным данным. Такие ключи также называют *общими* или *секретными*.
- **Асимметричные ключи.** Используется пара ключей: *открытый* и *закрытый*. Открытый ключ может быть распространен, а расшифровка или подпись выполняется только с помощью закрытого ключа, который должен храниться в секрете.



**Дельный совет:** алгоритмы симметричного шифрования работают быстро и подходят для обработки больших объемов данных — например, при потоковом шифровании. Асимметричные алгоритмы медленнее и используются, как правило, для небольших блоков — например, при создании и проверке цифровых подписей. Симметричные ключи несут больший риск: поскольку один и тот же ключ нужен обеим сторонам, его компрометация ведет к утечке данных.

На практике часто сочетают оба подхода: данные шифруются симметричным ключом, а сам ключ передается с помощью асимметричного шифрования. Этот гибридный подход лежит в основе работы протокола *Transport Layer Security (TLS)* — стандарта, который используется в интернете для защиты соединений. Ранее для этой цели применялся устаревший протокол *Secure Sockets Layer (SSL)*, где ключи сессии шифровались напрямую с помощью RSA. Сегодня такой подход не используется, но аббревиатура SSL все еще встречается в обиходе, когда говорят о TLS.

## Векторы инициализации и размеры блоков

При шифровании больших объемов данных могут встречаться повторяющиеся последовательности. Например, в английском тексте слово `the` используется очень часто. Если алгоритм шифрует каждый одинаковый фрагмент одинаково — например, каждый раз превращает `the` в `hQ2`, — злоумышленник может воспользоваться этим, чтобы попытаться взломать шифр. Один из таких случаев показан в следующем выводе:

```
When the wind blew hard the umbrella broke.
5:s4&hQ2aj#D f9d1d£8fh"&hQ2s0)an DF8SFd#[1]
```

Чтобы предотвратить повторение зашифрованных последовательностей, данные разбиваются на блоки. После шифрования первого блока результат используется для модификации шифрования следующего блока. Это означает, что даже

если два блока содержат одинаковые данные, их зашифрованные версии будут различаться.

Для шифрования первого блока требуется специальный набор байтов — *вектор инициализации* (*Initialization Vector, IV*). Его роль — ввести случайность в процесс шифрования с самого начала. Вектор инициализации:

- генерируется случайным образом для каждого нового сообщения;
- передается вместе с зашифрованными данными;
- не является секретом, поэтому его не нужно защищать так же строго, как ключ.

## Соль

Соль — это случайный массив байтов, добавляемый к данным перед применением односторонней хеш-функции. Ее задача — сделать каждый хеш уникальным, даже если исходные данные совпадают. Без соли одинаковые входные значения дают одинаковый дайджест. Например, если несколько пользователей выберут популярный пароль **123456** (по данным за 2023 год, так поступили примерно 23 млн человек — и это самый распространенный пароль в мире), их дайджесты будут идентичны. Это создает уязвимость: злоумышленник может использовать радужные таблицы — заранее вычисленные хеши для распространенных паролей — и быстро подобрать нужный.



**В копилку знаний:** любопытные цифры по статистике паролей можно найти по следующим адресам: <https://techjury.net/blog/password-statistics/> и <https://explodingtopics.com/blog/password-stats>.

Когда пользователь создает учетную запись, система должна сгенерировать случайную соль и соединить ее с введенным паролем. Получившаяся строка затем передается в хеш-функцию. В результате получается дайджест, который сохраняется в базе данных вместе с солью, но не с исходным паролем.

При следующем входе пользователь снова вводит свой пароль. Программа находит соответствующую соль, соединяет ее с новым вводом, хеширует полученную строку и сравнивает результат с ранее сохраненным дайджестом. Если значения совпадают — вход разрешен.

Важно понимать: даже хеширование с солью — это только первый шаг к безопасному хранению паролей. Для действительно надежной защиты используют дополнительные техники — добавление перца (дополнительный секрет, известный только серверу), растягивание ключей (например, с помощью PBKDF2, bcrypt или scrypt) и регулярное обновление стратегий хранения и проверки. Но такие подходы выходят за рамки этой главы. Здесь мы ограничимся базовым примером хеширования с солью.

## Генерация ключей и векторов инициализации

Ключи и векторы инициализации (IV) — это массивы байтов. Обе стороны, которые хотят обмениваться зашифрованными данными, должны использовать одинаковые значения ключа и IV. Но безопасный обмен такими массивами может быть непростым.

Для надежной генерации ключей и IV можно использовать функцию получения ключа на основе пароля (PBKDF2). Один из подходящих классов — `Rfc2898DeriveBytes`. Он принимает пароль, соль, количество итераций и алгоритм хеширования (по умолчанию — SHA-1, но лучше выбрать как минимум SHA256). Затем с помощью метода `GetBytes` этот класс генерирует нужное количество байтов — например, для ключа и IV. Количество итераций определяет, сколько раз будет хешироваться пароль. Чем оно больше, тем дольше будет выполняться операция — и тем сложнее взломать пароль.

Хотя `Rfc2898DeriveBytes` можно использовать и для генерации IV, в большинстве случаев IV нужно генерировать случайно для каждого сообщения. Его передают вместе с зашифрованными данными в открытом виде — скрывать IV не нужно.



**Дельный совет:** рекомендуется использовать соль размером не менее 8 байт, а количество итераций выбрать таким, чтобы генерация ключа и IV занимала около 100 мс на целевом компьютере. Это значение будет со временем расти по мере совершенствования процессоров. В примере в следующем разделе мы используем 150 000 итераций, но к моменту чтения этой книги для некоторых компьютеров этого может быть уже недостаточно.

## Шифрование и расшифровка данных

В .NET доступно несколько алгоритмов шифрования, из которых можно выбрать подходящий для своей задачи.

В устаревшем .NET Framework часть алгоритмов шифрования реализуется на уровне операционной системы. Такие реализации часто имеют суффиксы `CryptoServiceProvider` или `Cng`. Другие алгоритмы реализуются в библиотеке базовых классов (BCL) и отмечаются суффиксом `Managed` — это значит, что они полностью написаны на C# и работают независимо от операционной системы.

В современном .NET (начиная с .NET Core) вся криптография реализуется на уровне операционной системы. Если ОС поддерживает сертифицированные алгоритмы FIPS (Federal Information Processing Standards), то .NET отдает предпочтение этим алгоритмам и не использует собственные реализации в BCL.



Криптографические операции в .NET выполняются средствами операционной системы. Благодаря этому, если в ОС устраняется уязвимость в системе безопасности, приложения .NET сразу начинают использовать обновленную реализацию. Но есть и обратная сторона: приложения .NET могут использовать только те криптографические функции, которые поддерживает операционная система. Подробную информацию об этом можно найти по адресу [learn.microsoft.com/ru-ru/dotnet/standard/security/cross-platform-cryptography](https://learn.microsoft.com/ru-ru/dotnet/standard/security/cross-platform-cryptography).

В большинстве случаев вы будете создавать экземпляры алгоритмов шифрования с помощью абстрактных классов, таких как `Aes`, и их фабричного метода `Create`. Поэтому нет необходимости беспокоиться, используется ли реализация с суффиксом `CryptoServiceProvider` или `Managed` — .NET сам выберет подходящую версию.

Одни алгоритмы шифрования задействуют симметричные ключи, а другие — асимметричные. Симметричные алгоритмы, такие как AES, хорошо подходят для шифрования больших объемов данных. Они обычно применяются через класс `CryptoStream`, который позволяет обрабатывать данные потоком. Асимметричные алгоритмы — например, RSA — предназначены для работы с небольшими объемами данных, которые обрабатываются как массивы байтов.

Основной алгоритм асимметричного шифрования — RSA — был разработан в 1977 году тремя учеными — Роном Ривестом, Ади Шамиром и Леонардом Адлеманом. Название алгоритма составлено из их фамилий.



Интересный исторический факт: аналогичный алгоритм был создан еще в 1973 году британским математиком Клиффордом Коксом, который работал в разведке GCHQ. Однако эта разработка оставалась засекреченной до 1997 года, и именно Ривест, Шамир и Адлеман получили признание за алгоритм, названный в их честь, — RSA.

Наиболее часто используемые алгоритмы симметричного шифрования в .NET основаны на абстрактном классе `SymmetricAlgorithm`. Среди них:

- `Aes`
- `DESCryptoServiceProvider`
- `TripleDES`
- `RC2CryptoServiceProvider`
- `RijndaelManaged`

Если нужно расшифровать данные, полученные от внешней системы, следует использовать тот же алгоритм, который применялся для их шифрования. Здесь выбор за вами уже не стоит. То же самое касается обратной ситуации: если вы отправляете зашифрованные данные в систему, которая принимает только определенный алгоритм, придется придерживаться его.

А вот если шифрование и расшифровка выполняются полностью в вашем приложении, вы можете самостоятельно выбрать алгоритм, учитывая требования к безопасности, производительности и совместимости.



**Дельный совет:** для симметричного шифрования предпочтительнее всего использовать стандарт AES (Advanced Encryption Standard), основанный на алгоритме Rijndael. Для асимметричного шифрования разумный выбор — RSA. Обратите внимание: RSA и DSA — разные алгоритмы. Стандарт DSA (Digital Signature Algorithm) используется только для цифровых подписей — он не подходит для шифрования данных.

## Симметричное шифрование с помощью AES

Чтобы было проще повторно использовать код, связанный с защитой данных, в нескольких проектах этой главы, мы создадим статический класс `Protector` в отдельной библиотеке классов. Затем подключим эту библиотеку к консольному приложению, где и реализуем симметричное шифрование с помощью AE.

Приступим!

- В редакторе кода создайте проект консольного приложения, указав следующие настройки:
  - выберите шаблон проекта `Console App` (Консольное приложение) / `console`;
  - в поле `Solution name` (Имя решения) укажите значение `Chapter08`;
  - в поле `Project name` (Имя проекта) укажите значение `EncryptionApp`.
- Создайте проект `Class Library` (Библиотека классов) / `classlib`, присвойте ему имя `CryptographyLib` и добавьте в решение `Chapter08`.
- В проекте `CryptographyLib` переименуйте файл `Class1.cs` в `Protector.cs`.
- В том же проекте настройте обработку предупреждений как ошибок и выполните глобальный статический импорт класса `System.Console`.
- В проекте `EncryptionApp` включите обработку предупреждений как ошибок, выполните глобальный статический импорт класса `System.Console` и добавьте ссылку на библиотеку `CryptographyLib`:

```
<ItemGroup>
  <ProjectReference Include="..\CryptographyLib\CryptographyLib.csproj" />
</ItemGroup>
```

- Выполните сборку проекта `EncryptionApp` и удостоверьтесь, что компиляция проходит без ошибок.
- В файле `Protector.cs` определите статический класс `Protector`. Добавьте в него поля для хранения массива байтов соли и числа итераций, а также методы для шифрования и расшифровки данных:

```
using System.Diagnostics; // Подключение класса Stopwatch
using System.Security.Cryptography; // Подключение классов Aes
                                         // и других криптографических типов
using System.Text; // Подключение класса Encoding

using static System.Convert; // Подключение метода ToBase64String
                           // и других вспомогательных методов

namespace Packt.Shared;

public static class Protector
{
    // Размер соли должен быть не меньше 8 байт – здесь используется 16 байт
    private static readonly byte[] salt =
        Encoding.Unicode.GetBytes("7BANANAS");

    // По умолчанию класс Rfc2898DeriveBytes использует 1000 итераций
    // Количество итераций должно быть достаточно большим, чтобы генерация
    // ключа и IV занимала на целевой системе не менее 100 мс –
    // 150 000 итераций занимают примерно 139 мс
    // на Intel Core i7-1165G7 11-го поколения @ 2.80 ГГц
    private static readonly int iterations = 150_000;

    public static string Encrypt(
        string plainText, string password)
    {
        byte[] encryptedBytes;
        byte[] plainBytes = Encoding.Unicode.GetBytes(plainText);

        using (Aes aes = Aes.Create()) // Получение экземпляра с помощью
                                       // фабричного метода абстрактного класса
        {
            // Измерение времени, требуемого для генерации ключа и IV
            Stopwatch timer = Stopwatch.StartNew();

            using (Rfc2898DeriveBytes pbkdf2 = new(
                password, salt, iterations, HashAlgorithmName.SHA256))
            {
                WriteLine($"PBKDF2 algorithm: {pbkdf2.HashAlgorithm
                    }, Iteration count: {pbkdf2.IterationCount:N0}");

                aes.Key = pbkdf2.GetBytes(32); // Установка 256-битного ключа
                aes.IV = pbkdf2.GetBytes(16); // Установка 128-битного вектора
                                              // инициализации (IV)
            }

            timer.Stop();

            WriteLine($"{timer.ElapsedMilliseconds:N0} milliseconds to generate Key and IV.");
        }

        if (timer.ElapsedMilliseconds < 100)
        {
            ConsoleColor previousColor = ForegroundColor;
            ForegroundColor = ConsoleColor.Red;
        }
    }
}
```

```
        WriteLine("WARNING: The elapsed time to generate the Key and IV " +
                  "may be too short to provide a secure encryption key.");
        ForegroundColor = previousColor;
    }

    WriteLine($"Encryption algorithm: {nameof(Aes)}-{aes.KeySize},
              {aes.Mode} mode with {aes.Padding} padding.");

    using (MemoryStream ms = new())
    {
        using (ICryptoTransform transformer = aes.CreateEncryptor())
        {
            using (CryptoStream cs = new(
                ms, transformer, CryptoStreamMode.Write))
            {
                cs.Write(plainBytes, 0, plainBytes.Length);

                if (!cs.HasFlushedFinalBlock)
                {
                    cs.FlushFinalBlock();
                }
            }
            encryptedBytes = ms.ToArray();
        }
    }

    return ToBase64String(encryptedBytes);
}

public static string Decrypt(
    string cipherText, string password)
{
    byte[] plainBytes;
    byte[] cryptoBytes = FromBase64String(cipherText);

    using (Aes aes = Aes.Create())
    {
        using (Rfc2898DeriveBytes pbkdf2 = new(
            password, salt, iterations, HashAlgorithmName.SHA256))
        {
            aes.Key = pbkdf2.GetBytes(32);
            aes.IV = pbkdf2.GetBytes(16);
        }

        using (MemoryStream ms = new())
        {
            using (ICryptoTransform transformer = aes.CreateDecryptor())
            {
                using (CryptoStream cs = new(
                    ms, aes.CreateDecryptor(), CryptoStreamMode.Write))
                {
                    cs.Write(cryptoBytes, 0, cryptoBytes.Length);
                }
            }
        }
    }
}
```

```

        if (!cs.HasFlushedFinalBlock)
        {
            cs.FlushFinalBlock();
        }
    }
    plainBytes = ms.ToArray();
}
}

return Encoding.Unicode.GetString(plainBytes);
}
}

```

Ниже перечислены важные аспекты, которые стоит учитывать касательно предыдущего кода.

- Соль и количество итераций можно зафиксировать в коде, но более надежным вариантом будет сохранять их вместе с сообщением. Пароль, напротив, следует передавать в момент вызова методов `Encrypt` и `Decrypt`.
- Конструкторы класса `Rfc2898DeriveBytes` требуют объект `HashAlgorithmName`. В данном случае выбран `SHA256`, поскольку он считается более предпочтительным вариантом по сравнению с устаревшим значением `SHA1`.
- Класс `Stopwatch` используется для измерения времени, затраченного на генерацию ключа и вектора инициализации (IV). Если это время оказывается меньше 100 мс, выводится предупреждение.
- Промежуточный результат шифрования и расшифровки сохраняется во временном потоке `MemoryStream`. После завершения работы поток преобразуется в массив байтов с помощью метода `ToArray`.
- Класс `CryptoStream` выполняет буферизацию данных. Это важно, потому что криптографические преобразования требуют корректной обработки блоков фиксированного размера. Поэтому необходимо явно завершить обработку последнего блока.
- Для удобства восприятия зашифрованные байтовые массивы преобразуются в строковое представление с помощью кодировки Base64 — и затем преобразуются обратно при расшифровке.



**Дельный совет:** никогда не вставляйте пароль непосредственно в исходный код. Даже если он скомпилирован, его все равно можно извлечь с помощью инструментов дизассемблирования.

8. В проекте `EncryptionApp` откройте файл `Program.cs`. Удалите в нем содержимое, затем импортируйте пространства имен для классов `Protector` и `CryptographicException`:

```

using System.Security.Cryptography; // Подключение класса CryptographicException
using Packt.Shared; // Подключение класса Protector

```



В реальном проекте вы можете выполнить статический импорт класса Packt.Share.Protector, чтобы обращаться к его методам, например Encrypt, без префикса Protector. В этой главе мы сохраняем полное имя, чтобы вы не забывали, что эти методы написаны вами.

9. В этом же файле добавьте код, который запрашивает у пользователя текст сообщения и пароль, а затем выполняет шифрование и расшифровку сообщения:

```
Write("Enter a message that you want to encrypt: ");
string? message = ReadLine();

Write("Enter a password: ");
string? password = ReadLine();

if ((password is null) || (message is null))
{
    WriteLine("Message or password cannot be null.");
    return; // Выход из приложения
}

string cipherText = Protector.Encrypt(message, password);

WriteLine($"Encrypted text: {cipherText}");

Write("Enter the password: ");
string? password2Decrypt = ReadLine();

if (password2Decrypt is null)
{
    WriteLine("Password to decrypt cannot be null.");
    return;
}

try
{
    string clearText = Protector.Decrypt(cipherText, password2Decrypt);
    WriteLine($"Decrypted text: {clearText}");
}
catch (CryptographicException)
{
    WriteLine("You entered the wrong password!");
}
catch (Exception ex)
{
    WriteLine($"Non-cryptographic exception: {
        ex.GetType().Name}, {ex.Message}");
}
```

10. Выполните код. Введите, например, сообщение Hello Bob и пароль secret, затем повторите тот же пароль при расшифровке. Вы увидите результат, аналогичный приведенному в следующем выводе:

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
```

```
PBKDF2 algorithm: SHA256, Iteration count: 150,000
107 milliseconds to generate Key and IV.
WARNING: The elapsed time to generate the Key and IV may be too short to
provide a secure encryption key.
Encryption algorithm: AES-256, CBC mode with PKCS7 padding.
Encrypted text: eWt8sgL7aSt5DC9g740NEP07mj551XB/MmCZpUsFE0=
Enter the password: secret
Decrypted text: Hello Bob
```



Если время генерации ключа и вектора окажется меньше 100 мс, увеличивайте количество итераций до тех пор, пока не превысите этот порог. Имейте в виду, что при изменении количества итераций результат шифрования тоже изменится — шифротекст будет отличаться от приведенного выше.

- Еще раз выполните код. Введите сообщение и пароль для шифрования, но при расшифровке введите другой пароль. В результате будет выведено сообщение об ошибке:

```
Enter a message that you want to encrypt: Hello Bob
Enter a password: secret
PBKDF2 algorithm: SHA256, Iteration count: 150,000
134 milliseconds to generate Key and IV.
Encryption algorithm: AES-256, CBC mode with PKCS7 padding.
Encrypted text: eWt8sgL7aSt5DC9g740NEP07mj551XB/MmCZpUsFE0=
Enter the password: 123456
You entered the wrong password!
```



**Дельный совет:** чтобы в будущем было легче модернизировать алгоритм шифрования (например, перейти на более сильный алгоритм или увеличить размер ключа), сохраняйте информацию о текущих настройках. Для этого проекта можно зафиксировать: используется AES-256, режим CBC с добавлением байтов по схеме PKCS#7, стандарт PBKDF2, алгоритм хеширования SHA256 и количество итераций. Такая практика называется криптографической гибкостью.

## Хеширование данных

В .NET доступно несколько алгоритмов хеширования. Некоторые из них не используют ключей, другие — работают с симметричными или асимметричными ключами.

При выборе алгоритма важно учитывать два основных критерия.

- **Устойчивость к коллизиям** — насколько редко встречаются разные входные данные с одинаковым хешем.
- **Сопротивление к поиску прообраза** — насколько трудно найти другие данные, дающие тот же результат хеширования.

В таблице ниже перечислены распространенные алгоритмы хеширования без использования ключей:

Алгоритм	Размер хеша	Описание
<b>MD5</b>	16 байт	Быстрый, но уязвимый к коллизиям. Распространен
<b>SHA-1</b>	20 байт	Использование в интернете запрещено с 2011 года
<b>SHA-256</b> <b>SHA-384</b> <b>SHA-512</b>	32 байта 48 байт 64 байта	Представители второго поколения алгоритмов SHA ( <i>SHA-2</i> )
<b>SHA3-256</b> <b>SHA3-384</b> <b>SHA3-512</b>	32 байта 48 байт 64 байта	Третье поколение алгоритмов SHA ( <i>SHA-3</i> ), отличающихся размером выходного хеша. Они обеспечивают высокую криптостойкость, но работают медленнее по сравнению с SHA-2. Особенно это заметно на процессорах без специализированных инструкций для ускорения вычислений по алгоритму Кессак. Например, SHA2-512 обычно более чем вдвое быстрее, чем SHA3-512



**Дельный совет:** старайтесь не использовать MD5 и SHA-1 — у этих алгоритмов есть известные уязвимости. Лучше выбрать SHA-2 или SHA-3 с подходящей длиной хеша. Первая коллизия MD5 была обнаружена в 2010 году, для SHA-1 — в 2017 году. Подробности можно прочитать на странице по адресу [arstechnica.co.uk/information-technology/2017/02/at-deaths-door-for-years-widely-used-sha1-function-is-now-dead/](http://arstechnica.co.uk/information-technology/2017/02/at-deaths-door-for-years-widely-used-sha1-function-is-now-dead/).

## Хеширование с помощью популярного алгоритма SHA-256

Выбор между SHA-256, SHA-384 и SHA-512 зависит от ваших требований к безопасности, производительности и особенностям оборудования. SHA-256 подходит для большинства задач, где важна защита данных. Он остается устойчивым к известным криптоатакам, связанным с целостностью или коллизиями, и работает быстрее на 32-битных системах благодаря меньшему размеру хеша и использованию 32-битных операций. Кроме того, он часто необходим для соблюдения требований протоколов и стандартов безопасности.

SHA-384 формирует 384-битный хеш и использует увеличенный блок по сравнению с SHA-256. Это повышает устойчивость к коллизионным атакам, хотя может немного замедлить работу. В 64-битных системах производительность SHA-384 может быть сопоставимой или даже выше, чем у SHA-256. Такой алгоритм выбирают, если нормативы требуют хеш длиннее 256 бит, но применение SHA-512 выглядит чрезмерным.

SHA-512 создает хеш длиной 512 бит и считается самым надежным в семействе SHA-2. Он защищен от всех известных видов атак, включая коллизии и поиск прообраза. В системах с 64-битной архитектурой SHA-512 показывает отличную производительность, так как использует 64-битные операции, полностью соответствующие возможностям современных процессоров. Этот алгоритм подходит для задач, в которых особенно важна защита критически важных данных.

Если вам нужен максимальный уровень безопасности — стоит выбрать SHA-512. Однако для большинства случаев SHA-256 дает отличное соотношение скорости и защищенности, поэтому в этой главе мы будем использовать именно его.

Дополнительно: попробуйте выполнить тестирование всех трех алгоритмов на своей системе, чтобы сравнить их производительность.



В .NET 8 добавлена поддержка алгоритмов семейства SHA-3, включая SHAKE-128 и SHAKE-256. Эти алгоритмы отличаются от привычных хеш-функций с фиксированной длиной, таких как SHA-256. SHAKE-128 и SHAKE-256 относятся к классу функций с расширяемым выходом (Extendable Output Functions, XOF), то есть могут генерировать хеш любой необходимой длины. Благодаря этому они особенно удобны в тех случаях, когда нужно получить выходные данные нестандартного размера. Поддержка SHA-3 доступна в Windows 11 начиная с версии сборки 25324, а также в Linux при наличии OpenSSL версии 1.1.1 и выше.

Теперь мы добавим класс, который будет представлять пользователя. Эти данные могут храниться в памяти, файле или базе данных. Чтобы в памяти можно было работать сразу с несколькими пользователями, удобно использовать словарь.

1. В проекте библиотеки классов `CryptographyLib` создайте файл `User.cs`, удалите из этого файла все содержимое и определите запись с тремя свойствами: имя пользователя, соль и хеш пароля с солью:

```
namespace Packt.Shared;

public record class User(string Name, string Salt,
    string SaltedHashedPassword);
```

2. В файле `Protector.cs` добавьте объявление словаря пользователей, а также определите два метода — для регистрации нового пользователя и для проверки его пароля при входе в систему:

```
private static Dictionary<string, User> Users = new();

public static User Register(string username,
    string password)
{
    // Генерирует случайную соль
    RandomNumberGenerator rng = RandomNumberGenerator.Create();
    byte[] saltBytes = new byte[16];
```

```
rng.GetBytes(saltBytes);
string saltText = ToBase64String(saltBytes);

// Создает хеш пароля с использованием соли
string saltedhashedPassword = SaltAndHashPassword(password, saltText);

User user = new(username, saltText, saltedhashedPassword);

Users.Add(user.Name, user);

return user;
}

// Проверяет пароль пользователя по данным в словаре Users
public static bool CheckPassword(string username, string password)
{
    if (!Users.ContainsKey(username))
    {
        return false;
    }

    User u = Users[username];

    return CheckPassword(password,
        u.Salt, u.SaltedHashedPassword);
}

// Сравнивает введенный пароль с сохраненным хешем, используя соль
public static bool CheckPassword(string password,
    string salt, string hashedPassword)
{
    // Повторно вычисляет хеш с той же солью для сравнения
    string saltedHashedPassword = SaltAndHashPassword(
        password, salt);

    return (saltedHashedPassword == hashedPassword);
}

private static string SaltAndHashPassword(string password, string salt)
{
    using (SHA256 sha = SHA256.Create())
    {
        string saltedPassword = password + salt;
        return ToBase64String(sha.ComputeHash(
            Encoding.Unicode.GetBytes(saltedPassword)));
    }
}
```

3. В редакторе кода создайте консольное приложение `HashingApp` и добавьте его в решение `Chapter08`.

4. В файле `HashingApp.csproj` проекта включите обработку предупреждений как ошибок, добавьте ссылку на библиотеку `CryptographyLib`, а также выполните глобальный статический импорт класса `System.Console`.
5. Выполните сборку проекта `HashingApp`.
6. Удалите содержимое в файле `Program.cs` проекта `HashingApp`, а затем код для регистрации пользователя Alice и запроса на регистрацию пользователя Bob. Затем предложите войти в систему под одним из имен и проверьте пароль:

```
using Packt.Shared; // Импортирует класс Protector

WriteLine("Registering Alice with Pa$$w0rd:");
User alice = Protector.Register("Alice", "Pa$$w0rd");

WriteLine($" Name: {alice.Name}");
WriteLine($" Salt: {alice.Salt}");
WriteLine($" Password (salted and hashed): {alice.SaltedHashedPassword}");
WriteLine();

Write("Enter a new user to register: ");
string? username = ReadLine();
if (string.IsNullOrEmpty(username)) username = "Bob";

Write($"Enter a password for {username}: ");
string? password = ReadLine();
if (string.IsNullOrEmpty(password)) password = "Pa$$w0rd";

WriteLine("Registering a new user:");
User newUser = Protector.Register(username, password);
WriteLine($" Name: {newUser.Name}");
WriteLine($" Salt: {newUser.Salt}");
WriteLine($" Password (salted and hashed): {newUser.SaltedHashedPassword}");
WriteLine();

bool correctPassword = false;

while (!correctPassword)
{
    Write("Enter a username to log in: ");
    string? loginUsername = ReadLine();
    if (string.IsNullOrEmpty(loginUsername))
    {
        WriteLine("Login username cannot be empty.");
        Write("Press Ctrl+C to end or press ENTER to retry.");
        ReadLine();
        continue; // Возврат к циклу while
    }

    Write("Enter a password to log in: ");
    string? loginPassword = ReadLine();
```

```

if (string.IsNullOrEmpty(loginPassword))
{
    WriteLine("Login password cannot be empty.");
    Write("Press Ctrl+C to end or press ENTER to retry.");
    ReadLine();
    continue;
}

correctPassword = Protector.CheckPassword(
    loginUsername, loginPassword);

if (correctPassword)
{
    WriteLine($"Correct! {loginUsername} has been logged in.");
}
else
{
    WriteLine("Invalid username or password. Try again.");
}
}

```

7. Запустите приложение, зарегистрируйте пользователя Bob с тем же паролем, что и у Alice, и изучите результат — за счет разной соли хеши будут различаться, как показано в следующем выводе:

```

Registering Alice with Pa$$w0rd:
Name: Alice
Salt: I1I1dzIjkd7EYDf/6jaf4w==
Password (salted and hashed): pIoadjE4W/
XaRFkqS3br3UuAuPv/3LVQ8kj6mvcz+s=

Enter a new user to register: Bob
Enter a password for Bob: Pa$$w0rd
Registering a new user:
Name: Bob
Salt: 1X7ym/UjxTiuEWBC/vIHpw==
Password (salted and hashed):
DoBFtDhKeN0aaaLvDErtrZ3mpZSvpWDQ9TXDosTq0sQ=


Enter a username to log in: Alice
Enter a password to log in: secret
Invalid username or password. Try again.
Enter a username to log in: Bob
Enter a password to log in: secret
Invalid username or password. Try again.
Enter a username to log in: Bob
Enter a password to log in: Pa$$w0rd
Correct! Bob has been logged in.

```



Несмотря на то что два пользователя выбрали одинаковый пароль, каждому из них присваивается своя случайная соль. Поэтому итоговые хеши паролей с солью различаются.

## Создание цифровой подписи

Чтобы подтвердить подлинность данных и источник, которому можно доверять, используют цифровую подпись. Подписываются не сами данные, а их хеш — большинство алгоритмов подписи начинают с вычисления хеша перед созданием подписи. Некоторые из них также позволяют подписывать уже подготовленный хеш.

В этом примере будет использоваться алгоритм SHA-256 для генерации хеша и RSA для создания подписи.

Можно взять и алгоритм DSA — он подходит как для хеширования, так и для подписи. DSA быстрее при генерации подписи, но медленнее при проверке. Поскольку подпись обычно создается один раз, а проверяется многократно, приоритет отдается алгоритмам с быстрой проверкой. Поэтому в большинстве сценариев RSA предпочтительнее DSA.



**Дельный совет:** сегодня алгоритм DSA используется все реже. На смену ему пришел стандарт Elliptic Curve DSA (ECDSA) — усовершенствованный аналог, который обеспечивает ту же степень безопасности, но при этом создает более компактные подписи. Хотя он и работает медленнее, чем RSA, в ряде случаев ECDSA считается более эффективным.

## Создание подписи с помощью SHA-256 и RSA

Рассмотрим, как подписывать данные и проверять подпись с помощью открытого ключа.

1. В классе `Protector` добавьте поле для хранения открытого ключа в виде строки, а также реализуйте два метода — один для создания подписи, второй для ее проверки:

```
public static string? PublicKey;

public static string GenerateSignature(string data)
{
    byte[] dataBytes = Encoding.Unicode.GetBytes(data);
    SHA256 sha = SHA256.Create();
    byte[] hashedData = sha.ComputeHash(dataBytes);
    RSA rsa = RSA.Create();

    PublicKey = rsa.ToXmlString(false); // экспорт закрытого ключа

    return ToBase64String(rsa.SignHash(hashedData,
        HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1));
}

public static bool ValidateSignature(
    string data, string signature)
{
    if (PublicKey is null) return false;
```

```

byte[] dataBytes = Encoding.Unicode.GetBytes(data);
SHA256 sha = SHA256.Create();

byte[] hashedData = sha.ComputeHash(dataBytes);
byte[] signatureBytes = FromBase64String(signature);

RSA rsa = RSA.Create();
rsa.FromXmlString(PublicKey);

return rsa.VerifyHash(hashedData, signatureBytes,
    HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
}

```

Несколько важных моментов по приведенному коду.

- Проверка подписи требует только открытой части ключа. Поэтому при вызове метода `ToXmlString` нужно передать значение `false`, чтобы экспортировать только открытую часть. Закрытая часть используется для создания подписи и должна оставаться конфиденциальной — если кто-то получит доступ к закрытому ключу, он сможет подписывать данные от вашего имени.
- Алгоритм хеширования, применяемый при подписи, должен совпадать с алгоритмом, который указывается при проверке подписи. В приведенном примере использовался `SHA256`.

Теперь можно перейти к тестированию: подпишем данные и проверим подпись.

2. В редакторе кода создайте консольное приложение `SigningApp` и добавьте его в решение `Chapter08`.
3. В файле созданного проекта включите обработку предупреждений как ошибок, добавьте ссылку на проект `CryptographyLib`, а также выполните глобальный статический импорт класса `System.Console`.
4. Выполните сборку проекта `SigningApp` и убедитесь в отсутствии ошибок компиляции.
5. В файле `Program.cs` удалите все содержимое, а затем импортируйте пространство имен `Packt.Shared`. Также добавьте код, предлагающий пользователю ввести текст, подписать его и проверить подпись, а затем измените текст и повторно выполните проверку, чтобы смоделировать ситуацию, в которой подпись становится недействительной:

```

using Packt.Shared; // Подключение класса Protector

Write("Enter some text to sign: ");
string? data = ReadLine();

if (string.IsNullOrEmpty(data))
{
    WriteLine("You must enter some text.");
    return; // Завершение работы приложения
}

string signature = Protector.GenerateSignature(data);

```

```

WriteLine($"Signature: {signature}");
WriteLine("Public key used to check signature:");
WriteLine(Protector.PublicKey);

if (Protector.ValidateSignature(data, signature)){
    WriteLine("Correct! Signature is valid. Data has not been manipulated.");
}
else
{
    WriteLine("Invalid signature or the data has been manipulated.");
}

// Имитация изменения текста: замена первого символа на X
// (или на Y, если он уже X)
char newFirstChar = 'X';

if (data[0] == newFirstChar)
{
    newFirstChar = 'Y';
}

string manipulatedData = $"{newFirstChar}{data.Substring(1)}";

if (Protector.ValidateSignature(manipulatedData, signature))
{
    WriteLine("Correct! Signature is valid. Data has not been manipulated.");
}
else
{
    WriteLine($"Invalid signature or manipulated data: {manipulatedData}");
}

```

6. Запустите приложение и введите произвольный текст. Ниже приведен пример вывода (фрагментированный):

```

Enter some text to sign: The cat sat on the mat.
Signature: BXStdM...4Wrg==
Public key used to check signature:
<RSAKeyValue><Modulus>nHtwl3...mw3w==</Modulus><Exponent>AQAB</
Exponent></RSAKeyValue>
Correct! Signature is valid. Data has not been manipulated.
Invalid signature or manipulated data: Xhe cat sat on the mat.

```

## Генерация случайных чисел для криптографии

Во многих криптографических сценариях — например, при шифровании или подписи — необходимо генерировать случайные числа. Однако важно понимать, что не все генераторы случайных чисел одинаково надежны.

Класс `Random` в .NET генерирует псевдослучайные числа, которые не подходят для задач, связанных с безопасностью. Такие числа могут быть предсказуемыми, и при использовании их в криптографии защита может оказаться уязвимой.

Для генерации криптографически стойких псевдослучайных чисел следует использовать типы, производные от `RandomNumberGenerator`. Обычно используется вызов фабричного метода `RandomNumberGenerator.Create()`, который возвращает подходящую реализацию по умолчанию или может быть вызван с указанием конкретного алгоритма.

Далее мы добавим метод, который создает массив байтов со случайными значениями. Его можно применять, например, для генерации ключей или векторов инициализации в алгоритмах шифрования.

1. В классе `Protector` добавьте выражения для определения метода, возвращающего массив криптографически случайных байтов. Этот метод можно будет использовать для генерации ключа шифрования или вектора инициализации:

```
public static byte[] GetRandomKeyOrIV(int size)
{
    RandomNumberGenerator r = RandomNumberGenerator.Create();
    byte[] data = new byte[size];

    // Массив заполняется криптографически стойкими случайными байтами
    r.GetBytes(data);
    return data;
}
```

Теперь вы сможете проверить, как выглядят случайные байты, сгенерированные для действительно случайного ключа или вектора инициализации.

2. В редакторе кода создайте проект консольного приложения с именем `RandomizingApp` и добавьте его в решение `Chapter08`.
3. В файле созданного проекта включите обработку предупреждений как ошибок, добавьте ссылку на библиотеку `CryptographyLib` и настройте глобальный статический импорт класса `System.Console`.
4. Выполните сборку проекта `RandomizingApp`, чтобы проверить, что компиляция проходит без ошибок.
5. Удалите содержимое файла `Program.cs`, а затем подключите в нем пространство имен `Packt.Shared` и добавьте код, предлагающий пользователю ввести желаемую длину массива. После этого вызовите метод для генерации случайных байтов и отобразите их в консоли:

```
using Packt.Shared; // Подключение класса Protector

Write("How big do you want the key (in bytes): ");
string? size = ReadLine();

if (string.IsNullOrEmpty(size))
{
    WriteLine("You must enter a size for the key.");
    return; // Выход из приложения
}
```

```
byte[] key = Protector.GetRandomKeyOrIV(int.Parse(size));  
  
WriteLine($"Key as byte array:");  
for (int b = 0; b < key.Length; b++)  
{  
    Write($"{key[b]:x2} ");  
    if (((b + 1) % 16) == 0) WriteLine();  
}  
WriteLine();
```

6. Запустите приложение. Введите, к примеру, значение 256, чтобы получить массив длиной 256 байтов, и изучите результат, выведенный в консоль:

```
How big do you want the key (in bytes): 256  
Key as byte array:  
f1 57 3f 44 80 e7 93 dc 8e 55 04 6c 76 6f 51 b9  
e8 84 59 e5 8d eb 08 d5 e6 59 65 20 b1 56 fa 68  
...
```

## Проверка подлинности и авторизация пользователей

*Проверка подлинности* — это процесс, с помощью которого организация удостоверяется в том, что пользователь действительно тот, за кого себя выдает. Чаще всего для этого используются имя пользователя и пароль, но могут применяться и другие способы — например, отпечатки пальцев или сканирование лица. После прохождения проверки подлинности приложение или сервис может использовать сведения о пользователе — например, его адрес электронной почты или список групп/ролей, в которые он входит.

*Авторизация* — это процесс, в ходе которого система проверяет, к каким группам или ролям принадлежит пользователь, чтобы определить, разрешить ли ему доступ к определенным ресурсам: функциям приложения, файлам, данным. Даже если в группе только один пользователь, рекомендуется использовать авторизацию по ролям: так легче управлять доступом. Если права доступа назначены не конкретному пользователю, а роли, в будущем достаточно просто заменить одного участника роли на другого — без необходимости изменять сами разрешения.

Например, вместо того чтобы назначать права доступа к Белому дому и другим объектам Дональду Трампу (пользователю), вы назначите эти права Президенту США (роль), а затем добавите Трампа в качестве единственного участника этой роли. Тогда, когда произойдут выборы президента, вам не придется заново назначать все права — достаточно будет исключить одного пользователя из роли и включить другого. Такой подход позволяет упростить управление доступом и делает систему более гибкой — особенно если реализовать преемственность в виде очереди.

## Механизмы проверки подлинности и авторизации

Существует множество способов организовать проверку подлинности и авторизацию пользователей. Все они опираются на реализацию пары интерфейсов из пространства имен `System.Security.Principal` — `IIdentity` и `IPrincipal`.

### Идентификация пользователя

Интерфейс `IIdentity` представляет пользователя. Он содержит, в частности, свойство `Name`, указывающее имя пользователя, и `IsAuthenticated`, определяющее, прошел ли пользователь проверку подлинности или остался анонимным.

```
namespace System.Security.Principal;

public interface IIdentity
{
    string? AuthenticationType { get; }
    bool IsAuthenticated { get; }
    string? Name { get; }
}
```

Одним из распространенных классов, реализующих этот интерфейс, является `GenericIdentity`. Этот класс, в свою очередь, наследуется от `ClaimsIdentity` — более продвинутого типа.

```
namespace System.Security.Principal;

public class GenericIdentity : ClaimsIdentity
{
    public GenericIdentity(string name);
    public GenericIdentity(string name, string type);
    protected GenericIdentity(GenericIdentity identity);
    public override string AuthenticationType { get; }
    public override IEnumerable<Claim> Claims { get; }
    public override bool IsAuthenticated { get; }
    public override string Name { get; }
    public override ClaimsIdentity Clone();
}
```

Объекты `Claim` содержат свойство `Type`, которое указывает, какого рода утверждение задано: имя пользователя, принадлежность к роли или группе, дата рождения и т. д.

```
namespace System.Security.Claims;

public class Claim
{
    // различные конструкторы Claim
```

```

public string Type { get; }
public ClaimsIdentity? Subject { get; }
public IDictionary<string, string> Properties { get; }
public string OriginalIssuer { get; }
public string Issuer { get; }
public string ValueType { get; }
public string Value { get; }
protected virtual byte[]? CustomSerializationData { get; }
public virtual Claim Clone();
public virtual Claim Clone(ClaimsIdentity? identity);
public override string ToString();
public virtual void WriteTo(BinaryWriter writer);
protected virtual void WriteTo(BinaryWriter writer, byte[]? userData);
}

public static class ClaimTypes
{
    public const string Actor = "http://schemas.xmlsoap.org/ws/2009/09/identity/
        claims/actor";
    public const string NameIdentifier = "http://schemas.xmlsoap.org/ws/2005/05/
        identity/claims/nameidentifier";
    public const string Name = "http://schemas.xmlsoap.org/ws/2005/05/identity/
        claims/name";
    public const string PostalCode = "http://schemas.xmlsoap.org/ws/2005/05/
        identity/claims/postalcode";

    // Множество других строковых констант для типа Claim

    public const string MobilePhone = "http://schemas.xmlsoap.org/ws/2005/05/
        identity/claims/mobilephone";
    public const string Role = "http://schemas.microsoft.com/ws/2008/06/identity/
        claims/role";
    public const string Webpage = "http://schemas.xmlsoap.org/ws/2005/05/
        identity/claims/webpage";
}

```

## Членство пользователя

Интерфейс `IPrincipal` связывает идентификатор пользователя с ролями и группами, в которые он входит. Поэтому его можно применять для авторизации и проверки прав доступа:

```

namespace System.Security.Principal;

public interface IPrincipal
{
    IIdentity? Identity { get; }
    bool IsInRole(string role);
}

```

У текущего потока, выполняющего ваш код, есть свойство `CurrentPrincipal`, которому можно присвоить любой объект, реализующий интерфейс `IPrincipal`.

Это свойство затем используется для проверки разрешений на выполнение защищенных действий. Наиболее распространенный класс, реализующий этот интерфейс, — `GenericPrincipal`, который наследуется от класса `ClaimsPrincipal`:

```
namespace System.Security.Principal;

public class GenericPrincipal : ClaimsPrincipal
{
    public GenericPrincipal(IIdentity identity, string[]? roles);
    public override IIdentity Identity { get; }
    public override bool IsInRole([NotNullWhen(true)] string? role);
}
```

## Реализация проверки подлинности и авторизации

Рассмотрим базовую реализацию пользовательской проверки подлинности и авторизации.

1. В проекте `CryptographyLib` в записи `User.cs` добавьте свойство для хранения массива ролей:

```
namespace Packt.Shared;

public record class User(string Name, string Salt,
    string SaltedHashedPassword, string[]? Roles);
```

2. В файле `Protector.cs` настройте метод `Register` на прием массива ролей в качестве необязательного параметра:

```
public static User Register(string username,
    string password, string[]? roles = null)
```

3. В методе `Register` передайте этот параметр в создаваемый объект `User`:

```
User user = new(username, saltText,
    saltedhashedPassword, roles);
```

4. В верхней части файла `Protector.cs` добавьте оператор импорта пространства имен для работы с идентификатором пользователя:

```
// Подключение классов GenericIdentity, GenericPrincipal
using System.Security.Principal;
```

5. В классе `Protector` добавьте метод `LogIn`. Если имя пользователя и пароль совпадают, создайте объект `GenericIdentity`, затем объект `GenericPrincipal` и установите его в свойство `CurrentPrincipal` текущего потока. В качестве типа проверки подлинности укажите строку "PacktAuth":

```

public static void LogIn(string username, string password)
{
    if (CheckPassword(username, password))
    {
        GenericIdentity gi = new(
            name: username, type: "PacktAuth");

        GenericPrincipal gp = new(
            identity: gi, roles: Users[username].Roles);

        // Установка объекта GenericPrincipal в текущем потоке – это значение
        // будет использоваться по умолчанию для авторизации.
        Thread.CurrentPrincipal = gp;
    }
}

```

6. В редакторе кода создайте консольное приложение с именем `SecureApp` и добавьте его в решение `Chapter08`.
7. В файле `SecureApp.csproj` проекта включите обработку предупреждений как ошибок, добавьте ссылку на проект `CryptographyLib` и выполните глобальный статический импорт класса `System.Console`.
8. Выполните сборку проекта и удостоверьтесь, что ошибок компиляции нет.
9. В файле `Program.cs` данного проекта удалите содержимое, а затем импортируйте пространства имен для проверки подлинности и авторизации пользователей:

```

using Packt.Shared; // Подключение класса Protector
using System.Security.Principal; // Подключение интерфейса IPrincipal
using System.Security.Claims; // Подключение объектов ClaimsPrincipal, Claim

```

10. В файле `Program.cs` проекта `SecureApp` добавьте код, который:
  - регистрирует пользователей `Alice`, `Bob` и `Eve` с разными ролями;
  - предлагает пользователю ввести имя и пароль;
  - выполняет вход и выводит информацию об авторизованном пользователе.

```

WriteLine("Registering Alice, Bob, and Eve with passwords Pa$$w0rd.");

Protector.Register("Alice", "Pa$$w0rd", roles: new[] { "Admins" });

Protector.Register("Bob", "Pa$$w0rd",
    roles: new[] { "Sales", "TeamLeads" });

// Зарегистрировать пользователя Eve, не включенного ни в одну из ролей
Protector.Register("Eve", "Pa$$w0rd");

WriteLine();

// Запросить у пользователя имя и пароль для входа в систему
// Войти в систему от имени одного из трех пользователей

```

```

Write("Enter your username: ");
string? username = ReadLine()!;

Write("Enter your password: ");
string? password = ReadLine()!;

Protector.LogIn(username, password);

if (Thread.CurrentPrincipal == null)
{
    WriteLine("Log in failed.");
    return; // Завершить работу приложения
}

IPrincipal p = Thread.CurrentPrincipal;

WriteLine($"IsAuthenticated: {p.Identity?.IsAuthenticated}");
WriteLine($"AuthenticationType: {p.Identity?.AuthenticationType}");
WriteLine($"Name: {p.Identity?.Name}");
WriteLine($"IsInRole(\"Admins\"): {p.IsInRole("Admins")}");
WriteLine($"IsInRole(\"Sales\"): {p.IsInRole("Sales")}");

if (p is ClaimsPrincipal principal)
{
    WriteLine($"{principal.Identity?.Name} has the following claims:");

    foreach (Claim claim in principal.Claims)
    {
        WriteLine($"{claim.Type}: {claim.Value}");
    }
}

```

11. Запустите проект и войдите под именем `Alice` с паролем `Pa$$word`. В консоли отобразится информация о текущем пользователе и его ролях:

```

Registering Alice, Bob, and Eve with passwords Pa$$w0rd.

Enter your username: Alice
Enter your password: Pa$$w0rd
IsAuthenticated: True
AuthenticationType: PacktAuth
Name: Alice
IsInRole("Admins"): True
IsInRole("Sales"): False
Alice has the following claims:
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: Alice
http://schemas.microsoft.com/ws/2008/06/identity/claims/role: Admins

```

12. Повторите запуск и войдите под именем `Alice`, но с паролем `secret`, чтобы увидеть сообщение об ошибке проверки подлинности:

```
Enter your username: Alice
Enter your password: secret
Log in failed.
```

13. Повторите запуск и войдите под именем **Bob** с паролем **Pa\$\$word**, чтобы увидеть его роль и статус входа:

```
Enter your username: Bob
Enter your password: Pa$$w0rd
IsAuthenticated: True
AuthenticationType: PacktAuth
Name: Bob
IsInRole("Admins"): False
IsInRole("Sales"): True
Bob has the following claims:
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name: Bob
http://schemas.microsoft.com/ws/2008/06/identity/claims/role: Sales
http://schemas.microsoft.com/ws/2008/06/identity/claims/role: TeamLeads
```

Вы научились регистрировать пользователей с паролями и назначать им роли. Теперь пришло время использовать эту информацию для авторизации — чтобы управлять доступом к функциям приложения.

## Защита функциональности приложения

Рассмотрим, как с помощью авторизации можно ограничить доступ к определенным возможностям приложения.

- В верхней части файла `Program.cs` проекта `SecureApp` добавьте оператор импорта пространства имен `System.Security`, которое используется для работы с исключениями безопасности:

```
using System.Security; // Подключение класса SecurityException
```

- В нижней части того же файла определите метод, доступ к которому защищается проверкой разрешений. Метод должен выбрасывать исключения, если пользователь анонимен или не входит в роль `Admins`:

```
static void SecureFeature()
{
    if (Thread.CurrentPrincipal is null)
    {
        throw new SecurityException(
            "A user must be logged in to access this feature.");
    }

    if (!Thread.CurrentPrincipal.IsInRole("Admins"))
    {
        throw new SecurityException(
            "Only Admins can access this feature.");
    }
}
```

```

        "User must be a member of Admins to access this feature.");
    }

    WriteLine("You have access to this secure feature.");
}

```

3. Над методом `SecureFeature` добавьте выражения, вызывающие этот метод внутри блока `try`. Если будет выброшено исключение, обработайте его в блоке `catch`:

```

try
{
    SecureFeature();
}
catch (Exception ex)
{
    WriteLine($"{ex.GetType()} : {ex.Message}");
}

```

4. Запустите код, выполните вход под именем `Alice` с паролем `Pa$$word` и обратите внимание на результат — доступ к защищенной функции будет предоставлен:

```
You have access to this secure feature.
```

5. Затем запустите код повторно, войдите под именем `Bob` с тем же паролем и проанализируйте результат — доступ будет запрещен, и в консоли отобразится сообщение об отказе:

```
System.Security.SecurityException: User must be a member of Admins to access
this feature.
```

## Проверка подлинности и авторизация в реальных приложениях

Хотя полезно изучить, как работает проверка подлинности и авторизация на примерах, в практических проектах не рекомендуется создавать собственные механизмы безопасности. Слишком велик риск ошибок, которые могут сделать приложение уязвимым.

Вместо этого лучше использовать проверенные решения — коммерческие или с открытым исходным кодом. Такие решения обычно реализуют отраслевые стандарты, например OAuth 2.0 и OpenID Connect. Официальная рекомендация Microsoft гласит: «В Microsoft уже есть команда и продукт для этой задачи — Azure Active Directory, которым можно бесплатно управлять до 500 000 объектов». Подробнее об этом читайте на сайте [devblogs.microsoft.com/aspnet/asp-net-core-6-and-authentication-servers/](https://devblogs.microsoft.com/aspnet/asp-net-core-6-and-authentication-servers/).

Полный стек OAuth 2.0 и OpenID Connect для .NET можно найти на сайте [github.com/openiddict](https://github.com/openiddict).

## Новые возможности .NET 9

В .NET 9 планируется улучшить поддержку криптографических алгоритмов, в числе которых единоразовое хеширование (one-shot hash) в классе `CryptographicOperations` и новые реализации стандарта Keccak Message Authentication Code (KMAC).

### Метод `CryptographicOperations.HashData`

Новый API `CryptographicOperations.HashData` обеспечивает хеширование данных в виде единой операции. Это предпочтительный подход, поскольку он минимизирует выделение памяти и обеспечивает высокую производительность. В качестве алгоритма можно указать структуру `HashAlgorithmName`. Примеры уже существующих методов с поддержкой one-shot API:

- `SHA256.HashData()`
- `HMACSHA256.HashData()`

В .NET 9 появится API `CryptographicOperations.HashData`, который позволит выполнять хеширование в виде одной операции. Алгоритм, который будет использоваться, задается через структуру `HashAlgorithmName`:

```
static void HashAndProcessData(HashAlgorithmName hashAlgorithmName, byte[] data)
{
    byte[] hash = CryptographicOperations.HashData(hashAlgorithmName, data);
    ProcessHash(hash);
}
```

### Алгоритм KMAC

KMAC — это псевдослучайная функция/хеш-функция с ключом, основанная на алгоритме Keccak. Она описана в спецификации NIST SP800-185 и может использоваться как псевдослучайная функция. В .NET 9 реализованы четыре варианта: `Kmac128`, `Kmac256`, `KmacXof128` и `KmacXof256`.

Поддержка KMAC зависит от платформы. Она доступна в Linux с установленным OpenSSL 3.0 или новее, а также в Windows 11 начиная со сборки 26016. Проверить доступность алгоритма на текущей платформе можно с помощью статического свойства `IsSupported`:

```
if (Kmac128.IsSupported)
{
    byte[] key = GetKmacKey();
    byte[] input = GetInputToMac();
    byte[] mac = Kmac128.HashData(key, input, outputLength: 32);
}
else
{
    // Обработка ситуации, когда алгоритм KMAC недоступен
}
```

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 8.1. Дополнительные онлайн-материалы

Ознакомьтесь с рекомендациями и лучшими практиками Microsoft по использованию шифрования на сайте [learn.microsoft.com/ru-ru/security/sdl/cryptographic-recommendations](https://learn.microsoft.com/ru-ru/security/sdl/cryptographic-recommendations). Документ основан на внутренних стандартах Microsoft, разработанных в рамках жизненного цикла безопасной разработки (Security Development Lifecycle, SDL).

### Упражнение 8.2. Практические задания

В решение Chapter08 добавьте консольное приложение Ch08Ex02\_EncryptData, которое будет использоваться для защиты конфиденциальных данных — например, номеров банковских карт и паролей, хранящихся в XML-файле:

```
<?xml version="1.0" encoding="utf-8" ?>
<customers>
    <customer>
        <name>Bob Smith</name>
        <creditcard>1234-5678-9012-3456</creditcard>
        <password>Pa$$w0rd</password>
    </customer>
    ...
</customers>
```

В исходном XML-файле номер карты и пароль клиента представлены в открытом виде. Номер банковской карты следует зашифровать, чтобы в дальнейшем его можно было расшифровать и использовать. Пароль необходимо хешировать с солью.



**Дельный совет:** не храните номера кредитных карт в своих приложениях. Этот пример приведен только для демонстрации защиты конфиденциальной информации. Если вашей программе действительно требуется хранить такие данные, потребуется выполнить гораздо больше шагов, чтобы ваше ПО соответствовало требованиям стандарта безопасности индустрии платежных карт (PCI DSS).

Затем добавьте еще одно консольное приложение, Ch08Ex03\_DecryptData, которое будет открывать XML-файл, созданный предыдущим приложением, и расшифровывать зашифрованный номер банковской карты.

### Упражнение 8.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Какой алгоритм из стандартной библиотеки .NET рекомендуется использовать для симметричного шифрования?
2. Какой алгоритм из стандартной библиотеки .NET предпочтительнее для асимметричного шифрования?
3. Что такое радужная таблица и как она используется при взломе хешей?
4. Для алгоритмов симметричного шифрования предпочтительнее больший или меньший размер блока?
5. Что такое криптографический хеш и для чего он нужен?
6. Что такое криптографическая подпись и как она используется?
7. В чем разница между симметричным и асимметричным шифрованием?
8. Что означает аббревиатура RSA?
9. Зачем к паролям добавляют соль перед их сохранением?
10. Почему использование алгоритма SHA-1 считается небезопасным и не рекомендуется?

## Упражнение 8.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-08---protecting--data-using-cryptography](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-08---protecting--data-using-cryptography).

## Резюме

В этой главе вы узнали, как:

- шифровать и расшифровывать данные с помощью симметричного шифрования;
- хешировать данные с добавлением соли;
- подписывать данные и проверять подписи;
- генерировать криптографически стойкие случайные числа;
- использовать проверку подлинности и авторизацию для ограничения доступа к функциям приложения.

В следующей главе речь пойдет о создании собственного чат-сервиса на основе большой языковой модели (LLM).

# 9

## Создание чат-сервиса на основе LLM

Интеграция *большой языковой модели* (*Large Language Model, LLM*), например GPT-4 от компании OpenAI, в .NET-проект может значительно расширить его функциональность — в том числе за счет дополненной обработки *естественного языка* (*Natural Language Processing, NLP*), генерации текста и других интеллектуальных возможностей.

В этой главе мы создадим чат-сервис на основе LLM, дополнив стандартную модель пользовательской информацией — в частности, биографией автора книги и сведениями из базы данных Northwind. Благодаря этому пользователи смогут задавать вопросы об авторе и вымышленной корпорации.



На момент написания этой книги облачные API LLM предоставлялись на платной основе. В примерах используется один из наиболее доступных вариантов — GPT-3.5 Turbo компании OpenAI. Как говорится на сайте компании, «GPT-3.5 Turbo — наша быстрая и недорогая модель для более простых задач. gpt-3.5-turbo-0125 — флагманская модель этого семейства, поддерживает контекстное окно объемом 16 тысяч токенов и оптимизирована для диалоговых сценариев». Стоимость: 0,5 доллара США за 1 млн входных токенов и 1,5 доллара США за 1 млн выходных. Во время написания этой главы затраты на выполнение примеров составили менее 0,03 доллара.

Альтернатива — GPT-4o, самая продвинутая мультимодальная модель OpenAI. Она быстрее и дешевле GPT-4 Turbo и при этом обладает расширенными возможностями обработки изображений. Контекстное окно модели составляет 128 000 токенов, а ее обучение завершилось в октябре 2023 года. Стоимость: 5 долларов за 1 млн входных токенов и 15 долларов за 1 млн выходных.

### Обзор технологии LLM

Термины LLM (Large Language Model — большая языковая модель) и GPT (Generative Pre-trained Transformer — генеративный предобученный трансформер) часто воспринимаются как синонимы, но между ними есть важные различия.

- *LLM* — обобщенный термин, обозначающий *любую* языковую модель с миллиардным числом параметров, обученную на больших объемах текстовых данных.

Такие модели решают широкий спектр задач, связанных с языком: от перевода и резюмирования до ответов на вопросы и генерации текста.

- *GPT* – это конкретная серия LLM, разработанная компанией OpenAI. Аналогия со смартфонами: GPT среди языковых моделей – это как iPhone среди смартфонов. Сама аббревиатура указывает на архитектуру – «трансформер», предложенную в 2017 году. Эта архитектура особенно эффективна для работы с последовательностями, например с текстом. Все модели GPT построены на базе трансформеров, но не все LLM используют именно такую архитектуру.

Таким образом, все модели GPT – это LLM, но не все LLM – это GPT.

## Принципы работы LLM

Большие языковые модели (LLM) основаны на методах машинного обучения, в частности – на подходах глубокого обучения.

LLM обучаются на огромных объемах текстовых данных – от книг и статей до материалов с сайтов вроде Stack Overflow и Reddit. Благодаря такому разнообразному корпусу модель приобретает широкое понимание языка, контекста и фактических сведений, актуальных на момент завершения обучения.



Подробности о партнерстве компаний Stack Overflow и OpenAI опубликованы на сайте [stack overflow.co/company/press/archive/openai-partnership](https://stack overflow.co/company/press/archive/openai-partnership). Информация о сотрудничестве компаний OpenAI и Reddit – на странице [openai.com/index/openai-and-reddit-partnership/](https://openai.com/index/openai-and-reddit-partnership/).

В основе LLM лежит тип архитектуры нейронной сети под названием *трансформер*. Эта архитектура использует несколько слоев внимания и нейронных сетей с прямой связью. Рассмотрим ключевые компоненты трансформера.

- **Механизмы внимания** – позволяют модели определять значимость отдельных слов во входной последовательности, помогая фокусироваться на наиболее важных участках текста при формировании ответа.
- **Полносвязные слои** – обрабатывают данные после внимания, преобразуя их на каждом уровне сети и обеспечивая выходной сигнал, который учитывает контекст и значимость.



Ограничения современных LLM во многом связаны с их архитектурой. Такие модели не обладают способностью к саморефлексии или самоналюдению. Исследования показывают, что системы, способные «обращаться к себе», то есть анализировать собственные выводы и действия, потенциально могут выйти на более высокий уровень когнитивных способностей – в частности, за счет непрерывного обучения и самооценки.

Следующим этапом обработки текста становится *токенизация* — преобразование входного текста в последовательность токенов. Токены представляют собой отдельные элементы: слова, части слов или даже символы. Такая структура упрощает работу модели с большими словарями и позволяет эффективно обрабатывать текст на разных языках. Например, фраза `Hello, world!` может быть разбита на токены:

```
["Hello", " ", "world", "!"]
```

Процесс обучения LLM включает несколько ключевых этапов.

1. **Прямой проход** — входной текст передается через модель, которая формирует предсказание. Например, получив часть предложения, модель прогнозирует, какое слово должно идти дальше.
2. **Вычисление функции потерь** — предсказанное значение сравнивается с правильным следующим словом из обучающих данных. Разница между ними рассчитывается с помощью функции потерь, которая отражает точность предсказания.
3. **Обратное распространение и оптимизация** — с помощью алгоритма обратного распространения ошибки и методов градиентного спуска модель корректирует свои внутренние параметры (веса), чтобы минимизировать значение потерь. Этот цикл повторяется многократно для всего обучающего набора.

После завершения основного этапа обучения модели часто проходят *тонкую настройку* (*fine-tuning*) — дополнительное обучение на меньших, специализированных наборах данных. Такая адаптация помогает повысить точность модели в конкретных задачах или доменах.

Во время *вывода* (или *инфереенса*) обученная модель формирует текст на основе полученных входных данных. Этот процесс включает следующие этапы.

1. **Анализ контекста** — модель использует свои обученные представления языка, чтобы понять смысл входного текста и выявить ключевые элементы контекста.
2. **Генерация предсказания** — модель выбирает наиболее вероятную следующую лексему (токен) или последовательность токенов, создавая при этом связный и логичный ответ, соответствующий контексту.

Благодаря механизму внимания LLM способны справляться с многозначностью и сохранять связность на протяжении больших текстов. Модель может «вспомнить» важную информацию из предыдущих сообщений в разговоре, формируя осмысленные и точные ответы. Контекстные окна современных моделей продолжают расти — это значит, что они могут учитывать все больше входных данных одновременно.

Несмотря на свои впечатляющие возможности, LLM обладают рядом ограничений.

- **Предвзятость** — модели могут унаследовать предвзятые суждения, присутствовавшие в обучающих данных, и в результате генерировать необъективные или несправедливые ответы.

- **Галлюцинации (фактические ошибки)** — хотя модели и создают текст, похожий на человеческий, они могут выдавать неточные или абсурдные утверждения. Особенно это опасно, когда ответы звучат убедительно — например, в юридических или медицинских контекстах. Относитесь к LLM как к младшим членам команды: их работу всегда следует перепроверять.
- **Ограниченностю данных** — знания модели ограничены информацией, на которой она обучалась. Эти данные могут устаревать и не отражать текущую ситуацию.

LLM находят применение в самых разных сферах.

- **Чат-боты и виртуальные помощники** — предоставляют поддержку и имитируют диалог с человеком. Примеры: OpenAI ChatGPT-4o, Microsoft Copilot.
- **Генерация контента** — помогают создавать статьи, отчеты и даже художественные тексты. Например, Microsoft Designer: [designer.microsoft.com](https://designer.microsoft.com)
- **Машинный перевод** — используются для перевода текста между языками.
- **Помощь в программировании** — подсказывают фрагменты кода, помогают с отладкой и объясняют ошибки. Примеры: GitHub Copilot, JetBrains AI Assistant.

Теперь, когда вы познакомились с принципами работы LLM, их сильными сторонами и ограничениями, перейдем к практической части — подключению модели и началу работы с ней.

## Получение доступа к LLM

Чтобы использовать LLM в чат-сервисе, необходимо получить доступ к облачной модели. Самый простой и доступный способ интеграции в .NET-проекты — это использовать сервисы OpenAI или Azure OpenAI. На момент написания этой главы для получения доступа к Azure требуется адрес корпоративной электронной почты — например, с доменом вашей компании. Учетные записи Gmail, Outlook.com и MSN не принимаются. Поэтому в примерах этой главы мы будем работать преимущественно с OpenAI.



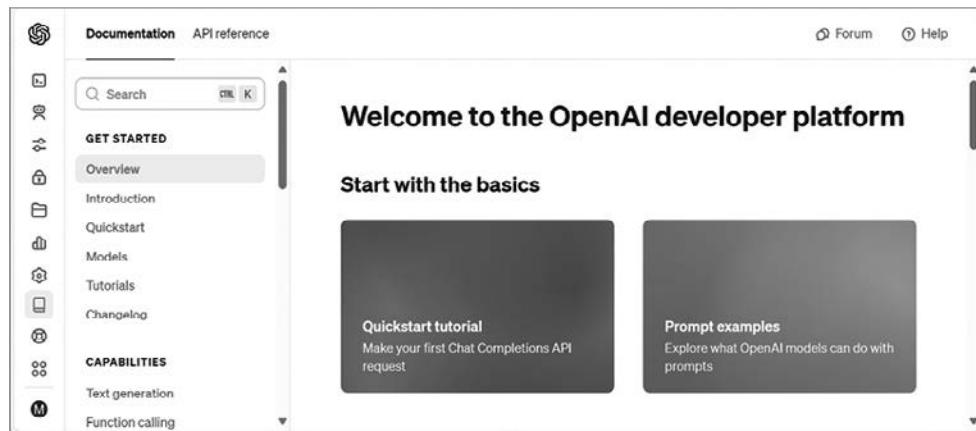
Если у вас есть корпоративная почта и вы предпочитаете платную подписку Azure, вы можете ознакомиться с инструкцией по созданию ресурса Azure OpenAI на сайте [Microsoft learn.microsoft.com/azure/ai-services/openai/how-to/create-resource](https://learn.microsoft.com/azure/ai-services/openai/how-to/create-resource).

Для подключения к облачной модели понадобятся два элемента.

- **Ключ API** — создается на портале управления сервисом LLM. Его следует хранить в секрете: любой, у кого есть этот ключ, сможет использовать ресурсы вашей учетной записи.
- **Идентификатор модели чата** — в этой главе мы будем использовать модель *gpt-3.5-turbo*, поскольку она сочетает разумную стоимость и высокую производительность. Подробности об этой модели доступны на сайте OpenAI по адресу [platform.openai.com/docs/models/gpt-3-5-turbo](https://platform.openai.com/docs/models/gpt-3-5-turbo).

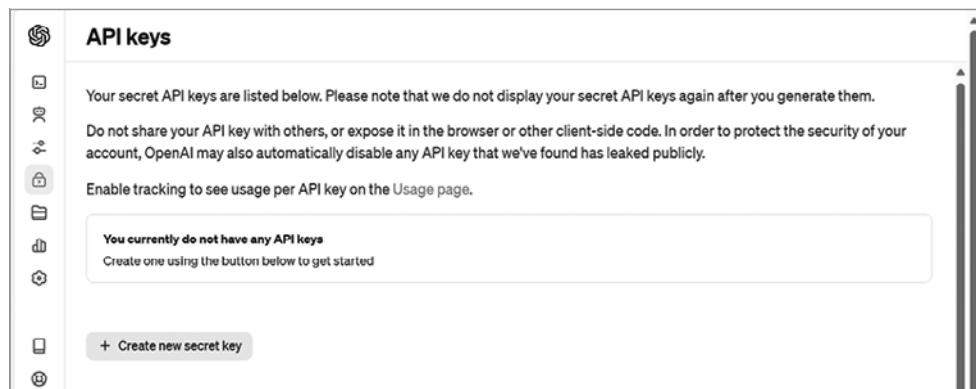
Приступим.

1. Запустите браузер и перейдите на страницу регистрации в системе OpenAI по адресу [platform.openai.com/signup](https://platform.openai.com/signup).
2. После регистрации и входа в систему откроется главная страница платформы разработчика OpenAI (рис. 9.1).



**Рис. 9.1.** Главная страница платформы разработчиков OpenAI

3. На панели навигации слева выберите раздел API keys (Ключи API) или откройте страницу [platform.openai.com/api-keys](https://platform.openai.com/api-keys) напрямую. Затем нажмите кнопку '+ Create new secret key (+ Создать новый секретный ключ)' (рис. 9.2).



**Рис. 9.2.** Страница ключей API в интерфейсе OpenAI

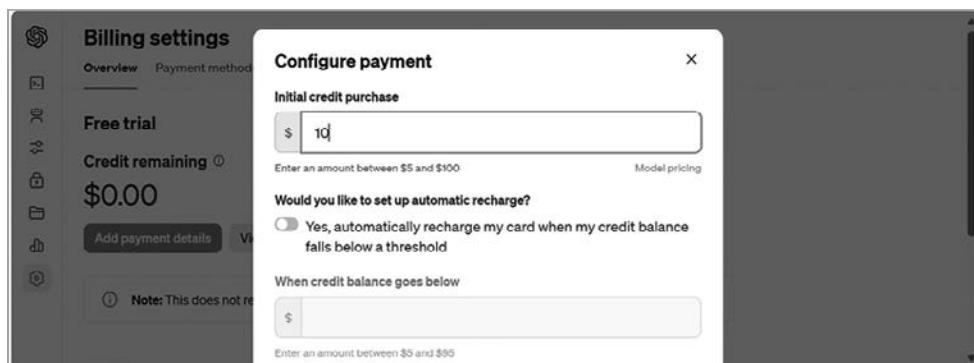
4. Введите имя ключа, например `tools-skills-net8`, и нажмите кнопку 'Create secret key' (Создать секретный ключ).

5. В открывшемся диалоговом окне **Save your key** (Сохраните ключ) скопируйте ключ в буфер обмена и сохраните его в надежном и доступном месте.



**Дельный совет:** сохраните секретный ключ в надежном и доступном месте сразу после создания. Повторно просмотреть его в личном кабинете OpenAI невозможно. В случае утери ключа придется сгенерировать новый.

6. На панели слева выберите раздел **Settings** ▶ **Billing** (Настройки ▶ Биллинг), перейдите на вкладку **Payment methods** (Способы оплаты) и добавьте способ оплаты (например, карту Visa). Либо откройте страницу [platform.openai.com/account/billing/overview](https://platform.openai.com/account/billing/overview).
7. На вкладке **Overview** (Обзор) страницы **Bill settings** (Настройки биллинга) нажмите кнопку **Add to credit balance** (Пополнить баланс) и внесите небольшую сумму, например 5 долларов (минимальный платеж) (рис. 9.3).

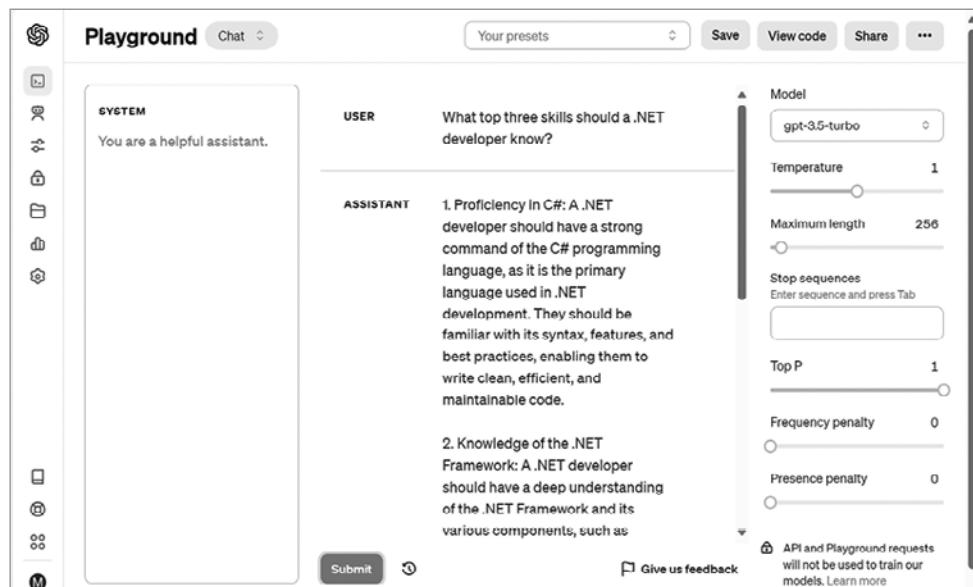


**Рис. 9.3.** Интерфейс пополнения баланса в OpenAI

8. На панели навигации слева выберите раздел **Playground** либо перейдите по адресу [platform.openai.com/playground?model=gpt-3.5-turbo](https://platform.openai.com/playground?model=gpt-3.5-turbo). В раскрывающемся списке рядом с названием страницы выберите режим **Chat** (Чат), а в качестве модели — вариант **gpt-3.5-turbo** (рис. 9.4). Можно также выбрать конкретную версию, например **gpt-3.5-turbo-0125**, а если оставить общее имя, будет использоваться самая актуальная версия модели.
9. Чтобы проверить свои знания, введите следующее пользовательское сообщение: **What top three skills should a .NET developer know?**, нажмите кнопку **Submit** (Отправить), а затем проверьте ответ (см. рис. 9.4).

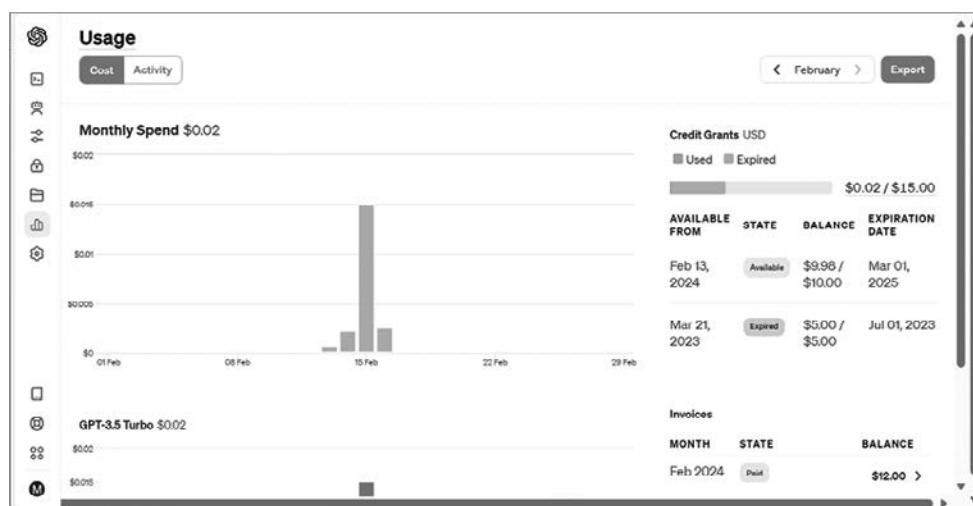


**Внимание!** Если появится сообщение о недостаточном балансе, подождите несколько минут — системе может потребоваться немного времени, чтобы обработать недавнее внесение средств. По этой причине рекомендуется сначала протестировать работу в Playground — так вы убедитесь, что все настроено правильно, прежде чем начнете обращаться к API из кода .NET.



**Рис. 9.4.** Выполнение запроса после оплаты доступа к платформе Playground

10. Затем на панели навигации выберите раздел **Usage** (Использование) либо откройте страницу [platform.openai.com](https://platform.openai.com). Как показано на странице, стоимость API-вызовов в Playground составляет всего несколько центов (рис. 9.5).
11. После проверки можно закрыть браузер.



**Рис. 9.5.** График использования вызовов API в личном кабинете

Теперь, когда все настроено и протестировано, можно переходить к следующему этапу – интеграции LLM в .NET-проект.

## Интеграция семантического ядра с моделью OpenAI

Большинство специалистов, создающих LLM, работают в среде Linux и программируют на языке Python. Учебные материалы и документация также часто ориентированы на такие инструменты. Если вы используете другой язык программирования или платформу, например .NET, это может привести к трудностям: придется самостоятельно решать вопросы совместимости и сопровождения, что усложняет работу с проектом в долгосрочной перспективе.

### Знакомство с семантическим ядром

Семантическое ядро разработано с учетом архитектуры экосистемы .NET и обеспечивает естественную интеграцию с языком C# и другими языками этой платформы. Благодаря этому .NET-разработчикам проще использовать библиотеку: ее интерфейсы и пакеты оформлены в идиоматичном стиле, привычном для среды C#.

Семантическое ядро поддерживает *асинхронную модель программирования .NET*, что особенно важно при работе с сетевыми сервисами, такими как API LLM. Это позволяет выполнять *операции ввода-вывода (I/O)* в неблокирующем режиме, сохраняя отзывчивость приложения даже при интенсивной сетевой активности.

Использование семантического ядра предоставляет гибкие возможности настройки интеграции с вашим проектом. Вы получаете полный контроль над взаимодействием с LLM – будь то настройка последовательности обработки запросов, преобразование ответов или интеграция с другими сервисами .NET. Такая архитектура хорошо масштабируется и позволяет эффективно управлять использованием модели по мере развития приложения.

Ввиду популярности платформы .NET подход с использованием семантического ядра особенно привлекателен: разработчики могут рассчитывать на поддержку сообщества, наличие подробной документации, форумов и инструментов сторонних разработчиков. Все это может оказаться крайне полезным при отладке, расширении функциональности и повышении качества приложения.

Теперь мы готовы приступить к созданию проекта .NET, который будет вызывать API OpenAI с использованием секретного ключа.

Для этого нам нужно надежно хранить параметры конфигурации. Мы воспользуемся встроенным механизмом конфигурирования в .NET, объединив JSON-файл

настроек (например, для указания имени модели) и переменные среды (для хранения секретного ключа API).

Посмотрим, как это реализуется на практике.

1. В редакторе кода создайте проект Console App (Консольное приложение) / console, присвойте ему имя ChatApp и добавьте в решение Chapter09.

2. Измените файл ChatApp.csproj. Добавьте следующие элементы:

- статический импорт System.Console для всех файлов проекта;
- ссылки на пакеты для работы с конфигурацией (JSON и переменные среды);
- подключение Semantic Kernel;
- настройку обработки предупреждений как ошибок;
- включение файла appsettings.json в выходную сборку:

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

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

    <ItemGroup>
        <!-- Подключение конфигурации из файла appsettings.json и переменных среды -->
        <PackageReference Version="8.0.1"
            Include="Microsoft.Extensions.Configuration.Binder" />
        <PackageReference Version="8.0.0"
            Include="Microsoft.Extensions.Configuration.Json" />
        <PackageReference Version="8.0.0"
            Include="Microsoft.Extensions.Configuration.EnvironmentVariables" />

        <!-- Импорт пакета для работы с семантическим ядром.-->
        <PackageReference Include="Microsoft.SemanticKernel" Version="1.13.0" />
    </ItemGroup>

    <ItemGroup>
        <Content Include="appsettings.json">
            <CopyToOutputDirectory>Always</CopyToOutputDirectory>
        </Content>
    </ItemGroup>

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

3. Выполните сборку проекта ChatApp, чтобы восстановить все зависимости и загрузить нужные пакеты.

4. Создайте в папке проекта ChatApp файл appsettings.json:

```
{
  "Settings": {
    "ModelId": "gpt-3.5-turbo-0125",
    "OpenAISecretKey": "<your-secret-key>"
  }
}
```

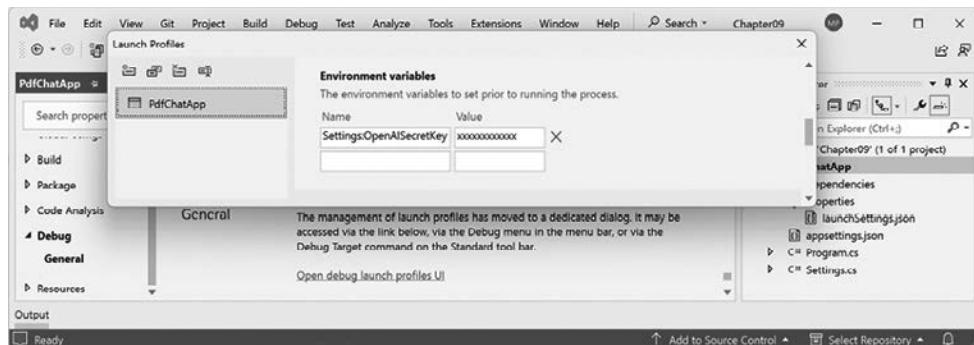
5. В той же папке создайте файл класса Settings.cs:

```
public sealed class Settings
{
    public required string ModelId { get; set; }
    public required string OpenAISecretKey { get; set; }
}
```



Класс Settings, как и автоматически сгенерированный класс Program, не принадлежит никакому пространству имен.

6. Если вы работаете в Visual Studio, выберите команду меню Project ▶ ChatApp Properties (Проект ▶ Свойства ChatApp). В разделе Debug (Отладка) нажмите кнопку Open debug launch profiles UI (Открыть UI профилей запуска отладки) и введите переменную среды — секретный ключ API (рис. 9.6).



**Рис. 9.6.** Установка переменной среды в профиле запуска проекта

7. Закройте окно Launch Profiles (Профили запуска). Убедитесь, что в проекте появилась папка Properties, содержащая файл launchSettings.json с заданной переменной среды:

```
{
  "profiles": {
    "ChatApp": {
      "commandName": "Project",
      "environmentVariables": {
        "Settings:OpenAISecretKey": "xxxxxxxxxxxx"
      }
    }
  }
}
```

```

    "environmentVariables": {
      "Settings:OpenAISecretKey": "<your-secret-key>"
    }
  }
}

```



Переменную среды можно задать не только в профиле запуска, но и напрямую в терминале. Если вы не знакомы с этим способом, подробную инструкцию можно найти по адресу [github.com/markjprice/cs12dotnet8/blob/main/docs/ch09-environment-variables.md](https://github.com/markjprice/cs12dotnet8/blob/main/docs/ch09-environment-variables.md). Кроме того, для локальной разработки можно использовать секреты пользователей (.NET User Secrets), а для развертывания в рабочей среде — Azure Key Vault или другой аналогичный инструмент для безопасного хранения конфиденциальных данных.

8. Создайте в папке проекта файл `.gitignore`, чтобы исключить `launchSettings.json` из репозитория Git:

```

# Не храните конфиденциальные учетные данные в репозитории.
launchSettings.json
**/Properties/launchSettings.json

```



**Дельный совет:** файлы вроде `launchSettings.json`, содержащие конфиденциальные данные (например, секретные ключи), не следует добавлять в публичные репозитории Git.

9. В папке проекта создайте файл класса `Program.GetSettings.cs`.
10. Удалите содержимое файла `Program.GetSettings.cs` и напишите код, который:
  - подключает `ConfigurationBuilder` и другие необходимые классы;
  - создает объект конфигурации с JSON и провайдером переменных среды;
  - связывает настройки с типизированным классом `Settings` и возвращает его:

```

// Подключение класса ConfigurationBuilder и т. д.
using Microsoft.Extensions.Configuration;

partial class Program
{
  private static Settings? GetSettings()
  {
    const string settingsFile = "appsettings.json";
    const string settingsSectionKey = nameof(Settings);

    // Создание объекта конфигурации с использованием JSON
    // и поставщиков переменных среды
    IConfigurationRoot config = new ConfigurationBuilder()
      .AddJsonFile(settingsFile)
      .AddEnvironmentVariables()
      .Build();
  }
}

```

```
// Получение настроек из конфигурации с указанием ключа
// и сильно типизированного класса
Settings? settings = config.GetRequiredSection(
    settingsSectionKey).Get<Settings>();

if (settings is null)
{
    WriteLine($"{settingsSectionKey} section not found in {settingsFile}.");
    return null;
}
else
{
    return settings;
}
}
```

11. В папке проекта создайте файл класса `Program.GetKernel.cs`.

12. Удалите содержимое этого файла и напишите в нем код, который:

- подключает `Kernel`;
- настраивает чат с OpenAI, используя имя модели и секретный ключ;
- возвращает готовый экземпляр ядра:

```
using Microsoft.SemanticKernel; // Подключение класса Kernel

partial class Program
{
    private static Kernel GetKernel(Settings settings)
    {
        IKernelBuilder kernelBuilder = Kernel.CreateBuilder();

        // Настройка чата OpenAI с моделью и секретным ключом
        kernelBuilder.AddOpenAIChatCompletion(
            settings.ModelId,
            settings.OpenAISecretKey);

        Kernel kernel = kernelBuilder.Build();

        return kernel;
    }
}
```

13. В файле `Program.cs` удалите все содержимое и:

- получите объект `Settings` через `ConfigurationHelper`;
- создайте экземпляр ядра через `KernelBuilderHelper`;
- запустите чат в цикле до тех пор, пока пользователь не нажмет клавишу X.

```
using Microsoft.SemanticKernel; // Подключение класса Kernel

Settings? settings = GetSettings();
if (settings is null)
```

```

{
    WriteLine("Settings not found or not valid. Exiting the app.");
    return; // Выход из приложения
}

Kernel kernel = GetKernel(settings);

ConsoleKey key = ConsoleKey.A;

while (key is not ConsoleKey.X)
{
    Write("Enter your question: ");
    string question = ReadLine()!;
    WriteLine(await kernel.InvokePromptAsync(question));
    WriteLine();
    WriteLine("Press X to exit or any other key to ask another question.");
    key = ReadKey(intercept: true).Key;
}

```

14. Запустите проект ChatApp без отладки.
15. Задайте вопрос *What top three skills should a .NET developer know?*. Обратите внимание на ответ модели (он может отличаться, поскольку генерация не детерминирована) — ниже приведен пример вывода:

Enter your question: What top three skills should a .NET developer know?

1. Proficiency in C# Programming Language: As a .NET developer, a strong understanding of the C# programming language is essential. Developers should be familiar with essential concepts, data types, classes, inheritance, and other key features of C#.

2. Knowledge of .NET Framework: A .NET developer should have a deep understanding of the .NET framework, including its various components such as ASP.NET, ADO.NET, and Windows Presentation Foundation (WPF). Familiarity with these components enables developers to build robust and scalable applications.

3. Understanding of Object-Oriented Programming (OOP): Object-Oriented Programming is a fundamental concept in .NET development. Developers should be well-versed in OOP principles such as inheritance, polymorphism, encapsulation, and abstraction. A good grasp of OOP enables developers to write clean, modular, and maintainable code.

Press X to exit or any other key to ask another question.

16. Нажмите любую клавишу, а затем введите другой запрос: *Please write a short biography of Mark Price*. Обратите внимание на ответ модели — ниже приведен пример вывода:

Mark Price is a former professional basketball player who was born on February 15, 1964, in Bartlesville, Oklahoma. Price played college basketball at Georgia Tech, where he was a standout point guard and earned All-American honors. He was selected in the second round of the 1986 NBA Draft by the Dallas Mavericks.

...



Самый известный в интернете Марк Прайс — баскетболист.

17. Нажмите клавишу X, чтобы выйти из приложения.

На практике чат-сервисы с ИИ часто работают с информацией, связанной с организацией, — например, с данными из реляционных БД или других источников.

Теперь разберемся, как расширить функциональность чат-сервиса, чтобы он мог использовать такую информацию и помогать нам эффективнее.

## Функции моделей OpenAI

В контексте моделей OpenAI *функции* — это вызываемые действия, которые можно встроить в процесс взаимодействия с моделью. Такие функции расширяют возможности языковой модели помимо обычной генерации текста, позволяя выполнять структурированные задачи и специфические операции по запросу пользователя.

Функции представляют собой *предопределенные задачи*, которые модель может выполнять в ответ на вызов. Благодаря этому модели становятся более полезными и гибкими — их можно адаптировать под конкретные сценарии и интеграции.

Функции могут быть самых разных типов в зависимости от задачи. Ниже приведены распространенные примеры.

- **Работа с текстом** — такие действия, как краткое изложение текста, перевод с одного языка на другой, извлечение ключевой информации из текста и пр.
- **Анализ данных** — выполнение вычислений, построение графиков, обработка наборов данных и выявление закономерностей.
- **Поддержка работы с кодом** — помощь в написании, отладке и разборе кода на разных языках программирования.
- **Взаимодействие с API** — вызовы внешних систем, баз данных и веб-сервисов для получения или отправки информации. Именно последний тип — взаимодействие с API — будет использоваться в этом проекте для расширения модели OpenAI пользовательской информацией.

Функции в моделях OpenAI, как правило, следуют структурированному процессу, состоящему из трех этапов.

- **Вызов** — функция активируется на основе определенных ключевых слов или инструкций, содержащихся в пользовательском запросе. Вызов может быть *явным* (например, «переведи это предложение») или *неявным*, если модель самостоятельно распознает задачу, которую можно сопоставить с одной из доступных функций.

- **Обработка** — после активации функция анализирует входные данные и выполняет соответствующую логику. Это может быть разбор текста, вычисления, обращение к базе данных или вызов внешнего API.
- **Генерация выходных данных** — по завершении обработки функция возвращает результат в модель. Модель может использовать эти данные для формирования финального ответа. В качестве выхода могут быть использованы обычный текст, структурированные данные или другой формат, подходящий для дальнейшего использования.

Функции значительно расширяют возможности языковых моделей, добавляя им структурированность и контекстную гибкость. Вот их основные преимущества.

- **Упрощение сложных задач** — функции позволяют быстро и точно отвечать на конкретные запросы.
- **Решение специализированных задач** — модель может выполнять действия, выходящие за рамки общего понимания текста.
- **Интеграция с внешними источниками** — с помощью функций модель получает доступ к API, базам данных и другим системам, превращаясь в универсальный инструмент для множества сценариев.

Использование функций — это мощный способ адаптировать модель OpenAI к реальным потребностям приложения. Благодаря им чат-сервис может взаимодействовать с внешними данными и выполнять действия, недоступные при обычной генерации текста.

Теперь добавим в семантическое ядро функцию, которая предоставит нашему чат-сервису дополнительную информацию — как обо мне, так и о базе данных Northwind.

## Добавление функций

Существуют два основных способа использовать функции в модели OpenAI.

1. Определить одну функцию и вызывать ее напрямую.
2. Настроить несколько функций и позволить API OpenAI выбирать нужную, основываясь на текстовых подсказках пользователя и описаниях функций.

В этом разделе мы рассмотрим оба подхода. Сначала создадим функцию, которая позволяет задавать вопросы об авторе книги, а затем добавим поддержку нескольких функций через плагин, чтобы модель могла выбирать подходящую автоматически.

Приступим.

1. В файле `ChatApp.csproj` добавьте ссылку на проект библиотеки классов контекста данных `Northwind`, созданный в главе 1:

```
<ItemGroup>
    <ProjectReference Include=
        "..\..\Chapter01\Northwind.DataContext\Northwind.DataContext.csproj" />
</ItemGroup>
```

2. Соберите проект **ChatApp**, чтобы одновременно скомпилировать зависимый проект и скопировать его сборку в локальную папку **bin**. Для этого выполните команду:

```
dotnet build
```

3. Создайте в проекте **ChatApp** файл класса **Program.ChatFunctions.cs**. Удалите все содержимое в нем и определите в частичном классе **Program** несколько статических методов: один — для получения информации об авторе, второй — для получения подмножества полей товаров по категориям из базы данных **Northwind**:

```
using Microsoft.EntityFrameworkCore; // Подключение класса Include
using Northwind.EntityModels; // Подключение класса NorthwindContext
using System.Text.Json; // Подключение класса JsonSerializer
```

```
partial class Program
{
    private static string GetAuthorBiography()
    {
        return """
Mark J Price is a former Microsoft Certified Trainer (MCT) and current Microsoft Specialist: Programming in C# and Architecting Microsoft Azure Solutions, with more than 20 years' of educational and programming experience.
```

Since 1993 Mark has passed more than 80 Microsoft programming exams and specializes in preparing others to pass them too. His students range from professionals with decades of experience to 16-year-old apprentices with none. Mark successfully guides all of them by combining educational skills with real-world experience consulting and developing systems for enterprises worldwide.

Between 2001 and 2003 Mark was employed full-time to write official courseware for Microsoft in Redmond, USA. Mark's team wrote the first training courses for C# while it was still an early alpha version. While with Microsoft he taught "train-the-trainer" classes to get other MCTs up-to-speed on C# and .NET.

Between 2016 and 2022, Mark created and delivered training courses for Optimizely's Digital Experience Platform, the best .NET CMS for Digital Marketing and E-commerce.

In 2010 Mark studied for a Post-Graduate Certificate in Education (PGCE). He taught GCSE and A-Level mathematics in two London secondary schools. Mark holds a Computer Science BSc. Hons. Degree from the University of Bristol, UK.

Mark J Price has authored the following books:

- C# 6 and .NET Core 1.0 – Modern Cross-Platform Development,  
1st Edition, 2016

```

- C# 7 and .NET Core – Modern Cross-Platform Development,
  2nd Edition, 2017
- C# 7.1 and .NET Core 2.0 – Modern Cross-Platform Development,
  3rd Edition, 2017
- C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development,
  4th Edition, 2019
- C# 9 and .NET 5 – Modern Cross-Platform Development, 5th Edition, 2020
- C# 10 and .NET 6 – Modern Cross-Platform Development, 6th Edition, 2021
- C# 11 and .NET 7 – Modern Cross-Platform Development Fundamentals,
  7th Edition, 2022
- Apps and Services with .NET 7, 1st Edition, 2022
- C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals,
  8th Edition, 2023
- Apps and Services with .NET 8, 2nd Edition, 2023
""";
}

// Эффективнее кэшировать объект, чем создавать его при каждом вызове
private static JsonSerializerOptions jsonSerializerOptions =
    new() { WriteIndented = true };

private static string GetProductsInCategory(string categoryName)
{
    using NorthwindContext db = new();

    var products = db.Products
        .Include(p => p.Category)
        .Where(p => p.Category!.CategoryName == categoryName)
        .Select(p => new
    {
        p.ProductId,
        p.ProductName,
        p.Category!.CategoryName,
        p.UnitPrice,
        p.UnitsInStock,
        p.UnitsOnOrder,
        p.Discontinued
    })
    .ToArray();

    // Преобразование массива товаров в строку JSON
    string json = JsonSerializer.Serialize(
        products, jsonSerializerOptions);

    return json;
}
}

```



Для демонстрации я выбрал подмножество полей таблицы товаров — этого достаточно, чтобы модель могла отвечать на интересные вопросы о продуктах и категориях. Связанные таблицы, такие как заказы и поставщики, были исключены, чтобы сократить объем сериализуемых данных. Поэтому модель не сможет ответить на такие вопросы, как «Кто поставляет Chai?» или «Сколько клиентов заказывают Chang?».

4. Откройте файл `Program.GetKernel.cs`, в котором реализован метод `GetKernel`. Перед возвратом объекта `kernel` добавьте вызовы для регистрации двух функций, соответствующих статическим методам, определенным выше:

```
// Создание функции как части плагина и ее добавление в ядро
kernel.ImportPluginFromFunctions(pluginName: "AuthorInformation",
[
    kernel.CreateFunctionFromMethod(
        method: GetAuthorBiography,
        functionName: nameof(GetAuthorBiography),
        description: "Gets the author's biography.")
]);

kernel.ImportPluginFromFunctions("NorthwindProducts",
[
    kernel.CreateFunctionFromMethod(
        method: GetProductsInCategory,
        functionName: nameof(GetProductsInCategory),
        description: "Get the products in a category from the Northwind database.")
]);
```

5. В файле `Program.cs` после получения компонента `kernel` определите функцию из одного из ваших методов. Закомментируйте строку с `InvokePromptAsync`, сохраните вопрос в переменную и вызовите функцию через `InvokeAsync`, передав нужные аргументы, как показано в выделенном фрагменте следующего кода:

```
Kernel kernel = GetKernel(settings);

// Стока $question будет передана в качестве аргумента
KernelFunction function = kernel.CreateFunctionFromPrompt("""
    Author biography: {{ authorInformation.getAuthorBiography }}.
    {{ $question }}
""");

KernelArguments arguments = new();

ConsoleKey key = ConsoleKey.A;

while (key is not ConsoleKey.X)
{
    Write("Enter your question: ");
    string question = ReadLine()!;

    // WriteLine(await kernel.InvokePromptAsync(question));

    arguments["question"] = question;
    // Вызов одной функции
    WriteLine(await function.InvokeAsync(kernel, arguments));

    WriteLine();
    WriteLine("Press X to exit or any other key to ask another question.");
    key = ReadKey(intercept: true).Key;
}
```

6. Запустите проект ChatApp без отладки.
7. Введите запрос *Please write a short biography of Mark Price*. Просмотрите сгенерированный ответ — он может отличаться:

Mark J Price is a highly experienced Microsoft Certified Trainer and specialist in programming in C# and architecting Microsoft Azure solutions. With over 20 years of educational and programming experience, Mark has passed more than 80 Microsoft programming exams and has helped numerous professionals achieve the same success.

During his time at Microsoft, Mark played a key role in developing the first training courses for C# and .NET. He has also authored several highly regarded books on modern cross-platform development using C# and .NET, with editions covering the latest versions of the technology.

In addition to his work in training and consulting, Mark has also taught mathematics at secondary schools in London and holds a degree in Computer Science from the University of Bristol, UK. Mark's expertise, combined with his passion for teaching and sharing knowledge, has made him a trusted resource for developers looking to enhance their skills and advance their careers in the world of technology.

8. Задайте следующий вопрос: *Write a one paragraph bio of the author*. Просмотрите сгенерированный ответ:

Mark J Price is a highly experienced Microsoft Certified Trainer and specialist in programming with over 20 years of educational and programming experience. He has authored multiple books on C# and .NET development and has successfully trained professionals of all levels, from beginners to experts. With a background in Computer Science and a Post-Graduate Certificate in Education, Mark combines his real-world experience in consulting and developing systems with his passion for teaching to effectively prepare others for Microsoft certifications and programming challenges.

9. Задайте следующий вопрос: *When did the author write a book about C# 10?* Проверьте результат, выданный программой:

The author, Mark J Price, wrote a book about C# 10 titled "C# 10 and .NET 6 - Modern Cross-Platform Development" in 2021.

10. Задайте следующий вопрос: *How many books has Mark written?* Затем просмотрите ответ, как показано в следующем выводе:

Mark J Price has written a total of ten books.

11. Задайте следующий вопрос: *Where did the author learn C#?* Изучите ответ, пример показан в выводе ниже:

Mark J Price learned C# through a combination of self-study, official Microsoft training courses, and teaching experience. He has a Computer Science degree from the University of Bristol, UK, which provided a solid foundation for his programming skills. Additionally, his time as a Microsoft Certified Trainer (MCT) and writing official courseware for Microsoft allowed him to dive deep into C# and .NET development. Mark's ongoing commitment to passing Microsoft programming exams, combined with real-world consulting and development experience, has helped him become an expert in C# programming and architecting solutions on the Microsoft stack.

12. Нажмите клавишу X, чтобы завершить сеанс работы приложения.

Теперь давайте включим автоматический выбор функций и добавим в семантическое ядро память. Благодаря этому чат-сервис сможет учитывать предыдущие вопросы и ответы в рамках текущей сессии и поддерживать связный диалог.

## Сохранение сеанса и поддержка нескольких функций

На текущем этапе наш чат-сервис не сохраняет контекст диалога. Это значит, что он не «помнит» предыдущие вопросы пользователя и каждое новое сообщение обрабатывается как отдельный, изолированный запрос. Например, представьте себе следующее взаимодействие:

```
Enter your question: Tell me a joke about programming.
Why do programmers prefer dark mode? Because the light attracts bugs!
```

```
Enter your question: Why is that funny?
Mark's ability to appreciate the lighter side of things adds a touch of charm
to his profile.
```

Без сохранения истории модель не понимает, о чем идет речь, и может начать «галлюцинировать» — то есть выдавать бессмысленные или выдуманные ответы.

Чтобы чат-сервис вел связный диалог, добавим в него память сеанса.

1. В верхней части файла `Program.cs` импортируйте пространство имен, связанное с историей чата:

```
// Подключение класса ChatHistory и других компонентов для управления памятью
using Microsoft.SemanticKernel.ChatCompletion;
```

2. В том же файле перед циклом `while` добавьте код для хранения истории сообщений и настройки автоматического выбора функций OpenAI. В цикле `while` закомментируйте вызов единственной функции и замените его выражениями, которые сохраняют ввод пользователя и ответ модели:

```
IChatCompletionService completion =
    kernel.GetService<IChatCompletionService>();
```

```

ChatHistory history = new(systemMessage: "You are an AI assistant based
on Mark J Price's knowledge, skills, and experience.");

OpenAIPromptExecutionSettings options = new()
{
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto() };

while (key is not ConsoleKey.X)
{
    Write("Enter your question: ");
    string question = ReadLine()!;

    // WriteLine(await kernel.InvokePromptAsync(question));

    // arguments["question"] = question;
    // Вызов единственной функции.
    // WriteLine(await function.InvokeAsync(kernel, arguments));

    history.AddUserMessage(question);
    ChatMessageContent answer = await
        completion.GetChatMessageContentAsync(history);
    history.AddAssistantMessage(answer.Content!);
    WriteLine(answer.Content);

    WriteLine();
    WriteLine("Press X to exit or any other key to ask another question.");
    key = ReadKey(intercept: true).Key;
}

```

3. Запустите проект **ChatApp** без отладки.
4. Введите запрос: *Tell me a joke about programming*. Посмотрите на ответ модели:

Why do programmers prefer dark mode? Because the light attracts bugs!

5. Задайте следующий вопрос: *Why is that funny?* Снова проанализируйте ответ:

This joke plays on the double meaning of "bugs" in the programming world. Bugs are errors or glitches in software code, and bugs are also attracted to light in the physical world. The humor lies in the unexpected connection between these two meanings.

6. Задайте следующий вопрос: *What is the most expensive seafood?* Затем изучите ответ, как показано в выводе:

To find out the most expensive seafood, I can retrieve information about seafood products available in the database and then identify the product with the highest price. Let me fetch the necessary data for you. The most expensive seafood product is "Carnarvon Tigers" with a unit price of \$62.5. It is the top-priced seafood item in the database. If you have any more questions or need further information, feel free to ask!

7. Задайте следующий вопрос: *What products are in the seafood category?* Посмотрите на ответ модели:

Here are the products in the Seafood category:

1. Ikura
2. Konbu
3. Carnarvon Tigers
4. Nord-Ost Matjeshering
5. Inlagd Sill
6. Gravad lax
7. Boston Crab Meat
8. Jack's New England Clam Chowder
9. Rogede sild
10. Spegesild
11. Escargots de Bourgogne
12. Röd Kaviar

8. Напишите следующий запрос: *Please include the prices.* Посмотрите на ответ модели:

Here are the products in the Seafood category along with their prices:

1. Ikura - \$31
2. Konbu - \$6
3. Carnarvon Tigers - \$62.5
4. Nord-Ost Matjeshering - \$25.89
5. Inlagd Sill - \$19
6. Gravad lax - \$26
7. Boston Crab Meat - \$18.4
8. Jack's New England Clam Chowder - \$9.65
9. Rogede sild - \$9.5
10. Spegesild - \$12
11. Escargots de Bourgogne - \$13.25
12. Röd Kaviar - \$15

9. Нажмите клавишу X, чтобы завершить работу приложения.

Иногда модель OpenAI не отвечает сразу. Чтобы взаимодействие стало более плавным и отзывчивым, можно перейти на асинхронную потоковую передачу — об этом пойдет речь в следующем разделе.

## Потоковая передача ответов

Теперь реализуем асинхронную потоковую передачу.

1. В файле `Program.cs` перед циклом `while` добавьте код для подключения класса, который будет использоваться при работе с потоковым выводом. Затем

в цикле `while` добавьте выражения, реализующие построчную передачу ответа от модели в реальном времени:

```
// Подключение вспомогательного класса для асинхронного потокового вывода
StringBuilder builder = new();

while (key is not ConsoleKey.X)
{
    Write("Enter your question: ");
    string question = ReadLine()!;

    // WriteLine(await kernel.InvokePromptAsync(question));

    // arguments["question"] = question;
    // Асинхронный вызов функции
    // WriteLine(await function.InvokeAsync(kernel, arguments));

    history.AddUserMessage(question);

    // ChatMessageContent answer = await
    // completion.GetChatMessageContentAsync(history);

    builder.Clear();
    await foreach (StreamingChatMessageContent message
        in completion.GetStreamingChatMessageContentsAsync(history))
    {
        Write(message.Content);
        builder.Append(message.Content);
    }
    history.AddAssistantMessage(builder.ToString());

    WriteLine();
    WriteLine("Press X to exit or any other key to ask another question.");
    key = ReadKey(intercept: true).Key;
}
```

2. Запустите проект `ChatApp` без отладки.
3. Обратите внимание, что ответы от сервиса OpenAI теперь передаются асинхронно.

## Добавление журналирования и обеспечение отказоустойчивости

Теперь улучшим надежность и наблюдаемость чат-сервиса, добавив журналирование и отказоустойчивость при работе с моделью.

1. В файле `ChatApp.csproj` проекта добавьте ссылки на пакеты, необходимые для реализации журналирования и обработки ошибок при HTTP-запросах:

```
<!-- Добавление поддержки журналирования в консоль и отказоустойчивости
для HTTP-запросов -->
<PackageReference Version="8.0.0" Include="Microsoft.Extensions.Http" />
```

```
<PackageReference Version="8.2.0" Include="Microsoft.Extensions.Http.Resilience" />
<PackageReference Version="8.0.0" Include="Microsoft.Extensions.Logging.Console" />
```

2. В верхней части файла `Program.GetKernel.cs` добавьте директивы `using` для пространства имен, отвечающего за журналирование:

```
using Microsoft.Extensions.DependencyInjection; // Подключение метода AddLogging
using Microsoft.Extensions.Logging; // Подключение перечисления LogLevel
```

3. Перед созданием ядра (`Kernel`) в методе `GetKernel` добавьте код для настройки журналирования и обработки сбоев:

```
// Добавление конфигурации журналирования
// Добавление политики отказоустойчивости
kernelBuilder.Services.AddLogging(c =>
    c.AddConsole().SetMinimumLevel(LogLevel.Trace));
kernelBuilder.Services.ConfigureHttpClientDefaults(c =>
    c.AddStandardResilienceHandler());
```

```
Kernel kernel = kernelBuilder.Build();
```

4. Запустите проект `ChatApp` без отладки. Теперь при запуске вы увидите дополнительные сообщения из журнала:

```
trce: Program[0]
    Created KernelFunction 'GetAuthorBiography' for 'GetAuthorBiography'
trce: Microsoft.SemanticKernel.KernelPromptTemplate[0]
    Extracting blocks from template: Author biography: {{authorInformation.getAuthorBiography }}.
    {{ $question }}
trce: Program[0]
    Created KernelFunction 'GetProductsInCategory' for
'GetProductsInCategory'
Enter your question:
```

5. Введите следующий запрос: *Please generate the source code HTML for a table of the products in the condiments category. Please include the price and number in stock.*
6. Просмотрите результат, как показано в следующем выводе (фрагмент):

```
I have retrieved the products in the Condiments category with their prices and
stock information. Let me generate the HTML source code for a table displaying
this information.
Here is the HTML source code for a table displaying the products in the
Condiments category along with their prices and number in stock:
```html


| Product Name  | Unit Price |
|---------------|------------|
| Mustard       | \$1.20     |
| Ketchup       | \$0.80     |
| Sriracha      | \$2.50     |
| Honey Mustard | \$1.50     |


```

```

<th>Units In Stock</th>
</tr>
</thead>
<tbody>
<tr>
  <td>Aniseed Syrup</td>
  <td>$10.00</td>
  <td>13</td>
</tr>
...
<tr>
  <td>Original Frankfurter grüne Soße</td>
  <td>$13.00</td>
  <td>32</td>
</tr>
</tbody>
</table>```
This table includes the product name, unit price, and units in stock for each
product in the Condiments category. Feel free to modify the code as needed.

```

7. Нажмите клавишу X, чтобы завершить работу приложения.

Теперь мы готовы перейти к следующему этапу — подключению локальной языковой модели вместо облачной версии OpenAI.

## Запуск локальной LLM-модели

В этом разделе мы рассмотрим, как можно загрузить и запустить большую языковую модель (LLM) на локальном компьютере.

## Платформа Hugging Face

*Hugging Face* — это популярная платформа с открытым исходным кодом, предназначенная для создания, распространения и использования современных моделей в области обработки естественного языка (NLP). Платформа предлагает обширный набор инструментов, библиотек и ресурсов, которые упрощают доступ к передовым достижениям в области машинного обучения. Ниже перечислены ключевые компоненты, которые *Hugging Face* предоставляет сообществу разработчиков LLM.

- **Библиотека Transformers** — предоставляет широкий выбор предварительно обученных моделей для задач генерации текста, перевода, краткого изложения, ответов на вопросы и других направлений NLP.
- **Библиотека Datasets** — содержит обширную коллекцию готовых к использованию наборов данных, охватывающих самые разные задачи и предметные области.
- **Model Hub (Хаб моделей)** — веб-платформа для публикации, поиска и загрузки моделей. Пользователи могут делиться собственными моделями, используя

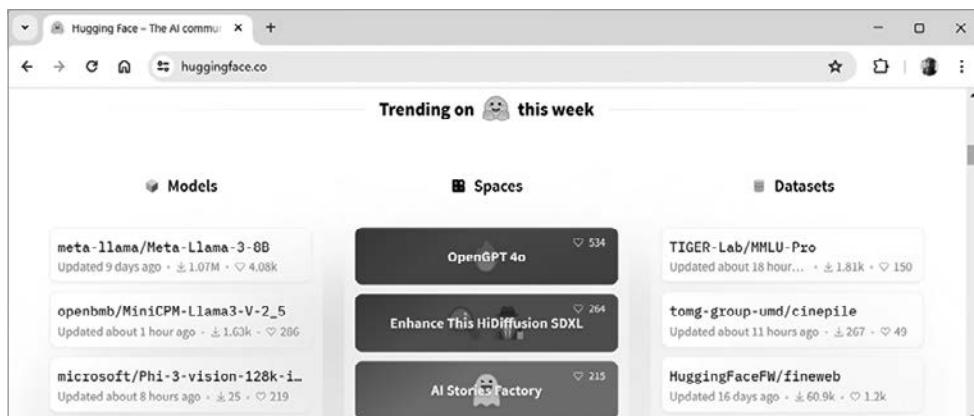
вать модели других разработчиков и сравнивать их эффективность с помощью встроенных инструментов оценки.

- **Библиотека Tokenizers** — предоставляет быстрые и гибкие методы токенизации текста, необходимые для подготовки входных данных к использованию в трансформерах.



Одна из новых возможностей, которая появится в .NET 9 (релиз ожидается в ноябре 2025 года), — это стандартный тип для работы с токенизаторами. Подробности можно найти по адресу [github.com/dotnet/core/blob/main/release-notes/9.0/preview/preview4/libraries.md#tokenizer-library-enhancements](https://github.com/dotnet/core/blob/main/release-notes/9.0/preview/preview4/libraries.md#tokenizer-library-enhancements).

На главной странице сайта Hugging Face ([huggingface.co](https://huggingface.co)) можно следить за текущими тенденциями в сообществе. Например, модель Llama 3 компании Meta с объемом 8 млрд параметров на момент написания была скачана более 1 млн раз (рис. 9.7).



**Рис. 9.7.** Главная страница сайта Hugging Face



Модель Llama 3 компании Meta популярна по ряду причин. Объем в 8 миллиардов параметров — удачный компромисс между производительностью и ресурсами: модель достаточно мощная, чтобы улавливать смысловые нюансы, но при этом может работать на обычных системах. Скорее всего, при ее обучении использовались современные методы и разнообразные наборы данных. Кроме того, популярность порождает экосистему — вокруг модели появляется документация, обсуждения на форумах и активное сообщество.

Теперь познакомимся с инструментами, которые помогут загружать и запускать модели Hugging Face локально — быстро, просто и удобно.

## Платформа Ollama

Ollama — это инструмент для запуска больших языковых моделей (LLM) на локальных устройствах. Он разработан в целях упрощения работы со сложными моделями вроде LLaMA (Large Language Model Meta AI) без подключения к облаку — прямо на вашем компьютере.

Локальное выполнение моделей с помощью Ollama обеспечивает высокий уровень конфиденциальности и безопасности, поскольку все данные остаются на устройстве пользователя. Это особенно важно при работе с чувствительной или служебной информацией — в отличие от облачных сервисов, здесь вы сохраняете полный контроль над своими данными.

Разработчики и исследователи могут использовать Ollama для проведения экспериментов и тонкой настройки LLM, не прибегая к облачным ресурсам. Локальный запуск моделей может оказаться более экономичным в долгосрочной перспективе, поскольку устраняет необходимость регулярной подписки на облачные сервисы. Однако следует учитывать, что локальные установки требуют регулярного обслуживания и обновлений, что может быть сложнее, чем работа с управляемыми облачными решениями.

Для эффективного запуска LLM на локальном компьютере требуется мощное оборудование. Как правило, это высокопроизводительный процессор, значительный объем оперативной памяти, а также видеокарта (GPU) для обработки ресурсоемких операций. Одно из ключевых преимуществ локального выполнения — низкая задержка при взаимодействии, что особенно важно для приложений с требованием немедленного отклика. В то же время не каждый пользователь располагает техникой, способной полноценно запускать такие модели.

Сотрудники компании Ollama стремятся максимально упростить установку и настройку. Платформа предлагает понятные инструменты и подробную документацию, благодаря которым можно начать работу без лишних сложностей.

Теперь перейдем к установке Ollama на локальное устройство.

1. Запустите браузер и перейдите на сайт [ollama.com](https://ollama.com).
2. На главной странице сайта Ollama нажмите кнопку **Download** (Загрузить).
3. Выберите операционную систему и следуйте инструкциям для установки Ollama на свой компьютер.

## Модели Ollama

Модели часто выпускаются в нескольких вариантах, как правило, отличающихся размером и возможностями. Например:

- **11lama** и **11lama:70b** различаются по размеру — у второй модели значительно больше параметров;



Небольшие модели, такие как llama, хорошо подходят для простых задач — генерации текста, ответов на короткие вопросы, краткого изложения. Они особенно эффективны в приложениях, где важна низкая задержка, например в чат-ботах и голосовых помощниках. Более крупные модели, такие как llama:70b, способны лучше понимать контекст, проводить рассуждения и генерировать длинные связные тексты. Однако они требуют значительно больше ресурсов — как при обработке запросов, так и при обучении. Кроме того, иногда компактная модель, прошедшая тонкую настройку, может превзойти универсальную крупную модель в конкретной предметной области.

- `mistral:instruct` — модель, ориентированная на выполнение инструкций;
- `mistral:text` — базовая модель без тонкой настройки для диалогов; наилучшим образом подходит для простого дополнения текста.

Чтобы быстро загрузить и запустить модель в интерактивном режиме, выполните следующую команду:

```
ollama run <модель>
```

Наиболее популярные модели, поддерживаемые Ollama, представлены в табл. 9.1.

**Таблица 9.1.** Распространенные модели Ollama

Модель	Параметры	Размер	Описание
llama3:70b	70 млрд параметров	40 Гбайт	Самая мощная на текущий момент LLM. Инструментированная версия Llama 3 оптимизирована для чатов, превосходит в производительности многие модели с открытым кодом
llama3	8 млрд параметров	4,7 Гбайт	Упрощенная версия семейства Llama 3
codellama	7 млрд параметров	3,8 Гбайт	Специализирована на генерации и анализе кода по текстовым подсказкам
llama2-uncensored	7 млрд параметров	3,8 Гбайт	Вариант Llama 2 без цензуры. Создан Джорджем Сунгом и Джаррадом Хоупом по методике Эрика Хартфорда (см. erichartford.com/uncensored-models)
phi3	3,8 млрд параметров	2,3 Гбайт	Phi-3 — это семейство легких открытых моделей компании Microsoft, включающее версии с 3 млрд (Mini) и 14 млрд параметров (Medium). На конференции Microsoft Build 2024 была представлена новая версия phi-3-silica, оптимизированная для работы на устройствах Copilot+ PC, оснащенных нейронным процессором (NPU) с поддержкой более 40 TOPS (триллионов операций в секунду)

*Продолжение ↗*

**Таблица 9.1** (окончание)

Модель	Параметры	Размер	Описание
mistral	7 млрд параметров	4,1 Гбайт	Модель с 7,3 млрд параметров, распространяется по лицензии Apache. Доступна в виде как инструктивной, так и базовой версии
gemma:7b	7 млрд параметров	4,8 Гбайт	Легкая современная модель (семейство) компании Google DeepMind
gemma:2b	2 млрд параметров	1,4 Гбайт	Самая компактная модель в семействе Gemma, подходит для запуска на мобильных устройствах



Uncensored (без цензуры) в контексте LLM означает, что модель работает без предопределенных фильтров и ограничений на генерируемый контент. Это позволяет получить более широкий спектр ответов, включая нестандартные, провокационные и даже потенциально спорные формулировки. Такие модели могут быть полезны в исследовательской среде, при разработке, а также, например, в творческих проектах — скажем, для написания шуток в стиле стендап-комика. Однако они несут в себе высокие риски, связанные с возможной генерацией вредного, оскорбительного или неэтичного контента, особенно при использовании в публичных или коммерческих продуктах.

В этом проекте мы будем использовать модель Llama 3, так как она сочетает универсальность и совместимость — с высокой вероятностью она сможет работать на вашем компьютере. Ознакомиться с лицензионным соглашением можно по адресу [llama.meta.com/llama3/license/](https://llama.meta.com/llama3/license/).

## Интерфейс командной строки Ollama

Ollama CLI предоставляет набор команд для управления языковыми моделями и их запуска на локальном компьютере. Хотя конкретный перечень команд может отличаться в зависимости от версии Ollama, ниже представлен общий обзор наиболее распространенных команд и их назначений.

Давайте разберемся, какие базовые действия можно выполнить через интерфейс командной строки Ollama, работая с моделью Llama 3.

- Откройте терминал и введите команду для проверки установленной версии Ollama:

```
ollama --version
```

- Посмотрите ответ — он содержит текущую версию Ollama:

```
ollama version is 0.1.38
```

- Извлеките (скачайте) модель Llama 3:

```
ollama pull llama3
```

4. Посмотрите на результат загрузки:

```
pulling manifest
pulling 6a0746a1ec1a... 100% [██████████]
██████████| 4.7 GB
pulling 4fa551d4f938... 100% [██████████]
██████████| 12 KB
pulling 8ab4849b038c... 100% [██████████]
██████████| 254 B
pulling 577073ffcc6c... 100% [██████████]
██████████| 110 B
pulling 3f8eb4da87fa... 100% [██████████]
██████████| 485 B
verifying sha256 digest
writing manifest
removing any unused layers
success
```



Удалить модель можно с помощью такой команды: `ollama rm llama3`.

5. Выведите список доступных локальных моделей:

```
ollama list
```

6. Оцените результат:

NAME	ID	SIZE	MODIFIED
llama3:latest	365c0bd3c000	4.7 GB	21 minutes ago

7. Запустите модель Llama 3 (модель будет загружена, если вы не сделали это ранее):

```
ollama run llama3
```

8. Убедитесь, что началась интерактивная сессия, как показано ниже:

```
>>> Send a message (/? for help)
```

9. Введите запрос, например:

```
What is .NET?
```

10. Чтобы завершить сессию, введите команду:

```
/bye
```



CLI Ollama также поддерживает команды для копирования моделей и создания собственных вариантов. Полный список доступных команд и их параметров вы найдете в документации по адресу [github.com/ollama/ollama#cli-reference](https://github.com/ollama/ollama#cli-reference).

На текущий момент официальные клиентские библиотеки Ollama доступны только для языков Python и JavaScript. Для .NET пока не существует официальной поддержки. Однако одна из сторонних компаний разработала библиотеку для платформы .NET, которую мы рассмотрим в следующем разделе.

## Пакет OllamaSharp для .NET

OllamaSharp — это библиотека клиента для .NET, позволяющая удобно работать с API Ollama. Пакет поддерживает все ключевые возможности Ollama: чат-интерфейс, встраивания (embeddings), перечисление моделей, а также загрузку и создание новых моделей — все это становится доступным в привычной среде .NET.



**В копилку знаний:** репозиторий OllamaSharp доступен на сайте GitHub по адресу [github.com/awaescher/OllamaSharp](https://github.com/awaescher/OllamaSharp).

Давайте попробуем этот пакет «на вкус».

1. В редакторе кода создайте проект Console App (Консольное приложение) / `console`, присвойте ему имя `OllamaApp` и добавьте в решение `Chapter09`.
2. Откройте файл `OllamaApp.csproj` и добавьте ссылки на пакеты `Spectre.Console` и `OllamaSharp`, а также статически и глобально импортируйте тип `System.Console`:

```
<ItemGroup>
  <PackageReference Include="Spectre.Console" Version="0.49.1" />
  <PackageReference Include="OllamaSharp" Version="1.1.9" />
</ItemGroup>

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



Проверить актуальные версии этих пакетов можно на сайте NuGet по адресам [www.nuget.org/packages/Spectre.Console](https://www.nuget.org/packages/Spectre.Console) и [www.nuget.org/packages/OllamaSharp](https://www.nuget.org/packages/OllamaSharp).

3. Откройте файл `Program.cs`, удалите все его содержимое и добавьте следующее:
  - импортируйте пространства имен, необходимые для работы с `OllamaSharp` и `Spectre Console`;
  - создайте клиента Ollama, настроенного на отправку запросов к локальному экземпляру Ollama по адресу `localhost` и стандартному порту `11434`;
  - выведите на экран таблицу с помощью `Spectre.Console`, чтобы отобразить список доступных локальных моделей;

- настройте клиента на использование последней версии модели Llama 3;
- запросите ввод от пользователя, отправьте запрос модели и выведите полученный ответ;
- измерьте время выполнения запроса, воспользовавшись `Stopwatch`.

```
using OllamaSharp; // Подключение класса OllamaApiClient
using OllamaSharp.Models; // Подключение модели данных Model
using Spectre.Console; // Подключение компонентов Spectre.Console:
                      // Table, AnsiConsole
using System.Diagnostics; // Подключение Stopwatch для замера времени

// Стандартный порт API Ollama – 11434
string port = "11434";
Uri uri = new($"http://localhost:{port}");

OllamaApiClient ollama = new(uri);

Table table = new();
table.AddColumn("Name");
table.AddColumn("Size");

// Получение списка доступных моделей
IEnumerable<Model> models = await ollama.ListLocalModels();
foreach (Model model in models)
{
    table.AddRow(model.Name, model.Size.ToString("N0"));
}

AnsiConsole.Write(table);

string modelName = "llama3:latest";

WriteLine();
WriteLine($"Selected model: {modelName}");
ollama.SelectedModel = modelName;

Write("Please enter your prompt: ");
string? prompt = ReadLine();
if (string.IsNullOrWhiteSpace(prompt))
{
    WriteLine("Prompt is required. Exiting the app.");
    return;
}

Stopwatch timer = Stopwatch.StartNew();

ConversationContext context = new([]);
context = await ollama.StreamCompletion(
    prompt, context, stream => Write(stream.Response));

timer.Stop();

WriteLine();
WriteLine();
WriteLine($"Elapsed time: {timer.ElapsedMilliseconds:N0} ms");
```

4. Запустите проект OllamaApp без отладки.
5. Введите запрос, например *What is .NET?*, и изучите результат (см. ниже):

```
Please enter your prompt: What is .NET?  
.NET (pronounced "dot net") is a software framework developed by Microsoft  
that allows developers to build robust, scalable, and efficient applications  
for Windows, web, mobile, and other platforms.  
...  
Elapsed time: 170,501 ms
```

Теперь рассмотрим альтернативу Ollama — еще один способ запускать и тестировать локальные LLM.

## Приложение LM Studio

Рассмотрим основные возможности LM Studio.

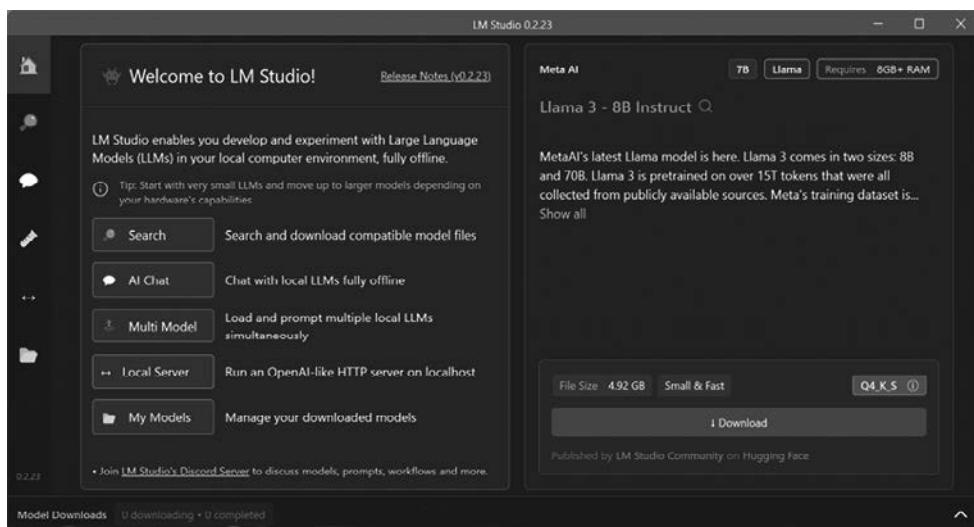
- Запуск больших языковых моделей полностью в автономном режиме, например на ноутбуке.
- Работа с моделями через встроенный чат-интерфейс или через локальный сервер, совместимый с OpenAI API.
- Загрузка любых совместимых моделей из Hugging Face напрямую через интерфейс приложения.
- Ознакомление с новыми и популярными моделями на главной странице приложения.



**Минимальные требования:** Mac на базе M1 или ПК под управлением операционной системы Windows с процессором с поддержкой технологии AVX2. Поддержка Linux пока тестируется.

Как начать работу с LM Studio?

1. Откройте браузер и перейдите на сайт lmstudio.ai.
2. Нажмите кнопку **Download** (Загрузить), чтобы загрузить LM Studio для своей операционной системы.
3. После загрузки установите и запустите классическое (настольное) приложение.
4. На главной странице вы, скорее всего, увидите модель Meta AI Llama 3 с 8 миллиардами параметров, поскольку она входит в число самых популярных. Если модель не отображается, воспользуйтесь полем **Search** (Поиск), чтобы найти ее вручную. Выберите модель и нажмите кнопку **Download** (Загрузить) (рис. 9.8).



**Рис. 9.8.** Главная страница LM Studio с моделью Llama 3 8B компании Meta AI

5. Дождитесь завершения загрузки. Статус можно отслеживать, развернув область состояния в нижней части окна (рис. 9.9).



**Рис. 9.9.** Процесс загрузки модели Llama 3 8B в LM Studio

6. Перейдите на вкладку AI Chat (ИИ-чат) и выберите загруженную модель.
7. В поле USER (Пользователь) введите запрос, например: *What is .NET?* Изучите ответ — он должен быть достаточно точным и связным.
8. Введите другой запрос, например: *Who says, “Why is Gamora?”, and why?*
9. Обратите внимание: ответ может содержать галлюцинации, то есть неточные или вымышленные факты — это распространенная особенность некоторых локальных моделей:

The phrase "Why is Gamora?" is famously uttered by Peter Quill (Star-Lord) in the Marvel movie "Guardians of the Galaxy" (2014). In the film, Star-Lord is trying to understand the motivations behind Gamora's actions. He asks her "Why is Gamora?" out of confusion and curiosity about her past as an assassin working for Thanos. Gamora responds with a cryptic message: "I was raised by Thanos. I'm the last one left."

10. Если задать тот же вопрос в ChatGPT-4o, скорее всего, ответ будет более точным и детализированным — за счет более свежих данных и более мощной архитектуры модели:

The line "Why is Gamora?" is said by Drax, a character from the movie Avengers: Infinity War. This moment occurs during a scene where the Avengers confront the Guardians of the Galaxy for the first time.

Если вы сравните работу локальной модели с облачной, то, скорее всего, заметите, что локальные LLM пока заметно уступают по производительности. Они работают медленнее, и скорость напрямую зависит от возможностей вашего оборудования. Однако по мере того, как технологии ИИ начинают переходить на уровень пользовательских устройств, ситуация меняется. С выходом устройств Copilot+ PC, в числе которых Surface Pro 11 и Surface Laptop 7 с новыми чипами Snapdragon X Elite / Snapdragon X Plus, можно ожидать значительного повышения скорости выполнения моделей на локальном компьютере. Я планирую протестировать, насколько быстрее эти устройства справляются с локальными моделями LLM. Но так как заказанные устройства я получу уже после завершения работы над этой книгой, результатами тестов я не успею поделиться в текущем издании.



**В копилку знаний:** подробнее о ПК Copilot+ можно узнать по адресу [blogs.microsoft.com/blog/2024/05/20/introducing-copilot-pcs/](https://blogs.microsoft.com/blog/2024/05/20/introducing-copilot-pcs/).

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 9.1. Дополнительные онлайн-материалы

- Вводный блог о Semantic Kernel: [devblogs.microsoft.com/semantic-kernel/hello-world/](https://devblogs.microsoft.com/semantic-kernel/hello-world/).
- Блог-пост *Semantic Kernel: Embeddings and Memories. Explore GitHub Repos with Chat UI*: [devblogs.microsoft.com/semantic-kernel/semantic-kernel-embeddings-and-memories-explore-github-repos-with-chat-ui/](https://devblogs.microsoft.com/semantic-kernel/semantic-kernel-embeddings-and-memories-explore-github-repos-with-chat-ui/).
- Блог *Making AI-powered .NET apps more consistent and intelligent with Redis*: [devblogs.microsoft.com/semantic-kernel/making-ai-powered-net-apps-more-consistent-and-intelligent-with-redis/](https://devblogs.microsoft.com/semantic-kernel/making-ai-powered-net-apps-more-consistent-and-intelligent-with-redis/).
- Блог *Building AI-powered Microsoft Copilot with SignalR and other open-source tools*: [devblogs.microsoft.com/dotnet/building-ai-powered-bing-chat-with-signalr-and-other-open-source-tools/](https://devblogs.microsoft.com/dotnet/building-ai-powered-bing-chat-with-signalr-and-other-open-source-tools/).
- Блог *Build Intelligent Applications using ChatGPT & Azure Cosmos DB*: [devblogs.microsoft.com/cosmosdb/chatgpt-azure-cosmos-db/](https://devblogs.microsoft.com/cosmosdb/chatgpt-azure-cosmos-db/).

- Блог *How to use Hugging Face models with Semantic Kernel* — о применении моделей Hugging Face через API Semantic Kernel: [devblogs.microsoft.com/semantic-kernel/how-to-use-hugging-face-models-with-semantic-kernel/](https://devblogs.microsoft.com/semantic-kernel/how-to-use-hugging-face-models-with-semantic-kernel/).

## Упражнение 9.2. Практические задания

Несколько идей для сервисов на базе LLM, которые вы могли бы реализовать.

1. **Научный ассистент с ИИ**, обученный на академических публикациях. Такой сервис сможет давать обоснованные научные ответы с точными ссылками на источники.
2. **Чат-сервис по программированию**, который считывает PDF-книги по разработке из локальной папки и умеет общаться в интерактивном режиме. Если вы затрудняетесь с решением на определенной странице книги, можно сообщить об этом чат-сервису, передать сообщение об ошибке — и получить осмысленное предложение, как решить проблему.
3. **Рекрутер по .NET** — чат, помогающий разработчику подготовиться к собеседованию, моделируя реальные вопросы и предлагая обратную связь.
4. **Проектный чат-бот для новых сотрудников**, обученный спецификации проекта, документации и кодовой базе. Он может познакомить новичка с проектом и ответить на вопросы — вместо того чтобы отвлекать тимлида в Slack.
5. **Генератор постов в блоге на тему .NET**, предлагающий идеи для публикаций. Вы выбираете тему, а он помогает написать текст, оформить его — и затем опубликовать в Habr, Дзене, Threads и Mastodon.

## Упражнение 9.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Что такое семантическое ядро?
2. Зачем используется метод `CreateFunctionFromMethod?`
3. Почему по умолчанию каждый запрос в LLM через Semantic Kernel обрабатывается отдельно и как добавить сессионную память для учета предыдущих сообщений?
4. Как включить потоковую передачу ответов от LLM, если по умолчанию необходимо ждать полный результат?
5. Что такое Hugging Face?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 9.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-9---building-a-custom-llm-based-chat-service](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-9---building-a-custom-llm-based-chat-service).

## Резюме

В этой главе вы узнали:

- как работают LLM и как получить к ним доступ;
- о концепциях, связанных с семантическим ядром;
- как расширить модель LLM с помощью функций;
- как добавить память сеанса и потоковую передачу ответов;
- о платформе Hugging Face и запуске локальных моделей.

В следующей главе вы узнаете о контейнерах для внедрения зависимостей (Dependency Injection, DI), которые помогают автоматизировать подключение зависимостей и управление временем жизни сервисов, обеспечивая более эффективную и устойчивую архитектуру приложения.

# 10

## Внедрение зависимостей, контейнеры и время жизни сервисов

Эта глава посвящена внедрению зависимостей (Dependency Injection, DI) — одному из ключевых принципов современной разработки ПО. DI способствует разработке тестируемого и поддерживаемого кода с низкой связанностью компонентов, разделяя ответственность за создание зависимостей и их использование.

DI-контейнеры автоматизируют процесс внедрения, избавляя разработчика от необходимости вручную создавать экземпляры и управлять их жизненным циклом. Конфигурируя зависимости в контейнере, приложение может разрешать и внедрять их динамически во время выполнения, обеспечивая корректные экземпляры для каждого компонента. Такой подход упрощает подключение зависимостей, повышает модульность и гибкость архитектуры, способствует многократному использованию и единообразию.

Один из ключевых аспектов DI — время жизни сервисов — определяет, как долго контейнер хранит экземпляр сервиса и в каких контекстах он используется повторно.

Независимо от того, разрабатываете ли вы простое консольное приложение или крупный веб-сервис, знание принципов DI, работы с контейнерами и понимание времени жизни сервисов помогают писать более устойчивый и масштабируемый код.

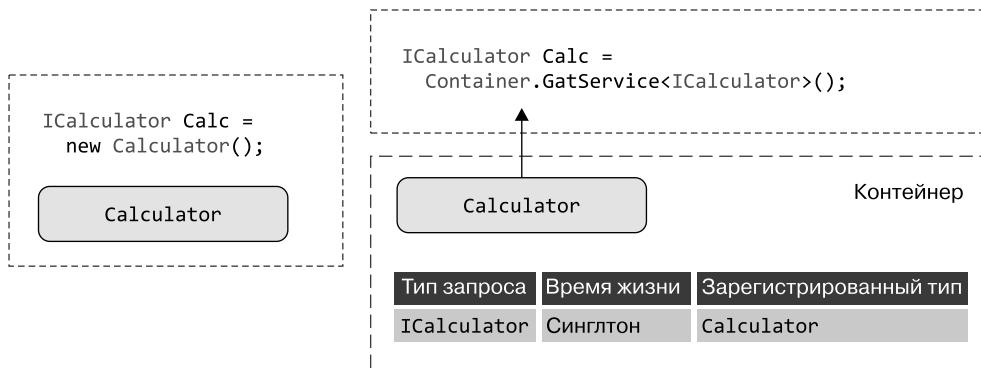
### Знакомство с внедрением зависимостей

Внедрение зависимостей (*Dependency Injection, DI*) — это паттерн проектирования, применяемый для реализации инверсии управления (*Inversion of Control, IoC*) при разрешении зависимостей внутри приложения. В традиционном подходе контроль за выполнением остается у приложения, которое само вызывает методы библиотек или фреймворков. Инверсия управления меняет эту схему: теперь именно фреймворк управляет выполнением, вызывая ваш код в определенные моменты — например, при получении HTTP-запроса.

Такой подход активно используется в ASP.NET Core, где фреймворк обрабатывает входящие запросы, а ваш код срабатывает в ответ на события, такие как GET или POST.

Основная идея DI — отделить создание зависимостей от их использования. Вместо того чтобы объект сам создавал свои зависимости, они внедряются извне, обычно во время выполнения — через контейнер зависимостей. Благодаря этому код становится более модульным, его проще тестировать и легче сопровождать и расширять.

Пример: на рис. 10.1, слева показано создание экземпляра калькулятора вручную, а справа — получение зарегистрированной реализации из контейнера сервисов.



**Рис. 10.1.** Сравнение ручного создания объектов и внедрения зависимостей через контейнер

Преимущества использования контейнера: можно заменить реализацию, не изменяя код, который ее использует; можно изменить область видимости (время жизни) объекта; контейнер отвечает за создание и управление зависимостями, освобождая вас от рутинной и потенциально ошибочной логики.

## Для чего внедряются зависимости

Перечислю основные причины, по которым внедрение зависимостей (DI) широко применяется в современных проектах.

- **Снижение связанности.** DI помогает разделить компоненты и их зависимости, делая систему более модульной. Когда код разделен на модули, вносить изменения можно локально, не затрагивая остальную часть системы. Это упрощает обновление, исправление ошибок и добавление новых функций. Четкая модульная структура облегчает понимание и навигацию по проекту, особенно для новых разработчиков, и снижает кривую вхождения в команду.
- **Повышение тестируемости.** DI позволяет легко заменять реальные зависимости на заглушки или макеты, что особенно полезно при модульном тестировании. Вы сможете изолированно тестировать компоненты, не затрагивая все приложение. Это помогает быстрее находить ошибки, локализовать проблемы и упрощает отладку. Много примеров такого подхода вы найдете в главе 11.

- **Гибкость архитектуры.** DI позволяет вносить изменения в зависимости без влияния на основной код компонентов. Модули можно заменять, использовать повторно, отключать без риска сломать остальную систему. Инкапсуляция зависимостей помогает скрыть внутреннюю реализацию модуля и снижает вероятность возникновения побочных эффектов.
- **Удобство сопровождения и масштабируемость.** Централизованная регистрация зависимостей упрощает их обновление и поддержку. Команды могут параллельно работать над разными модулями, не мешая друг другу, ускоряя процесс разработки. Такой подход особенно полезен в больших многокомпонентных проектах, где нужно быстро масштабировать разработку.

## Механизмы внедрения зависимостей в .NET

Предположим, у нас есть интерфейс и его реализация, например:

```
public interface INotificationService
{
    void Notify(string message);
}

public class NotificationService : INotificationService
{
    public void Notify(string message)
    {
        // Отправка уведомления
    }
}
```

В .NET применяются три основных способа внедрения зависимостей.

- **Внедрение через конструктор.** Зависимости передаются в конструктор класса. Такой подход считается наилучшим, так как упрощает модульное тестирование сервисов:

```
public class DataService
{
    private readonly INotificationService _service;

    // Внедрение через конструктор
    public DataService(INotificationService service)
    {
        _service = service;
    }

    public void ProcessData(string data)
    {
        // Обработка данных и отправка уведомлений через внедренный сервис
        _service.Notify(data);
    }
}
```

- **Внедрение через свойства.** Зависимости присваиваются через публичное свойство класса. Этот подход полезен, если зависимость необязательна или может меняться в течение жизненного цикла объекта. Прием обеспечивает большую гибкость по сравнению с внедрением через конструктор:

```
public class DataService
{
    // Внедрение через свойство
    public INotificationService Service { get; set; }

    public void ProcessData(string data)
    {
        // Обработка данных и использование зависимости, если она установлена
        Service.Notify(data);
    }
}
```

- **Внедрение через метод.** Зависимости передаются как параметр метода. Уместно, если зависимость нужна только для одного метода и вы хотите сохранить класс сфокусированным на своих основных задачах и чистым:

```
public class DataService
{
    // ...прочий код

    // Внедрение через метод
    public void ProcessData(string data, INotificationService service)
    {
        // Использование зависимости внутри одного метода
        service.Notify(data);
    }
}
```

## Примеры в современном .NET

Платформа .NET имеет встроенную поддержку внедрения зависимостей (DI), что позволяет легко реализовать соответствующие паттерны в различных типах приложений. Один из самых распространенных и рекомендуемых подходов в ASP.NET Core — внедрение через конструктор. Его и рассмотрим первым.

### Пример внедрения через конструктор

Предположим, у вас есть интерфейс `IEmailService` и его реализация `EmailService`. Вы хотите внедрить этот сервис в контроллер `UserController`, отвечающий за регистрацию пользователей на сайте. Интерфейс, класс и контроллер показаны в следующем коде:

```
public interface IEmailService
{
    void SendEmail(string to, string subject, string body);
}
```

```

public class EmailService : IEmailService
{
    public void SendEmail(string to, string subject, string body)
    {
        // Код для отправки электронного письма
    }
}

public class UserRegistrationController
{
    private readonly IEmailService _emailService;

    public UserRegistrationController(IEmailService emailService)
    {
        _emailService = emailService;
    }

    public void SendUserConfirmationEmail(string userId)
    {
        // Реализация _emailService для отправки писем
    }
}

```

Чтобы DI-контейнер .NET знал, как разрешать эту зависимость, ее нужно зарегистрировать в разделе конфигурации сервисов, в файле `Program.cs`:

```

var builder = WebApplication.CreateBuilder(args);

// Добавление сервисов в контейнер
builder.Services.AddControllersWithViews();

builder.Services.AddSingleton<IEmailService, EmailService>();

// Регистрация других сервисов

var app = builder.Build();

```

Теперь контейнер знает, что при запросе `IEmailService` необходимо предоставить экземпляр `EmailService`.

## Примеры внедрения через свойства

В компонентах Blazor внедрение зависимостей можно осуществлять с помощью директивы `@inject`:

```
@inject IWeatherService WeatherService
```

Эта директива автоматически преобразуется в свойство с атрибутом `[Inject]`:

```
[Inject]
public IWeatherService WeatherService { get; set; }
```

Теперь вы можете использовать внедренное свойство внутри компонента как обычную зависимость:

```
weather = WeatherService.GetWeather();
```

Система внедрения зависимостей Blazor распознает атрибут [Inject] и вставляет соответствующий сервис автоматически.

Внедрение через свойства также может быть реализовано с помощью сторонних контейнеров или в собственных DI-решениях. Этот подход уместен в ситуациях, когда внедрение через конструктор не подходит или когда задействованы необязательные зависимости.

Пример: свойственное внедрение в Autofac. Начиная с версии Autofac 7.0 контейнер может автоматически разрешать все необходимые свойства аналогично параметрам конструктора:

```
public class NorthwindService
{
    // Эти свойства будут автоматически установлены Autofac
    public required ILogger Logger { protected get; init; }
    public required IConfigReader ConfigReader { protected get; init; }

    // Дополнительная реализация
}
```

**Важно:** все свойства должны быть разрешаемыми через контейнер — иначе при попытке создать объект будет выброшено исключение.

Для использования Autofac необходимо зарегистрировать зависимости:

```
ContainerBuilder builder = new();
builder.RegisterType<NorthwindService>();
builder.RegisterType<ConsoleLogger>().As<ILogger>();
builder.RegisterType<ConfigReader>().As<IConfigReader>();
var container = builder.Build();
```



Autofac — это сторонний DI/IoC-контейнер для .NET. Подробнее о нем можно узнать по адресу [autofac.org](http://autofac.org).

## Пример внедрения через метод

Внедрение через метод применяется, когда только определенный метод класса требует зависимость, а остальная часть — нет. Такой подход используется реже, чем внедрение через конструктор, но может быть полезен, особенно в минимальных API, где отсутствуют классы с конструкторами:

```
app.MapGet("/weather", (IWeatherService weatherService) =>
{
    return Results.Ok(new { Weather = weatherService.GetWeather() });
});
```

В методе `MapGet` параметр `IWeatherService` в лямбда-выражении указывает DI-контейнеру, что эту зависимость нужно внедрить автоматически. Контейнер анализирует сигнатуру метода-обработчика и разрешает `IWeatherService` из зарегистрированных сервисов, передавая его при выполнении запроса.

## Регистрация времени жизни сервиса зависимостей

В .NET вы можете зарегистрировать сервисы с различным временем жизни в зависимости от того, как долго должен существовать экземпляр сервиса. Ниже представлены три основных варианта.

- **Transient (временные сервисы).** Создаются каждый раз при обращении. Подходят для легких и независимых компонентов, не использующих общее состояние. Используйте их для утилит, вспомогательных сервисов, операций, где требуется новый экземпляр при каждом вызове.
- **Scoped (сервисы с областью видимости).** В ASP.NET Core создаются один раз на каждый HTTP-запрос и удаляются после отправки ответа клиенту. В других типах приложений (например, консольных) области видимости нужно создавать вручную. Вы можете также создавать дополнительные области видимости вручную, если требуется более тонкий контроль. Используются, когда нужно обеспечить отдельный экземпляр на каждый запрос либо изолировать работу с данными, например в контексте базы данных.
- **Singleton (одиночные сервисы).** Создаются один раз и затем используются повторно в течение всего времени работы приложения. Можно либо позволить контейнеру создать экземпляр, либо зарегистрировать вручную созданный объект. Применяются, когда требуется общий доступ к одному экземпляру, нужно сохранить состояние или конфигурацию или используется ресурс, не зависящий от запроса или пользователя.

Выбор подходящего времени жизни — важная часть проектирования архитектуры. Он влияет на эффективность использования ресурсов, надежность и предсказуемость поведения приложения и управление жизненным циклом объектов — когда и как они создаются и утилизируются.

В этой главе вы познакомитесь со всеми тремя типами времени жизни, управляемыми через область видимости DI-контейнера.

## Регистрация нескольких реализаций

Начиная с версии .NET 8 контейнер внедрения зависимостей поддерживает регистрацию сервисов с ключами. Это дает возможность регистрировать несколько реализаций одного интерфейса, разграничивать доступ к ним с помощью уникальных ключей и выбирать нужную реализацию в момент внедрения зависимости.

Рассмотрим пример с интерфейсом `IMemoryCache`, для которого регистрируются две разные реализации с различными ключами для идентификации:

```
builder.Services.AddKeyedSingleton<IMemoryCache, BigCache>("big");
builder.Services.AddKeyedSingleton<IMemoryCache, SmallCache>("small");

class BigCacheConsumer([FromKeyedServices("big")] IMemoryCache cache)
{
```

```
    public object? GetData() => cache.Get("data");
}

class SmallCacheConsumer(IKeyedServiceProvider keyedServiceProvider)
{
    public object? GetData() => keyedServiceProvider
        .GetRequiredKeyedService<IMemoryCache>("small").Get("data");
}
```

В приведенном выше коде:

- метод `AddKeyedSingleton` используется для регистрации сервисов с ключом в контейнере внедрения зависимостей. В данном случае регистрируются две реализации интерфейса `IMemoryCache`: `BigCache` с ключом "big" и `SmallCache` с ключом "small";
- класс `BigCacheConsumer` принимает параметр `IMemoryCache` с аннотацией `[FromKeyedServices("big")]`. То есть при создании экземпляра этого класса контейнер внедрит именно ту реализацию `IMemoryCache`, которая зарегистрирована с ключом "big". Я предполагаю этот способ получения сервиса по ключу;
- класс `SmallCacheConsumer` принимает параметр `IKeyedServiceProvider`. Он использует метод `GetRequiredKeyedService<IMemoryCache>("small")` для получения нужной реализации. Это альтернативный способ получения сервиса по ключу.

Ассоциируя сервисы с определенными ключами, вы получаете возможность гибко управлять зависимостями и внедрять нужные реализации на основе ключа — без необходимости создавать отдельные интерфейсы для каждой из них.

## Когда возникают исключения?

При разрешении зависимостей в DI-контейнере могут возникать различные сценарии, приводящие к исключениям. Понимание этих ситуаций помогает быстрее находить и устранять ошибки, связанные с конфигурацией внедрения зависимостей.

Приведу несколько распространенных случаев.

- **Сервис не зарегистрирован.** Если нужный сервис не зарегистрирован в DI-контейнере, попытка его разрешения вызовет исключение. Пример:

```
InvalidOperationException: Unable to resolve service for type 'IMyService'
while attempting to activate 'MyComponent'.
```

- **Циклическая зависимость.** Возникает, когда два или более компонента зависят друг от друга, образуя замкнутый цикл. В этом случае контейнер не может разрешить зависимости и выбрасывает исключение. Пример:

```
InvalidOperationException: A circular dependency was detected for the service
of type 'ServiceA'.
```

- **Отсутствие подходящего конструктора.** Если класс зависит от параметров конструктора, которые не могут быть разрешены, произойдет ошибка. Пример:

```
InvalidOperationException: Unable to resolve service for type  
'NonRegisteredDependency' while attempting to activate 'MyService'.
```

- **Неправильная конфигурация времени жизни.** Если сервисы зарегистрированы с несовместимым временем жизни, например **Scoped**-сервис зависит от **Transient** внутри **Singleton**, результатом могут стать исключения или непредсказуемое поведение. Например, **ObjectDisposedException**, если **Scoped**-сервис утилизирован, а **Singleton** продолжает на него ссылаться.
- **Несколько реализаций.** Если зарегистрировано несколько реализаций одного сервиса и не указано, какую использовать, контейнер либо выбрасывает исключение, либо выбирает конструктор неоднозначным образом. Пример:

```
InvalidOperationException: Multiple constructors accepting all given argument  
types have been found in type 'MyService2'. There should only be one applicable  
constructor.
```

- **Неверный дескриптор сервиса.** Если при регистрации указан тип, не реализующий интерфейс, контейнер не сможет сопоставить реализацию. Пример:

```
InvalidOperationException: Type 'MyService' does not implement interface  
'IMyService'.
```

Исключения при разрешении зависимостей чаще всего связаны с неправильной конфигурацией контейнера. Понимание таких ситуаций помогает избежать типичных ошибок и ускоряет отладку DI-инфраструктуры.

## Лучшие практики внедрения зависимостей (DI)

При использовании DI рекомендуется придерживаться следующих приемов.

- **Отдавайте предпочтение внедрению через конструктор.** Так зависимости класса будут явными, а объект будет создан в корректном состоянии.
- **Используйте интерфейсы в качестве зависимостей.** Работа через интерфейсы позволяет заменять реализации без изменения клиентского кода.
- **Избегайте шаблона сервис-локатора.** Сервис-локатор — это подход, при котором зависимость запрашивается явно внутри метода или конструктора, например через **IServiceProvider**. В этом случае нарушается принцип явности зависимостей, код становится менее понятным, тестирование затрудняется, а сопровождение усложняется.
- **Учитывайте область видимости и время жизни зависимостей.** Неправильно выбранное время жизни может привести к утечкам памяти, непредсказуемому поведению или ошибкам утилизации объектов.

Встроенный DI-контейнер в .NET помогает упростить управление зависимостями, позволяя разработчику сосредоточиться на бизнес-логике, а не на технических деталях создания и конфигурации объектов. Код становится чище и понятнее.

## Реализация универсального узла .NET

Универсальный узел (Generic Host) — это фреймворк для запуска и размещения .NET-приложений, впервые представленный в версии .NET Core 2.1 в 2018 году. Он предоставляет унифицированную модель запуска, которая охватывает управление временем жизни приложения, загрузку конфигурации, внедрение зависимостей (DI), журналирование, регистрацию фоновых сервисов и другие ключевые аспекты инфраструктуры. Generic Host используется для размещения разных типов приложений, в том числе веб-приложений на базе ASP.NET Core, фоновых сервисов и исполняемых консольных утилит.

## Ключевые особенности универсального узла .NET

Универсальный узел .NET (Generic Host) предоставляет ряд встроенных возможностей. Ниже перечислены основные из них.

- **Поддержка внедрения зависимостей (DI).** Generic Host автоматически настраивает контейнер зависимостей, что позволяет внедрять сервисы во все компоненты приложения без дополнительной конфигурации.
- **Инфраструктура журналирования.** Узел поставляется с предварительно настроенной системой журналирования, поддерживающей различных поставщиков.
- **Унифицированная конфигурация.** Generic Host предоставляет единый API для конфигурации приложений, поддерживающий JSON-файлы, переменные среды, аргументы командной строки и другие источники.
- **Управление жизненным циклом приложения.** Узел предоставляет интерфейсы и механизмы для реакции на события запуска, остановки и завершения.

Появление универсального узла позволило унифицировать процесс настройки и запуска .NET-приложений — от веб-приложений до фоновых сервисов. Он абстрагирует инфраструктурную рутину, связанную с инициализацией, конфигурацией и внедрением зависимостей, давая разработчикам возможность сосредоточиться на бизнес-логике, а не на шаблонном коде запуска.

## Создание универсального узла .NET

Универсальный узел .NET (Generic Host) отвечает за запуск приложения и управление его жизненным циклом. Хотя он чаще всего используется в веб-проектах ASP.NET Core и фоновых сервисах, вы можете использовать его в любом типе .NET-приложений, включая консольные утилиты.

Одним из ключевых методов для инициализации универсального узла является `CreateDefaultBuilder`. Этот метод выполняет предварительную настройку окружения, включая действия, перечисленные в табл. 10.1.

**Таблица 10.1.** Действия по настройке, выполняемые методом `CreateDefaultBuilder`

Действие	Описание
Загрузка конфигурации узла	Поддерживаются переменные среды с префиксом <code>DOTNET_</code> и аргументы командной строки
Загрузка конфигурации приложения	Поддерживаются: <code>appsettings.json</code> , <code>appsettings.{Environment}.json</code> , [Secret Manager] (в режиме разработки), переменные среды и аргументы командной строки
Добавление поставщиков журналирования	По умолчанию подключаются <code>Console</code> , <code>Debug</code> , <code>EventSource</code> и <code>EventLog</code> (только в Windows)
Проверка области видимости и зависимостей	Включается только в режиме разработки для предотвращения ошибок DI

Для создания универсального узла обычно используется `HostBuilder`, после чего происходит настройка сервисов (DI) и запуск узла.

Посмотрим на это в действии.

1. В редакторе кода создайте проект `Console App (Консольное приложение)` / `console`, присвойте ему имя `GenericHostApp` и добавьте в решение `Chapter10`.
2. В файле `GenericHostApp.csproj` проекта включите обработку предупреждений как ошибок, добавьте ссылку на пакет `Microsoft.Extensions.Hosting`, а также статически и глобально импортируйте класс `System.Console`:

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

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

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0" />
  </ItemGroup>

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

</Project>
```



Проверить наличие последней версии пакета Microsoft.Extensions.Hosting можно по адресу [www.nuget.org/packages/Microsoft.Extensions.Hosting/](http://www.nuget.org/packages/Microsoft.Extensions.Hosting/).

3. Создайте в проекте файл класса CounterService.cs.
4. В этом файле определите интерфейс и класс, реализующий счетчик. Класс должен увеличивать значение счетчика:

```
namespace Packt.Shared;

public interface ICounterService
{
    int Counter { get; set; }

    void IncrementCounter();
}

public class CounterService : ICounterService
{
    public int Counter { get; set; } = 0;

    public void IncrementCounter()
    {
        ++Counter;
    }
}
```



Обратите внимание, что у этого класса нет модификатора static и его члены также не являются статическими. В языке C# это означает, что экземпляры класса не могут быть общими для всех вызовов — каждый создается отдельно.

5. В файле Program.cs:
  - создайте и вызовите экземпляр сервиса напрямую;
  - затем зарегистрируйте его в DI-контейнере;
  - получите реализацию через контейнер и вызовите:

```
// Подключение методов GetRequiredService, AddSingleton,
// AddScoped, AddTransient
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting; // Подключение интерфейса IHostBuilder
                                    // и класса Host
using Packt.Shared; // Подключение интерфейса ICounterService
                    // и реализации CounterService

WriteLine("Creating a CounterService instance directly:");
ICounterService counterService = new CounterService();
counterService.IncrementCounter();
counterService.IncrementCounter();
WriteLine($"Counter: {counterService.Counter}");
```

```
WriteLine("Creating a CounterService instance indirectly using DI:");

// Создание узла с передачей аргументов, полученных консольным приложением
IHostBuilder builder = Host.CreateDefaultBuilder(args);

builder.ConfigureServices(services =>
{
    services.AddSingleton<ICounterService, CounterService>();
});

IHost host = builder.Build();

ICounterService service1, service2;

service1 = host.Services.GetRequiredService<ICounterService>();
service2 = host.Services.GetRequiredService<ICounterService>();

WriteLine($"Are the instances the same? {service1 == service2}");
service1.IncrementCounter();
service2.IncrementCounter();
WriteLine($"service1.Counter: {service1.Counter}");
WriteLine($"service2.Counter: {service2.Counter}");

await host.RunAsync();
```

6. Запустите проект `GenericHostApp` без отладки и изучите результат — вы увидите, что создается один экземпляр сервиса:

```
Creating a CounterService instance directly:
Counter: 2
Creating a CounterService instance indirectly using DI:
Are the instances the same? True
service1.Counter: 2
service2.Counter: 2
info: Microsoft.Hosting.Lifetime[0]
    Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
    Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
    Content root path: C:\tools-skills-net8\Chapter10\GenericHostApp\
bin\Debug\net8.0
```

7. Нажмите сочетание клавиш `Ctrl+C`, чтобы завершить работу узла.
8. В файле `Program.cs` закомментируйте строку, регистрирующую сервис как одиночку, а затем добавьте регистрацию в качестве временного сервиса:

```
builder.ConfigureServices(services =>
{
    // services.AddSingleton<ICounterService, CounterService>();
    services.AddTransient<ICounterService, CounterService>();
});
```

9. Перезапустите приложение и обратите внимание на результат — теперь создаются два разных экземпляра:

```
Creating a CounterService instance indirectly using DI:
Are the instances the same? False
service1.Counter: 1
service2.Counter: 1
```

10. В файле `Program.cs` закомментируйте строку с временной регистрацией и замените ее на регистрацию с областью видимости:

```
builder.ConfigureServices(services =>
{
    // services.AddSingleton<ICounterService, CounterService>();
    // services.AddTransient<ICounterService, CounterService>();
    services.AddScoped<ICounterService, CounterService>();
});
```

11. В файле `Program.cs` добавьте третью переменную — `service3`. Оберните выражения, которые получают и вызывают сервисы, в область видимости, созданную с помощью метода `CreateScope`. Затем определите вторую область видимости, в которой снова получите и вызовете сервис. При этом не забудьте изменить способ получения зависимостей — вместо `host.Services` используйте `scope.ServiceProvider`:

```
ICounterService service1, service2, service3;
// Если используется сервис с областью видимости, необходимо,
// чтобы была хотя бы одна явно созданная область
using (IServiceScope scope = host.Services.CreateScope())
{
    service1 = scope.ServiceProvider.GetRequiredService<ICounterService>();
    service2 = scope.ServiceProvider.GetRequiredService<ICounterService>();

    WriteLine($"Are the instances the same? {service1 == service2}");
    service1.IncrementCounter();
    service2.IncrementCounter();
    WriteLine($"service1.Counter: {service1.Counter}");
    WriteLine($"service2.Counter: {service2.Counter}");
}

using (IServiceScope scope = host.Services.CreateScope())
{
    service3 = scope.ServiceProvider.GetRequiredService<ICounterService>();

    WriteLine($"Are the instances the same? {service1 == service3}");
    WriteLine($"service1.Counter: {service1.Counter}");
    WriteLine($"service2.Counter: {service2.Counter}");
    WriteLine($"service3.Counter: {service3.Counter}");
}

await host.RunAsync();
```



Получать зависимости необходимо через scope.ServiceProvider, а не напрямую из host.Services.

12. Запустите проект GenericHostApp без отладки. Вы увидите, что было создано два экземпляра сервиса, как показано ниже:

```
Creating a CounterService instance indirectly using DI:
Are the instances the same? True
service1.Counter: 2
service2.Counter: 2
Are the instances the same? False
service1.Counter: 2
service2.Counter: 2
service3.Counter: 0
```

В проектах ASP.NET Core для каждого HTTP-запроса автоматически создается область видимости. Это означает, что:

- все сервисы с областью видимости, созданные в рамках одного запроса, — это один и тот же экземпляр;
- все временные сервисы, созданные во время одного запроса, — это разные экземпляры;
- одиночки всегда представляют собой один общий экземпляр, доступный на протяжении всего времени работы приложения.

Теперь, когда вы разобрались с управлением областью видимости и временем жизни сервисов, рассмотрим иные преимущества использования универсального узла .NET.

## Обзор сервисов и событий узла

*Сервис узла* — это компонент, реализующий интерфейс `IHostedService`. Такой компонент должен реализовать два метода: `StartAsync` — вызывается при запуске узла и `StopAsync` — вызывается при завершении работы узла:

```
public class MyBackgroundService : IHostedService
{
    public Task StartAsync(CancellationToken cancellationToken)
    {
        // Код для запуска фоновой задачи
        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        // Код для остановки фоновой задачи
        return Task.CompletedTask;
    }
}
```

Метод `StartAsync` вызывается при запуске интерфейса `IHostedService`. Он содержит логику для запуска фоновой задачи. Метод `StopAsync` вызывается, когда `IHostedService` останавливается. Он содержит логику для плавного завершения фоновой задачи. Оба метода асинхронны, что позволяет запускать и останавливать фоновую задачу без блокировки потока выполнения.

Чтобы использовать реализацию `IHostedService`, вам необходимо зарегистрировать ее в контейнере внедрения зависимостей (DI) в вашем файле `Program.cs`, например:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddHostedService<MyBackgroundService>();

var app = builder.Build();
app.Run();
```

Помимо регистрации собственных сервисов в контейнере узла, .NET также автоматически регистрирует некоторые специальные сервисы, как показано в табл. 10.2.

**Таблица 10.2.** Автоматически зарегистрированные сервисы

Сервис	Описание
<code>IHostApplicationLifetime</code>	Предоставляет три метода событий для прослушивания. <ul style="list-style-type: none"> <li>• <code>ApplicationStarted</code> — срабатывает, когда приложение полностью запущено. Полезно для инициализации после запуска.</li> <li>• <code>ApplicationStopping</code> — срабатывает при начале завершения работы. Полезно для освобождения ресурсов и остановки сервисов.</li> <li>• <code>ApplicationStopped</code> — срабатывает, когда приложение полностью остановлено. Полезно для выполнения задач по окончательной очистке</li> </ul>
<code>IHostLifetime</code>	Интерфейс, как правило, реализуется компонентами инфраструктуры, а не используется напрямую в коде приложений. <ul style="list-style-type: none"> <li>• <code>WaitForStartAsync</code> — откладывает запуск до наступления определенных условий.</li> <li>• <code>StopAsync</code> — управляет завершением работы и очисткой ресурсов</li> </ul>
<code>IHostEnvironment</code>	Предоставляет информацию о текущей среде выполнения
<code>ILogger</code>	Сервис журналирования. Подключаются поставщики <code>Console</code> , <code>EventLog</code> и другие по умолчанию

Разберемся, для чего могут использоваться эти сервисы.

1. В проекте `GenericHostApp` создайте файл класса `WorkerService.cs`.
2. В этом файле определите класс, содержащий методы для обработки общих событий жизненного цикла узла:

```
using Microsoft.Extensions.Hosting; // Подключение интерфейса IHostedService
using Microsoft.Extensions.Logging; // Подключение интерфейса ILogger

namespace Packt.Shared;
```

```
public sealed class WorkerService : IHostedService
{
    private readonly ILogger _logger;
    private readonly IHostEnvironment _environment;
    private readonly IHostApplicationLifetime _appLifetime;

    public WorkerService(
        ILogger<WorkerService> logger,
        IHostApplicationLifetime appLifetime,
        IHostEnvironment hostEnvironment)
    {
        _logger = logger;
        _logger.LogInformation("WorkerService constructor has been called.");

        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);
        _appLifetime = appLifetime;

        _environment = hostEnvironment;
        WriteLine($"_environment.EnvironmentName: {_environment.EnvironmentName}");
        WriteLine($"_environment.ApplicationName: {_environment.ApplicationName}");
        WriteLine($"_environment.ContentRootPath: {_environment.ContentRootPath}");
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("1. StartAsync has been called.");

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("4. StopAsync has been called.");

        return Task.CompletedTask;
    }

    private void OnStarted()
    {
        _logger.LogInformation("2. OnStarted has been called.");
    }

    private void OnStopping()
    {
        _logger.LogInformation("3. OnStopping has been called.");
    }

    private void OnStopped()
    {
        _logger.LogInformation("5. OnStopped has been called.");
    }

    private void GracefulShutdown()
    {
        _appLifetime.StopApplication();
    }
}
```



**Дельный совет:** быстрый, но грубый способ завершить работу приложения — это вызов Environment.Exit(...). Однако, если ваше приложение работает через механизм хостинга, лучше использовать метод IHostApplicationLifetime.StopApplication(), чтобы обеспечить корректное завершение работы всех сервисов и освобождение ресурсов.

3. В файле `Program.cs` зарегистрируйте реализацию размещенного сервиса, чтобы его экземпляр создавался и запускался автоматически при запуске узла:

```
builder.ConfigureServices(services =>
{
    // services.AddSingleton<ICounterService, CounterService>();
    // services.AddTransient<ICounterService, CounterService>();
    services.AddScoped<ICounterService, CounterService>();

    // Регистрация размещенного сервиса, который будет выполняться при запуске узла
    services.AddHostedService<WorkerService>();
});
```

4. В файле `Program.cs` добавьте поясняющий комментарий перед запуском узла:

```
// Запуск узла и любых размещенных на нем сервисов с ожиданием их завершения
await host.RunAsync();
```

5. Запустите проект `GenericHostApp` без отладки и обратите внимание на результат. В выводе отобразится информация о среде выполнения, как показано в следующем примере:

```
info: Packt.Shared.WorkerService[0]
      WorkerService constructor has been called.
_environment.EnvironmentName: Production
_environment.ApplicationName: GenericHostApp
_environment.ContentRootPath: C:\tools-skills-net8\Chapter10\
GenericHostApp\bin\Debug\net8.0
info: Packt.Shared.WorkerService[0]
      1. StartAsync has been called.
info: Packt.Shared.WorkerService[0]
      2. OnStarted has been called.
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\tools-skills-net8\Chapter10\GenericHostApp\
bin\Debug\net8.0
```



По умолчанию используется среда `Production`, если явно не задано иное — например, в переменной среды или через командную строку. Чтобы запустить проект в среде `Development`, можно воспользоваться командой в терминале:

```
dotnet run --environment Development
```

6. Нажмите сочетание клавиш Ctrl+C, чтобы остановить узел, и обратите внимание на результат, который будет выведен в консоль:

```
info: Packt.Shared.WorkerService[0]
  3. OnStopping has been called.
info: Microsoft.Hosting.Lifetime[0]
  Application is shutting down...
info: Packt.Shared.WorkerService[0]
  4. StopAsync has been called.
info: Packt.Shared.WorkerService[0]
  5. OnStopped has been called.
```

## Методы регистрации сервисов

В .NET предусмотрено несколько способов регистрации сервисов. Они позволяют гибко управлять зависимостями, в том числе на одном уровне времени жизни, например при регистрации одиночек.

Таблица 10.3 демонстрирует различные способы регистрации и поддерживаемые ими характеристики.



В столбцах характеристик (см. табл. 10.3):

- А означает автоматическое удаление (поддержка IDisposable);
- Н означает возможность регистрации нескольких реализаций;
- П означает возможность передачи аргументов в конструктор.

**Таблица 10.3.** Методы регистрации сервисов

Метод регистрации	А	Н	П
AddSingleton<IMyService, MyService>() AddSingleton<IMyService, Service2>()	Да	Да	Нет
AddSingleton<IMyService>(sp => new MyService()) AddSingleton<IMyService>(sp => new MyService(13)) AddSingleton<IMyService>(sp => new Service2(27))	Да	Да	Да
AddSingleton<MyService>()	Да	Нет	Нет
AddSingleton<IMyService>(new MyService()) AddSingleton<IMyService>(new MyService(13)) AddSingleton<IMyService>(new Service2(27))	Нет	Да	Да
AddSingleton(new MyService()) AddSingleton(new MyService(13))	Нет	Нет	Да

Чтобы получить несколько реализаций, зарегистрированных для одного и того же интерфейса, объягите параметр типа `IEnumerable<T>`:

```
public WorkerService(
    ILogger lastLoggerRegistered,
    IEnumerable<	ILogger> allRegisteredLoggers)
```

Если в `WorkerService` определены два параметра конструктора, параметр `ILogger` получит последнюю зарегистрированную реализацию, а параметр `IEnumerable<ILogger>` — все зарегистрированные реализации.

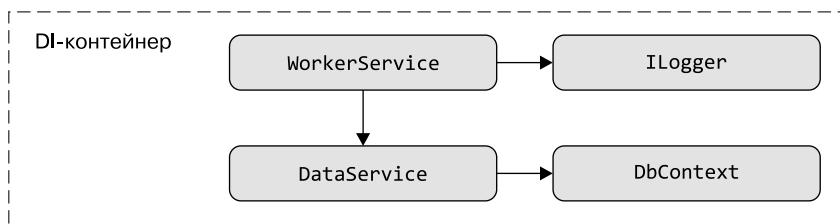


Если реализация интерфейса уже зарегистрирована, то вызов метода `TryAdd...` не перезапишет существующую регистрацию.

## Графы зависимостей и разрешение сервисов

Обычно техника внедрения зависимостей (DI) используется в цепочке. Каждая запрашиваемая зависимость, например `WorkerService`, в свою очередь, запрашивает собственные зависимости, например интерфейс `ILogger`. Контейнер внедрения зависимостей (DI) находит нужные зависимости в своем графе сервисов и возвращает полностью разрешенный сервис.

Набор зависимостей, которые необходимо разрешить, называется деревом зависимостей, графом зависимостей или графиком объектов (рис. 10.2).



**Рис. 10.2.** DI-контейнер и дерево/граф зависимостей

На рис. 10.2 DI-контейнер управляет зависимостями.

- `WorkerService` — это сервис, который зависит от других сервисов.
- `ILogger` — зависимость, необходимая `WorkerService`.
- `DataService` — еще одна зависимость, которая может понадобиться `WorkerService` или другим сервисам.
- `DbContext` — зависимость, необходимая сервису `DataService`.

Стрелка от `WorkerService` к `ILogger` показывает, что `WorkerService` зависит от `ILogger`. DI-контейнер разрешает `WorkerService` и в процессе разрешает его зави-

симости, такие как `ILogger` и `DataService`, которые, в свою очередь, могут иметь собственные зависимости, например `DbContext`. Такое разрешение по цепочке гарантирует, что все зависимости будут полностью разрешены и внедрены в нужные места.

## Удаление сервисов

Если сервис реализует интерфейс `IDisposable`, то DI-контейнер вызывает метод `Dispose` для сервисов, которые он создает автоматически. Нельзя самостоятельно утилизировать сервисы, созданные DI-контейнером, потому что:

- невозможно заранее знать, как устроено дерево зависимостей или граф объектов;
- то, что сервис больше не нужен вам, не означает, что он не используется другими сервисами;
- также нельзя точно знать, на какое время жизни он был зарегистрирован.

Преждевременная утилизация таких сервисов может привести к отказам других компонентов, которые продолжают на них полагаться.

Если же вы вручную создаете экземпляр сервиса и передаете его в контейнер, то ответственность за его утилизацию ложится на вас:

```
// Сервис инстанцирован вручную, поэтому должен быть вручную утилизирован
builder.Services.AddSingleton(new NorthwindService());
```

## Внедрение зависимостей с помощью ASP.NET Core

Наиболее распространенный тип проекта, в котором широко применяется внедрение зависимостей (DI), — это проект ASP.NET Core. В этом разделе рассматриваются особые случаи использования DI при создании веб-приложений и веб-сервисов.

### Регистрация сервисов для функций с помощью методов расширения

В крупных проектах ASP.NET Core обычно требуется зарегистрировать множество взаимосвязанных сервисов для каждой функциональной области сайта или веб-сервиса — как правило, в файле `Program.cs`:

```
builder.Services.AddScoped<IShoppingCart, InMemoryShoppingCart>();
builder.Services.AddScoped<ICustomerAccount, CustomerAccount>();
builder.Services.AddScoped<IUserRegistration, UserRegistration>();
```

Рекомендуется создать метод расширения, в котором группируются все связанные регистрации сервисов для конкретной функциональности:

```
public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddNorthwindFeatures(
        this IServiceCollection services)
    {
        services.AddScoped<IShoppingCart, InMemoryShoppingCart>();
        services.AddScoped<ICustomerAccount, CustomerAccount>();
        services.AddScoped<IUserRegistration, UserRegistration>();
        return services;
    }
}
```

Такой подход упрощает конфигурацию в файле `Program.cs`, делая ее компактной и структурированной:

```
builder.Services.AddNorthwindFeatures();
```

Он используется и внутри самой платформы ASP.NET Core, например в методах `AddControllers()`, `AddRazorPages()`.

## Когда конструктор не подходит для получения зависимостей

В проектах ASP.NET Core на основе MVC или веб-API контроллеры могут получать зависимости через конструктор — как и любые другие классы. Это обычный и рекомендуемый подход. Однако бывают ситуации, в которых такой способ не подходит или даже невозможен. В некоторых случаях попытка получить зависимость через конструктор приводит к тому, что объект, рассчитанный на ограниченный срок жизни, начинает вести себя как одиночка. Это нарушает принцип области видимости и может вызывать ошибки во время выполнения.

В ASP.NET Core такие ситуации возникают, как правило, если время жизни зависимостей несовместимо (например, временный или ограниченный по области сервис внедряется в одиночку) либо если контекст, в котором требуется зависимость, не поддерживает внедрение через конструктор, как, например, в случае некоторых встроенных компонентов фреймворка.

Перечислю некоторые из таких ситуаций.

- Фоновые сервисы, например те, что наследуются от `BackgroundService` или реализуют интерфейс `IHostedService`, обычно регистрируются как одиночки. В таких случаях внедрение сервисов с ограниченной областью видимости через конструктор не подходит, потому что они будут рассматриваться как одиночки. Это может привести к проблемам с управлением ресурсами и состоянием. Лучше получать такие зависимости внутри метода `ExecuteAsync`, воспользовавшись `IServiceProvider`.

- Вспомогательные методы тегов создаются для каждого экземпляра представления, и DI-контейнер не управляет ими так же, как контроллерами или другими сервисами. Поэтому внедрение через конструктор во вспомогательных методах тегов не поддерживается. Вместо этого зависимости можно получать через свойства с атрибутом [ViewContext].
- Фильтры могут быть зарегистрированы глобально, на уровне контроллера или действия. Если фильтр зарегистрирован как одиночка, он не может напрямую зависеть от сервисов с областью видимости через конструктор. В этом случае можно использовать атрибуты `ServiceFilter` или `TypeFilter` либо вручную разрешать зависимости через `IServiceProvider` внутри метода фильтра.

В целом внедрение через конструктор не всегда применимо, особенно при работе с промежуточным ПО, в фоновых задачах, вспомогательных методах тегов и фильтрах. В таких случаях лучше получать зависимости через методы, свойства или путем явного разрешения зависимостей в области видимости метода, чтобы обеспечить корректное создание и утилизацию сервисов с учетом их времени жизни.

Теперь подробнее разберем работу с промежуточным ПО.

## Использование сервисов с областью видимости в промежуточном ПО

В промежуточном ПО (middleware) можно передавать одиночные сервисы напрямую через конструктор. Однако если требуется использовать временные сервисы или сервисы с областью видимости, их следует получать не через конструктор, а внутри метода `Invoke` или `InvokeAsync`:

```
public class NorthwindMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger _logger;

    private readonly ISingletonService _singleton;

    // Сервисы-одиночки можно получать через конструктор
    public MyMiddleware(RequestDelegate next, ILogger<MyMiddleware> logger,
        ISingletonService singleton)
    {
        _logger = logger;
        _singleton = singleton;
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context,
        // Временные сервисы и сервисы с областью видимости – только внутри
        // метода Invoke/InvokeAsync
        ITransientService transient, IScopeService scoped)
    {
        _logger.LogInformation("Transient: " + transient.ProductId);
        _logger.LogInformation("Scoped: " + scoped.ProductId);
    }
}
```

```

    _logger.LogInformation("Singleton: " + _singleton.ProduceId);
    await _next(context);
}
}

```

## Разрешение сервисов при запуске

Если необходимо получить сервис с областью видимости во время запуска проекта ASP.NET Core, нужно вручную создать область видимости, чтобы корректно управлять временем жизни зависимостей:

```

using Packt.Shared; // Подключение интерфейса INorthwindService

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddScoped<INorthwindService, NorthwindService>();
WebApplication app = builder.Build();

using (IServiceScope scope = app.Services.CreateScope())
{
    INorthwindService service = scope.ServiceProvider
        .GetRequiredService<INorthwindService>();

    // Сервис
}

app.MapGet("/", () => "Hello World!");
app.Run();

```

В остальных частях проекта — например, в контроллерах или Razor-компонентах — область видимости создается автоматически для каждого входящего HTTP-запроса.

Помимо внедрения через конструктор, можно получить контейнер сервисов из текущего контекста:

```
IServiceProvider services = HttpContext.RequestServices;
```

## Внедрение зависимостей в представлениях

Чтобы получать зависимости прямо в представлениях Razor или страницах Razor Page, используется директива `@inject`:

```
@inject NorthwindService nw
```

## Методы действий и минимальные API

Если вы работаете с MVC-сайтами на контроллерах, веб-API или минимальными API, зависимости можно передавать через параметры методов, пометив их атрибутами, например `[FromServices]` или `[FromKeyedServices]`.

Сначала регистрируются нужные сервисы:

```
@inject NorthwindService nw
```

Затем в методе действия параметры помечаются атрибутами:

```
builder.Services.AddSingleton<IOrderProcessor, OrderProcessor>();
builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
public IActionResult SubmitOrder([FromServices] IOOrderProcessor processor)
{
    // Использование полученного сервиса
}

public ActionResult<object> GetCatalog([FromKeyedServices("big")] ICache cache)
{
    return cache.Get("catalog");
}
```

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 10.1. Дополнительные онлайн-материалы

Вы можете прочитать статью Мартина Фаулера *Inversion of Control Containers and the Dependency Injection pattern*, написанную более 20 лет назад, по адресу [www.martinfowler.com/articles/injection.html](http://www.martinfowler.com/articles/injection.html).

Дополнительные сведения о DI в ASP.NET Core Blazor представлены в официальной документации Microsoft по адресу [learn.microsoft.com/ru-ru/aspnet/core/blazor/fundamentals/dependency-injection](https://learn.microsoft.com/ru-ru/aspnet/core/blazor/fundamentals/dependency-injection).

### Упражнение 10.2. Практические задания

Если вы хотите попробовать альтернативу встроенной системе DI, рекомендую опробовать Autofac. Изучив, как устроены другие контейнеры, вы сможете лучше понять общие принципы процесса. Простой учебник по Autofac доступен по адресу [autofac.readthedocs.io/en/latest/getting-started/index.html](https://autofac.readthedocs.io/en/latest/getting-started/index.html).

### Упражнение 10.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. В чем заключается основная идея внедрения зависимостей (DI)?
2. Какие существуют три основных способа внедрения зависимостей? Какой из них не поддерживается в .NET напрямую, если не использовать специальные подходы?

3. Какие типы времени жизни используются при регистрации сервисов в .NET и чем они отличаются?
4. Какие события можно обрабатывать при работе с сервисом узла?
5. В какой момент запускаются сервисы узла в жизненном цикле приложения?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 10.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-10---dependency-injection-containers-and-service-lifetime](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-10---dependency-injection-containers-and-service-lifetime).

## Резюме

В этой главе вы изучили:

- основные концепции инверсии управления (IoC) и внедрения зависимостей (DI);
- как регистрировать и удалять сервисы;
- что такое время жизни сервиса и в каких случаях управлять им;
- как реализуется универсальный узел .NET;
- как внедрение зависимостей осуществляется в ASP.NET Core.

В следующей главе вы познакомитесь с различными типами тестирования, а затем подробно изучите модульное тестирование, включая техники имитации зависимостей. Это логичное продолжение текущей темы.

# 11 Модульное и мок-тестирование

Эта глава посвящена модульному и мок-тестированию. Однако начнется она с обзора всех основных видов тестирования, о которых должен знать разработчик .NET. Мы начнем с модульного и мок-тестирования, поскольку эти техники фокусируются на самых мелких единицах кода — отдельных методах и функциях. Такая детализация делает их естественной основой любой стратегии тестирования. Модульные тесты помогают убедиться, что каждая часть программы работает, как задумано, — изолированно, без влияния других компонентов. Они проверяют, что небольшие фрагменты кода функционируют корректно сами по себе. Мок-тестирование дополняет этот подход: с его помощью можно имитировать поведение сложных зависимостей, что делает модульные тесты проще и надежнее.

Прежде чем перейти к конкретным техникам, стоит рассмотреть более широкий набор методологий тестирования, на которых строится надежная разработка ПО. *Тестирование* — это не единая практика, а целый спектр стратегий, каждая из которых охватывает разные аспекты работы приложения.

В последующих главах мы рассмотрим и другие виды тестирования — интеграционное, проверку безопасности, тестирование производительности, функциональное тестирование, а также сквозное тестирование пользовательского интерфейса для сайтов и сервисов. Эти темы охватываются в отдельных главах. Интеграционное тестирование помогает понять, как различные модули приложения взаимодействуют между собой и работают в связке. Проверка безопасности показывает, насколько устойчив код к уязвимостям и вредоносным воздействиям. Тестирование производительности помогает оценить поведение приложения в различных условиях, выявить узкие места и повысить общую эффективность. Функциональное тестирование позволяет проверить, соответствует ли поведение приложения ожиданиям пользователей. А сквозное тестирование пользовательского интерфейса (UI) показывает, насколько последовательно и стабильно работает интерфейс — от первого взаимодействия до финального результата.

## Обзор всех видов тестирования

Тестирование — один из ключевых этапов в процессе разработки ПО. Качественные тесты помогают сделать приложение надежным, устойчивым к сбоям и готовым к эксплуатации. А неэффективные тесты могут обернуться серьезными затратами. Поэтому важно разумно распределять усилия, направленные на создание тестов. Что отличает качественный тест от неэффективного — один из центральных вопросов, и в этой, а также в последующих главах вы разберетесь с этим подробно.

Стратегия тестирования должна охватывать разные стороны работы приложения — от поиска ошибок и защиты от регрессий до оценки производительности и удобства использования. Так вы сможете заранее устраниить потенциальные проблемы, прежде чем они повлияют на пользователей.

### Модульное тестирование

*Модульное тестирование* направлено на изолированную проверку отдельных компонентов проекта. Цель — убедиться, что каждый элемент работает так, как задумано. Здесь в центре внимания бизнес-логика, алгоритмы и отдельные функции или методы.

Рекомендуется создать хотя бы один модульный тест для каждого метода, входящего в публичный API. Но обратите внимание: речь не обо *всех* методах подряд.

Непубличные методы, как правило, не тестируются напрямую. На это есть несколько причин:

- при тестировании через публичные интерфейсы косвенно проверяются и непубличные методы;
- непубличные методы — часть внутренней реализации, которая может со временем измениться;
- акцент на поведении, а не на реализации помогает писать надежные и легко сопровождаемые тесты;
- меньшие по объему тестовые наборы проще поддерживать;
- тесты более высокого уровня — интеграционные или сквозные — способны охватить сложные сценарии взаимодействия.

Если вы сосредоточитесь на тестировании публичных методов, ваши тесты будут более осмысленными, стабильными и устойчивыми к изменениям. Такой подход повышает эффективность и упрощает поддержку тестов.

Качественный модульный тест обычно:

- проверяет один конкретный аспект поведения — например, метод с бизнес-логикой;
- выполняется максимально быстро благодаря замене производственных зависимостей, например, использованию хранилища в памяти вместо реальной

базы данных. Качественный тестовый фреймворк позволяет задать тайм-аут, чтобы тест завершался, если он выполняется слишком долго;

- работает изолированно — независимо от других тестов и, при необходимости, от внешних компонентов.

Среди популярных инструментов для модульного тестирования в .NET — xUnit, NUnit и MSTest. В этой книге мы сосредоточимся на xUnit. Однако если вы используете другой инструмент, многие принципы все равно окажутся полезными — подходы у них во многом схожи. Далее в главе мы подробнее разберем, как применять модульное тестирование на практике.

## Интеграционное и сквозное тестирование, проверка безопасности

К тестированию более высокого уровня относятся интеграционные, сквозные тесты и проверки безопасности.

*Интеграционные тесты* могут внешне напоминать модульные и использовать те же инструменты — например, xUnit. Но, в отличие от модульных, они не соответствуют одному или нескольким ключевым критериям. Такие тесты часто охватывают больше кода, выполняются дольше и не всегда работают в полной изоляции.

Задача интеграционного тестирования — убедиться, что отдельные компоненты, модули и сервисы проекта корректно взаимодействуют между собой. Интеграционные тесты должны охватывать, в частности, работу с базами данных, интеграцию с внешними API и взаимодействие между слоями приложения — например, между уровнем доступа к данным и уровнем бизнес-логики.

*Сквозное тестирование* — подвид интеграционного, при котором проверяется работа системы с точки зрения конечного пользователя. Оно охватывает полный пользовательский сценарий — от ввода данных до результата — и позволяет убедиться, что все работает последовательно и согласованно на всех уровнях.

*Проверка (тестирование) безопасности* направлена на выявление уязвимостей в приложениях и сервисах, а также на защиту данных от несанкционированного доступа. Она включает проверку механизмов оценки подлинности и авторизации, шифрования данных, защиты от распространенных угроз, таких как SQL-инъекции, межсайтовый скрипting и т.п.

Для интеграционного тестирования используются те же инструменты, что и для модульного, — разница заключается не в средствах, а в цели тестов. В сфере безопасности часто применяются рекомендации OWASP и встроенные анализаторы безопасности для .NET.

Интеграционные, сквозные и тесты безопасности будут подробно рассмотрены в главе 12.

## Тестирование производительности, нагрузки и стресс-тестирование

Задача тестирования производительности, нагрузки и стресс-тестирования — убедиться, что приложение стабильно работает при ожидаемой нагрузке. В ходе таких тестов проверяются время отклика сервисов, пропускная способность системы и устойчивость к одновременным обращениям со стороны большого числа пользователей или запросов без снижения производительности.

Среди популярных инструментов — Apache JMeter, BenchmarkDotNet и k6. Эти виды тестирования подробно рассматриваются в главе 13.

## Функциональное и юзабилити-тестирование

*Функциональное тестирование* направлено на проверку, соответствует ли приложение заданным требованиям и ведет ли себя корректно во всех пользовательских сценариях. В рамках таких тестов оцениваются сценарии взаимодействия пользователя с системой, ключевые рабочие процессы и сквозные задачи, включая ввод данных через формы и навигационные потоки.

*Проверка удобства использования (юзабилити-тестирование)* фокусируется на интерфейсе и общем пользовательском опыте. В рамках этой проверки стоит обратить внимание на простоту и интуитивность взаимодействия, согласованность дизайна, структуру навигации и доступность для разных категорий пользователей. Применяются также такие методы, как A/B-тестирование, опросы и сессии с участием пользователей.

Среди распространенных инструментов для функционального тестирования — Playwright и Selenium для веб-приложений, а также Appium для мобильных. Об этих подходах подробнее рассказано в главе 14.

## Терминология тестирования

Прежде чем двигаться дальше, разберем несколько общих терминов, используемых в тестировании. Они сведены в табл. 11.1.

**Таблица 11.1.** Распространенные термины в области тестирования

Термин	Определение
SUT, MUT	<p><b>Тестируемая система (System Under Test, SUT)</b> — это тип, например класс, который проходит тестирование. Обычно для него создается отдельный тестовый класс, в котором сгруппированы все тестовые методы, относящиеся к SUT.</p> <p><b>Тестируемый метод (Method Under Test, MUT)</b> — это метод внутри SUT, который непосредственно проверяется</p>

<b>Термин</b>	<b>Определение</b>
<b>Тестовый дублер</b>	Объект, обладающий тем же публичным API, что и зависимость, но с упрощенным и предсказуемым поведением. Название связано с дублерами-каскадерами в кинопроизводстве и не имеет отношения к типу данных <code>double</code> . Этот термин охватывает все непроизводственные зависимости, используемые исключительно для тестирования
<b>Имитация (mock)</b>	Подтип тестового дублера, предназначенный для проверки взаимодействий между объектами. Для имитации задаются ожидаемые вызовы и поведение, а затем проверяется, были ли ожидания выполнены во время теста
<b>Регрессия</b>	Нарушение корректной работы кода после его изменения или модификации
<b>Метрика покрытия</b>	Показатель, отражающий долю кода, выполненного в процессе тестирования. Измеряется в процентах от 0 до 100. Рекомендуется стремиться к высокому уровню покрытия в основной бизнес-логике проекта, однако не стоит делать это строгим требованием для непрофильных частей. Четкое разделение бизнес-логики помогает определить, что именно стоит тестировать в первую очередь
<b>Фикстуры для тестирования</b>	Объекты или данные, необходимые для выполнения теста. Это может быть переданный аргумент, подготовленный файл или запись в базе данных. Важно, чтобы фикстура оставалась стабильной, обеспечивая одинаковый результат теста при каждом запуске

Успех зависит от того, насколько серьезно вы и ваша команда относитесь к тестированию.

## Характеристики качественных тестов

Качественные тесты предоставляют следующие функции.

- Проверку важнейших частей кодовой базы.** В случае модульных тестов это обычно модель предметной области и алгоритмы бизнес-логики. Для интеграционных тестов — контроллеры или оркестраторы, управляющие процессами с участием нескольких внешних систем. Покрытие кода тестами не обязательно должно быть полным. Важно тестировать конечный результат процесса, а не его техническую реализацию.
- Автоматическую интеграцию в процесс разработки.** Настройте систему непрерывной интеграции и развертывания так, чтобы тесты запускались автоматически.
- Выявление регрессий.** По мере развития проекта и роста сложности кодовой базы могут возникать ошибки, нарушающие работу приложения. Качественные тесты позволяют быстро обнаруживать такие проблемы и устранять их на ранней стадии, выступая системой раннего предупреждения.

4. **Устойчивость к рефакторингу.** После изменения реализации функции ее тесты должны продолжать успешно выполняться. Если тесты перестают проходить из-за рефакторинга, они считаются нестабильными и дают ложные срабатывания.
5. **Баланс затрат и выгод.** Важно находить разумное соотношение между стоимостью поддержки тестов и их реальной пользой. Тестирование тривиальных элементов, например установки и получения свойств модели, редко оправдывает затраченные усилия.



**Дельный совет:** лучше вообще не создавать тест, чем написать неэффективный. Каждое дополнительное выражение в тестах увеличивает стоимость поддержки проекта. Если тест не приносит пользы, он становится обузой.

## Результаты тестирования

При обсуждении результатов тестирования часто используются термины «положительный» и «отрицательный». Они показывают, указывает ли тест на наличие или отсутствие дефекта. Как и в медицине, отрицательный результат — это хорошо!

Также применяются термины «истина» и «ложь», которые отражают правильность результата теста относительно реального состояния кода. Истина означает корректный результат, ложь — ошибочный.

Часто для обозначения комбинаций используют сокращения:

- **TP (True Positive)** — истинно положительный результат;
- **TN (True Negative)** — истинно отрицательный результат;
- **FP (False Positive)** — ложноположительный результат;
- **FN (False Negative)** — ложноотрицательный результат.

Комбинируя термины «положительный/отрицательный» и «истина/ложь», можно получить четыре возможных исхода тестирования, как показано в табл. 11.2.

**Таблица 11.2.** Четыре возможных результата тестирования

	<b>Положительный результат</b>	<b>Отрицательный результат</b>
<b>Истина</b>	<b>TP (истинно положительный):</b> тест правильно обнаруживает дефект. Код действительно неисправен, и тест это показывает. Хороший результат	<b>TN (истинно отрицательный):</b> тест правильно подтверждает отсутствие дефекта. Код корректен, и тест это отражает. Хороший результат
<b>Ложь</b>	<b>FP (ложноположительный):</b> тест ошибочно сообщает о дефекте. На самом деле код работает корректно. Плохой результат	<b>FN (ложноотрицательный):</b> тест не выявляет реальный дефект. Код содержит ошибку, но тест ошибочно показывает, что все в порядке. Плохой результат

Чтобы лучше разобраться в этих вариантах, рассмотрим несколько примеров в табл. 11.3.

**Таблица 11.3.** Сценарии результатов тестирования

Сценарий	Состояние метода	Действие теста	Результат
<b>TP</b>	В методе есть ошибка	Модульный тест запускается и завершается неудачей, указывая на ошибку	Тест правильно обнаруживает ошибку
<b>TN</b>	Метод не содержит ошибок	Модульный тест запускается и успешно проходит	Тест правильно подтверждает отсутствие ошибок
<b>FP</b>	Метод не содержит ошибок	Модульный тест запускается и завершается неудачей, ошибочно указывая на ошибку	Тест ошибочно сообщает о дефекте
<b>FN</b>	В методе есть ошибка	Модульный тест запускается и успешно проходит, не выявляя ошибку	Тест не обнаруживает реальный дефект

## Тестовые дублеры, имитации и заглушки

*Тестовый дублер* — обобщенный термин для любой фальшивой зависимости, используемой в тестах. Тестовые дублеры заменяют реальные зависимости, настройка которых может быть сложной или нестабильной.

Существует несколько основных видов дублеров. Чаще всего встречаются имитации и заглушки.

- *Имитации (mock-объекты)* используются для имитации исходящих взаимодействий. Например, тест может обращаться к такому объекту, имитирующему отправку электронного письма при регистрации пользователя. Такие взаимодействия могут менять состояние внешних систем. Имитации обычно создаются с помощью специализированных мок-фреймворков. Если они пишутся вручную, их иногда называют *шипионами (spy)*.
- *Заглушки (stub)* используются для моделирования входящих взаимодействий. Например, тест может обратиться к заглушке зависимости, возвращающей информацию о товаре из базы данных, при этом не изменяя состояние внешней системы. Если заглушка создается еще до появления настоящей зависимости (например, при разработке через тестирование), ее называют *фальшивкой (fake)*. Если заглушка представляет собой простую подстановку значения без влияния на поведение теста, ее называют *пустышкой (dumpty)*.



Разделение имитаций и заглушек связано с принципом разделения команд и запросов (Command Query Separation, CQS). Согласно этому принципу, каждый метод должен быть либо командой (выполнять действие и потенциально изменять состояние), либо запросом (возвращать данные без изменения состояния). Имитации применяются для тестирования команд, а заглушки — для тестирования запросов.

## Формирование подхода к тестированию

Чтобы получить максимальную отдачу от тестирования, важно сформировать правильный подход к нему. Продуманная стратегия тестирования помогает выявлять и устранять ошибки на ранних этапах разработки, а также позволяет убедиться, что проекты соответствуют ожиданиям пользователей по производительности, безопасности и удобству использования. Программируя на платформе .NET, вкладывая время в изучение и применение различных методологий тестирования, вы сможете создавать более надежные, эффективные и удобные приложения.

Одна из стратегий, которую стоит рассмотреть, называется *разработкой через тестирование* (*Test-Driven Development, TDD*). При таком подходе тесты пишутся до написания кода. Это способствует улучшению архитектуры приложения и упрощает его сопровождение. Однако важно учитывать возможные компромиссы.

Давайте подробнее разберем преимущества и недостатки разработки через тестирование.

## Преимущества и недостатки разработки через тестирование

Разработка через тестирование (*Test-Driven Development, TDD*) — это подход к разработке ПО, при котором основное внимание уделяется написанию тестов до реализации функциональности. Такой процесс побуждает заранее продумывать архитектуру и требования к коду, что способствует созданию более качественного, надежного и удобного в сопровождении ПО с самого начала.

## Основные принципы разработки через тестирование

Подход TDD строится вокруг простого цикла, который часто описывают как «красный — зеленый — рефакторинг». Этот процесс включает три шага.

- Красный.** Напишите тест для следующего фрагмента функциональности, который вы планируете добавить. Тест должен завершиться неудачей, так как соответствующая реализация еще отсутствует, а метод может выбрасывать исключение `NotImplementedException`. Этот шаг помогает убедиться, что тест действительно проверяет нужную функциональность.

2. **Зеленый.** Напишите минимальный объем кода, необходимый для прохождения теста. Такой подход поощряет простоту реализации и позволяет сосредоточиться только на нужной функциональности.
3. **Рефакторинг.** Приведите новый код в порядок, чтобы он хорошо сочетался с остальной кодовой базой, соответствовал передовым практикам и оставался читаемым и поддерживаемым. После рефакторинга все тесты должны продолжать успешно выполняться — это подтверждает, что изменения не нарушили существующую функциональность.

## Преимущества разработки через тестирование

Основные преимущества TDD можно сформулировать так.

- **Более высокое качество кода.** Применение TDD способствует созданию лучше спроектированной кодовой базы, более модульной и простой в сопровождении.
- **Улучшенная документация.** Написанные тесты служат документацией, четко описывая, что должна выполнять каждая часть приложения.
- **Снижение стоимости ошибок.** Обнаружение проблем на ранних этапах разработки помогает существенно сократить затраты на исправление ошибок в дальнейшем.
- **Упрощение внесения изменений.** Наличие полного набора тестов позволяет безопасно добавлять новые функции и вносить изменения в код без риска случайно нарушить работу других частей приложения.

## Недостатки разработки через тестирование

Рассмотрим основные сложности и особенности, связанные с применением TDD.

- **Сложности внедрения в команду.** Работа по принципам TDD требует изменения привычного подхода к разработке. Переход к такому стилю может занять время и потребовать дополнительных усилий для полной интеграции в рабочий процесс.
- **Замедление разработки на начальном этапе.** Написание тестов перед кодом сначала замедляет процесс, особенно пока команда привыкает к циклу «красный — зеленый — рефакторинг». Со временем это компенсируется за счет сокращения времени на отладку и исправление ошибок, но объяснить такую перспективу не всегда легко, особенно руководству без технической подготовки.
- **Накладные расходы на сопровождение тестов.** Сам набор тестов требует регулярной поддержки и актуализации по мере развития проекта. Это приводит к дополнительным затратам, которые нужно воспринимать как инвестицию в поддержание качества кода.

В завершение раздела о TDD мы рассмотрим несколько рекомендуемых практик, которые помогут использовать этот подход максимально эффективно.

## Рекомендуемые приемы разработки через тестирование

Чтобы внедрять TDD эффективно, стоит придерживаться следующих передовых методов.

- **Начинайте с простых тестов и функциональности,** постепенно переходя к более сложным сценариям. Это поможет лучше понять процесс TDD и не перегружать себя на старте. TDD особенно хорошо подходит для модульного тестирования, где цикл «красный — зеленый — рефакторинг» легко применять. В случае интеграционного тестирования или сложных функций такой процесс становится менее очевидным.
- **Фокусируйтесь на одном teste за раз.** Это помогает сократить цикл разработки и держать внимание на реализации одной конкретной части функциональности.
- **Регулярно выполняйте рефакторинг кода и тестов.** Постоянная работа над улучшением кода и тестов повышает их читаемость, эффективность и облегчает поддержку проекта в будущем.
- **Комбинируйте TDD с парным программированием и обзорами кода.** Совместная работа усиливает преимущества TDD, способствует распространению знаний о проекте и повышает общее качество кода и тестов.



Подробнее о парном программировании и обзорах кода мы поговорим в главе 19. Но если кратко, парное программирование — это практика совместной работы двух разработчиков за одним компьютером, где один из них пишет код («водитель»), а другой — внимательно следит за процессом и комментирует каждую строку («штурман»). Обзор кода — это процесс, при котором коллеги проверяют код друг друга, что помогает обнаруживать ошибки, повышать качество и согласованность кодовой базы, а также способствует распространению знаний в команде.

TDD — это методология разработки ПО, которая при правильном применении помогает создавать более качественные проекты, способствует продуманному проектированию и приводит к созданию кодовой базы, удобной для поддержки и расширения. Метод требует предварительных инвестиций времени и изменений в культуре разработки, но потенциальные долгосрочные преимущества в виде повышения качества кода и сокращения времени на отладку оказываются очень значительными.

Теперь перейдем к практическим примерам модульного тестирования — в частности, с использованием популярного пакета xUnit.

## Модульное тестирование с помощью xUnit

Во вводном разделе мы познакомились с основами модульного тестирования. Теперь давайте углубимся в детали. Один из самых спорных вопросов в модульном тестировании — это степень изоляции тестов. Разберем этот момент подробнее.

### Насколько изолированными должны быть модульные тесты

В этом вопросе существуют две основные точки зрения.

- **Традиционный подход** предполагает, что модульные тесты не изолируются от зависимостей. В этом случае тестируются как тестируемый метод (MUT), так и его зависимости. Проблема может возникнуть, если ошибка находится в одной из зависимостей — это усложняет поиск причины сбоя, поскольку увеличивается количество возможных источников ошибки. Тем не менее для самих зависимостей также должны существовать отдельные тесты. При этом каждый тест все равно должен быть изолирован от других тестов, чтобы они не влияли друг на друга.
- **Современный подход** ориентирован на максимальную изоляцию тестируемого метода от его зависимостей. Зависимости заменяются тестовыми дублерами, что устраняет их влияние на поведение MUT и позволяет фокусироваться исключительно на тестировании самого метода.

### Схема именования методов модульного тестирования

Некоторые специалисты предлагают использовать стандартизированную схему именования тестовых методов, например [Имя\_метода]\_[Сценарий]\_[Результат], где три части означают:

- **Имя\_метода** — литеральное имя тестируемого метода (MUT). Например, `CheckOut` для метода оформления заказа в корзине покупок;
- **Сценарий** — краткое описание ситуации, которая проверяется. Например, «в корзине есть товар, которого нет в наличии»;
- **Результат** — ожидаемый исход. Например, «проверка прошла успешно» или «проверка завершилась с ошибкой».

Однако такая машинная схема именования менее полезна, чем может показаться. Разработчики и без того склонны чрезмерно фокусироваться на деталях реализации, а строгая схема только усиливает эту тенденцию.

Лучше использовать простые фразы на естественном языке. Такой подход более гибок, а названия тестов понятны не только разработчикам, но и тем, кто хорошо знает бизнес-логику приложения. Загадочные громоздкие имена тестов создают лишнюю когнитивную нагрузку — даже для самих разработчиков. **Будьте проще!**

Ниже приведены три примера названий тестов, в которых используются простые фразы на понятном языке:

- `Checkout_fails_when_item_is_out_of_stock` (Оформление заказа не удается, если товар отсутствует на складе);
- `User_receives_confirmation_email_after_successful_registration` (Пользователь получает письмо с подтверждением после успешной регистрации);
- `Login_is_blocked_after_three_failed_attempts` (Вход в систему блокируется после трех неудачных попыток).

Такие названия четко описывают суть теста и остаются понятными как для технических специалистов, так и для участников проекта, знакомых с бизнес-логикой приложения.



**Дельный совет:** именуйте методы тестирования так, словно объясняете сценарий человеку без технической подготовки, но хорошо знакомому с бизнес-логикой. Чтобы повысить читаемость, используйте символы подчеркивания между словами.

## Модульное тестирование с помощью xUnit

xUnit.net, часто называемый просто xUnit, — это популярный фреймворк для модульного тестирования в экосистеме .NET. Он был специально разработан для устранения некоторых ограничений, присущих более ранним фреймворкам тестирования, таким как NUnit и MSTest.

Даже сотрудники компании Microsoft отдают предпочтение xUnit вместо использования MSTest. Например, команда ASP.NET Core применяет xUnit в своих проектах — это можно увидеть на странице проекта по адресу [github.com/dotnet/aspnetcore/tree/main/src/Testing/src/xunit](https://github.com/dotnet/aspnetcore/tree/main/src/Testing/src/xunit).



Если вы уже знакомы с другими фреймворками тестирования, можно найти сравнительные таблицы на сайте [xunit.net/docs/comparisons](https://xunit.net/docs/comparisons).

Среди основных достоинств xUnit можно выделить следующие.

- **Открытый исходный код и активное сообщество.** За развитием xUnit стоят сильная команда разработчиков и большое сообщество. Ввиду этого увели-

чивается вероятность, что фреймворк будет поддерживать последние возможности .NET и лучшие практики. Для xUnit доступно множество учебных материалов, руководств и сторонних расширений.

- **Упрощенная и расширяемая архитектура.** По сравнению со старыми фреймворками xUnit предлагает более гибкий подход. Он поощряет использование пользовательских шаблонов тестирования и минимизирует зависимость от методов установки и разрушения, что способствует написанию более чистого тестового кода.
- **Читабельный тестовый код с помощью атрибутов .NET.** В xUnit тесты описываются с использованием стандартных атрибутов. Для обычных тестов применяется атрибут `[Fact]`, а для параметризованных — `[Theory]` вместе с `[InlineData]`, `[ClassData]` или `[MemberData]`. Благодаря этому допустимо тестирование, управляемое данными, которое охватывает множество сценариев в одном методе и повышает тщательность проверки при минимальных затратах усилий.
- **Богатая библиотека утверждений.** xUnit включает широкую коллекцию готовых утверждений для проверки различных условий без необходимости писать собственный код проверок. При желании можно дополнительно использовать библиотеки вроде FluentAssertions для более выразительного и читаемого описания ожиданий.
- **Поддержка параллельного выполнения тестов.** По умолчанию xUnit выполняет тесты параллельно внутри одной коллекции, что позволяет существенно сократить общее время тестирования — особенно в средах непрерывной интеграции, где скорость важна. Если тесты выполняются на виртуальном выделенном сервере (Virtual Private Server, VPS) с ограниченным объемом памяти, производительность может снижаться: серверу будет сложнее обрабатывать большие объемы данных и поддерживать одновременную работу нескольких процессов. В таких ситуациях стоит рассмотреть возможность отключения параллельного выполнения тестов, чтобы сохранить стабильную работу системы. Небольшие экземпляры VPS часто используют в качестве доступных сред для тестирования.
- **Точный контроль над жизненным циклом тестов.** В xUnit управление жизненным циклом тестов осуществляется с помощью шаблонов конструктора и деструктора, интерфейса `IDisposable`, а также атрибута `[BeforeAfterTest-Attribute]`, что дает разработчику полный контроль над инициализацией и очисткой состояния тестов.

Как уже отмечалось, xUnit активно использует атрибуты .NET для определения структуры и поведения тестов. Эти атрибуты играют ключевую роль в организации тестового набора, настройке поведения тестов и управлении тестовыми данными. Давайте рассмотрим их подробнее.

## Распространенные атрибуты xUnit

Наиболее часто используемые атрибуты xUnit с примерами и краткими пояснениями приведены в табл. 11.4.

**Таблица 11.4.** Распространенные атрибуты xUnit

Пример атрибута	Описание
[Fact] public void TestAdding2and2()	Атрибут [Fact] объявляет метод тестирования без параметров, который выполняется один раз программой запуска тестов
[Theory] [...] public void TestAdding(double expected, double number1, double number2)	Атрибут [Theory] объявляет метод тестирования с параметрами, который выполняется несколько раз с разными наборами данных. Его следует использовать вместе с атрибутами, предоставляющими данные, — например, [InlineData], [ClassData] или [MemberData]
[InlineData(4, 2, 2)] [InlineData(5, 2, 3)]	Атрибут [InlineData] предоставляет фиксированные значения параметров для метода, помеченного [Theory]. Данные передаются в порядке, соответствующем параметрам метода
[ClassData(typeof(AddingNumbersData))]	Атрибут [ClassData] предоставляет перечисляемые значения параметров для метода с атрибутом [Theory]. Класс данных должен реализовать <code>IEnumerable&lt;object[]&gt;</code> либо быть производным от <code>TheoryData</code> для работы с сильно типизированными данными
[MemberData(nameof(GetTestData))]	Атрибут [MemberData] предоставляет перечисляемые значения параметров для метода с атрибутом [Theory]. Данные берутся из статического метода, возвращающего <code>IEnumerable&lt;object[]&gt;</code>
[Trait("Feature", "Shopping Cart")]	Атрибут [Trait] позволяет добавить метаданные к тесту, распределяя тесты по категориям для удобной фильтрации при запуске

Атрибуты [Fact] и [Theory] позволяют не только определить структуру теста, но и изменить его отображаемое название в результатах или установить тайм-аут выполнения — в миллисекундах. Например:

```
[Fact(Timeout = 3000)] // Тест завершится через 3 секунды
```

Если необходимо временно пропустить тест, можно воспользоваться параметром `Skip`, указав причину пропуска в текстовом виде:

```
[Fact(Skip = "Skipping this test for now.")]
```

Теперь перейдем к практике и напишем немного кода, чтобы познакомиться с возможностями xUnit поближе.

## Создание тестируемой системы (SUT)

Начнем с подготовки проекта: создадим библиотеку классов и добавим в нее тестируемый класс (SUT) с методом (MUT), который будет содержать преднамеренную ошибку.

1. В редакторе кода создайте проект **Class Library** (Библиотека классов) / `classlib`, присвойте ему имя **BusinessLogic** и добавьте в решение **Chapter11**.
2. В файле **BusinessLogic.csproj** проекта настройте обработку предупреждений как ошибок и статически и глобально импортируйте класс **Console**.
3. Переименуйте файл **Class1.cs** в **Calculator.cs**.
4. В файле **Calculator.cs** определите класс с методом сложения двух вещественных чисел. В методе допустите преднамеренную ошибку:

```
namespace Packt.Shared;

// Тестируемая система (SUT)
public class Calculator
{
    // Тестируемый метод (MUT)
    public double Add(double number1, double number2)
    {
        return number1 * number2; // Преднамеренная ошибка!
    }
}
```

5. Выполните сборку проекта **BusinessLogic**.
6. В редакторе кода создайте новый C#-проект **xUnit Test Project** (Тестовый проект xUnit) / `xunit`, присвойте ему имя **BusinessLogicUnitTests** и добавьте в решение **Chapter11**. Для этого в терминале, находясь в папке **Chapter11**, выполните команды:

```
dotnet new xunit -o BusinessLogicUnitTests
dotnet sln add BusinessLogicUnitTests
```

7. В проекте **BusinessLogicUnitTests** добавьте ссылку на проект **BusinessLogic**:

```
<ItemGroup>
    <ProjectReference Include=".\\BusinessLogic\\BusinessLogic.csproj" />
</ItemGroup>
```



Путь к проекту можно указывать как через прямую (/), так и через обратную (\) косую черту. Пакет .NET SDK автоматически обработает путь в зависимости от операционной системы.

8. Выполните сборку проекта **BusinessLogicUnitTests**.

## Создание простых модульных тестов

Качественный модульный тест обычно строится по принципу AAA (Arrange – Act – Assert), что можно перевести как «Подготовка – Действие – Проверка». Эти три части можно описать так.

- **Подготовка (Arrange).** На этом этапе объявляются и инициализируются переменные, необходимые для теста: входные данные, ожидаемый результат и экземпляр тестируемой системы (SUT). Обычно это означает создание объекта тестируемого типа.
- **Действие (Act).** Здесь выполняется сам тестируемый модуль. В нашем случае это вызов метода (MUT) тестируемого класса. Если в разделе действия выполняется несколько операций с последующими утверждениями, скорее всего, вы пишете интеграционный тест, а не модульный. В такой ситуации тест стоит переработать, разделив его на отдельные операции, фокусирующиеся на одном действии.
- **Проверка (Assert).** На этом этапе выполняются проверки результата действия. Утверждение – это ожидаемое условие, которое должно быть истинным. Если оно оказывается ложным, тест считается неудачным. Например, при сложении 2 и 2 мы ожидаем получить результат 4. Если в тесте содержится несколько действий с отдельными проверками, скорее всего, это признак интеграционного теста.



**Дельный совет:** даже если тестируемый метод один, рекомендуется написать несколько тестов с разными входными данными. Так вы сможете проверить работу метода в различных сценариях и повысить надежность тестового покрытия.

Теперь создадим несколько модульных тестов для класса `Calculator`.

1. Переименуйте файл `UnitTest1.cs` в `CalculatorUnitTests.cs`, а затем откройте его.

Если вы работаете в редакторе кода Visual Studio Code, переименование класса нужно будет выполнить вручную в исходном коде. Visual Studio сама предложит переименовать класс при переименовании файла.

2. В файле `CalculatorUnitTests.cs` подключите пространство имен `Packt.Shared`, а затем измените код класса `CalculatorUnitTests`, добавив в него два тестовых метода: один для сложения 2 и 2, другой – для сложения 2 и 3:

```
using Packt.Shared; // Подключение класса Calculator

namespace BusinessLogicUnitTests;

public class CalculatorUnitTests
{
    [Fact]
    public void TestAdding2And2()
    {
```

```

// Подготовка: установка входных данных, ожидаемого результата
// и создание экземпляра SUT.
double number1 = 2;
double number2 = 2;
double expected = 4;
Calculator sut = new();

// Действие: вызов тестируемого метода (MUT).
double actual = sut.Add(number1, number2);

// Проверка: создание утверждений для сравнения ожидаемого
// и фактического результата.
Assert.Equal(expected, actual);
}

[Fact]
public void TestAdding2And3()
{
    // Подготовка: установка входных данных, ожидаемого результата
    // и создание экземпляра SUT.
    double number1 = 2;
    double number2 = 3;
    double expected = 5;
    Calculator sut = new();

    // Действие: вызов тестируемого метода (MUT).
    double actual = sut.Add(number1, number2);

    // Проверка: создание утверждений для сравнения ожидаемого
    // и фактического результата.
    Assert.Equal(expected, actual);
}
}

```



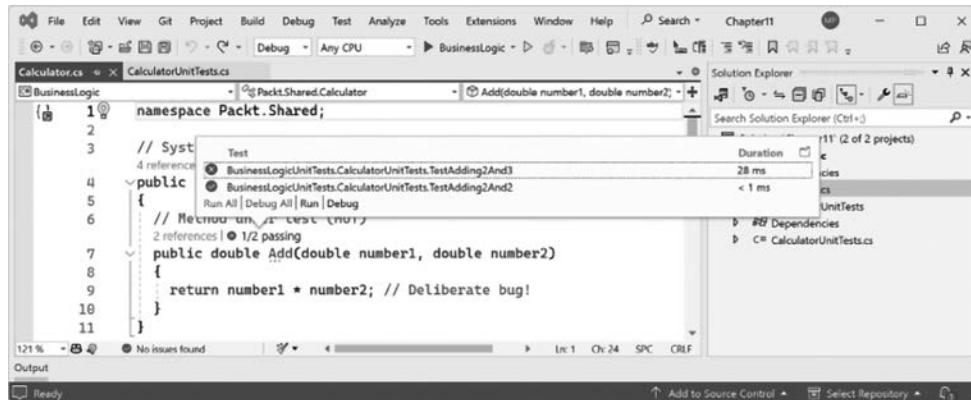
В Visual Studio до сих пор используется старый шаблон элемента проекта, в котором применяется вложенное пространство имен. В приведенных примерах показан современный стиль, используемый в командах dotnet new и редакторе JetBrains Rider, где применяется файловое пространство имен.



**Дельный совет:** присваивайте переменной для тестируемой системы (SUT) имя sut, чтобы она явно отличалась от любых других зависимостей. Рекомендуется использовать директиву #region для выделения трех частей теста — подготовки, действия и проверки — и сделать их сворачиваемыми. Минимальное требование — разделять эти части пустыми строками. Даже без комментариев с пометками Arrange, Act и Assert единообразное оформление тестов заметно улучшает их читаемость.

3. Выполните сборку проекта `BusinessLogicUnitTests`.
4. Запустите модульные тесты и проанализируйте результаты.
  - Если вы работаете в Visual Studio, выберите команду меню `Test ▶ Run All Tests` (`Тест ▶ Запуск всех тестов`), а затем — команду `Test ▶ Test Explorer` (`Тест ▶ Обозреватель тестов`). На панели `Test Explorer` (`Обозреватель тестов`) разверните дерево тестов. Вы увидите, что оба теста были выполнены: один

прошел успешно, один завершился неудачей. В проекте `BusinessLogic`, в классе `Calculator`, обратите внимание на индикатор **1/2 passing**, показывающий, что один из двух тестов выполнен успешно (рис. 11.1).



**Рис. 11.1.** Количество успешных тестов и общее количество тестов для MUT

- Если вы работаете в Visual Studio Code, выберите команду меню **View ▶ Testing** (Вид ▶ Тестирование). На открывшейся панели **Testing** (Тестирование) существует мини-панель с кнопками **Refresh Tests** (Обновить тесты), **Run Tests** (Запустить тесты), **Debug Tests** (Отладить тесты) и др. Наведите указатель мыши на класс `CalculatorUnitTests` и нажмите кнопку **Run Tests** (Запуск тестов) (значок черного треугольника). Затем перейдите на вкладку **Test Results** (Результаты тестов). На ней показано, что два теста были выполнены: один прошел успешно, один завершился неудачей.
- Если вы используете интерфейс командной строки (CLI), перейдите в терминале в папку `Chapter11` и запустите тестовый проект командой `dotnet test`. В выводе команды будет показано, что оба теста были выполнены, но один из них завершился неудачей:

```
Restore complete (0.5s)
  BusinessLogic succeeded (0.2s) → BusinessLogic\bin\Debug\net8.0\
  BusinessLogic.dll
    BusinessLogicUnitTests succeeded (0.1s) → BusinessLogicUnitTests\bin\
  Debug\net8.0\BusinessLogicUnitTests.dll
      BusinessLogicUnitTests failed with errors (0.0s)
        C:\tools-skills-net8\Chapter11\BusinessLogicUnitTests\
  CalculatorUnitTests.cs(36): error VTEST1: (BusinessLogicUnitTests.
  CalculatorUnitTests.TestAdding2And3) BusinessLogicUnitTests.
  CalculatorUnitTests.TestAdding2And3() Assert.Equal() Failure:
  Values differ Expected: 5 Actual: 6 [C:\tools-skills-net8\Chapter11\
  BusinessLogicUnitTests\BusinessLogicUnitTests.csproj]

Build failed with errors in 1.7s
```

На данном этапе мы могли бы сразу исправить ошибку в методе, но специально оставим ее, чтобы далее рассмотреть более эффективные способы написания тестов с использованием параметров.

## Методы тестирования с параметрами

На данный момент у нас есть два теста, которые выполняются независимо друг от друга, хотя фактически проверяют один и тот же метод (MUT), но с разными значениями параметров. Когда параметров немного, такой подход допустим — он остается простым и наглядным. Однако с увеличением числа проверяемых сценариев становится удобнее написать один метод тестирования и передавать в него различные значения параметров с помощью специальных приемов.

Существует несколько способов организации тестирования с параметрами.

- Пометить метод тестирования атрибутом [`InlineData`] и передать фиксированные значения параметров.
- Пометить метод тестирования атрибутом [`ClassData`] и использовать класс, предоставляющий коллекцию данных через `IEnumerable`.
- Пометить метод тестирования атрибутом [`ClassData`] и указать класс, производный от `TheoryData<T1, T2, ...>`. В его конструкторе вызвать унаследованный метод `Add`, чтобы добавить наборы параметров и ожидаемых значений.
- Пометить метод тестирования атрибутом [`MemberData`] и ссылаться на метод, который возвращает коллекцию данных.

### Теоретическое тестирование методов с помощью атрибута `InlineData`

Рассмотрим, как можно использовать атрибут [`InlineData`] для тестирования одного метода с разными входными данными.

1. В файле `CalculatorUnitTests.cs` проекта `BusinessLogicUnitTests` добавьте код для определения тестового метода с параметрами:

```
[Theory]
[InlineData(4, 2, 2)]
[InlineData(5, 2, 3)]
public void TestAdding(double expected,
    double number1, double number2)
{
    // Подготовка: создание экземпляра SUT
    Calculator sut = new();

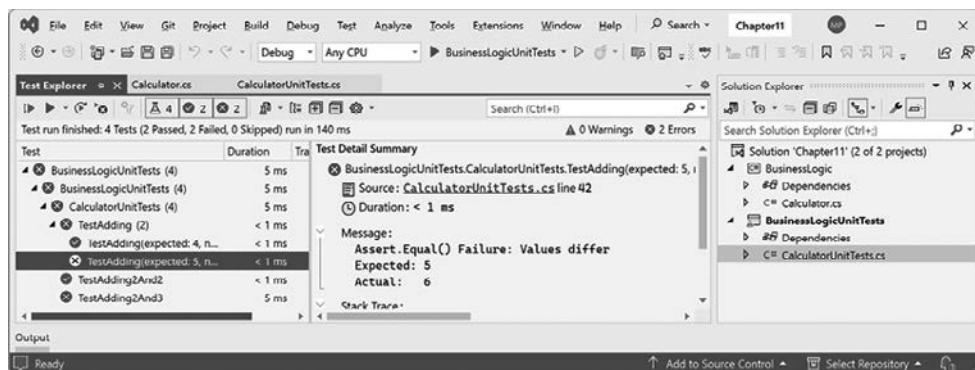
    // Действие: вызов MUT
    double actual = sut.Add(number1, number2);

    // Проверка: сравнение ожидаемого и фактического результата
    Assert.Equal(expected, actual);
}
```



Атрибут [InlineData] принимает набор параметров через механизм params. Это означает, что можно передавать любые лiteralные значения или переменные любого типа, разделенные запятыми. Внутри тестового метода они будут интерпретироваться как массив System.Object[].

2. Выполните сборку проекта BusinessLogicUnitTests.
3. Запустите модульный тест и изучите результаты. В качестве примера: если вы работаете в Visual Studio, то на панели Test Explorer (Обозреватель тестов) будет отображен метод TestAdding с двумя наборами переданных параметров. Один из тестов завершится успешно, а другой — с ошибкой, как показано на рис. 11.2.



**Рис. 11.2.** Выполнение теста с параметрами, один из которых завершился неудачно

## Теоретическое тестирование методов с помощью атрибута ClassData

Теперь воспользуемся отдельным классом для предоставления значений параметров.

1. В проекте BusinessLogicUnitTests создайте файл класса AddingNumbersData.cs.
2. В этом файле определите класс, реализующий интерфейс I Enumerable<object[]>. Этот класс должен возвращать последовательность массивов объектов — каждый массив представляет набор параметров для тестового метода:

```
using System.Collections; // Подключение интерфейса IEnumerable<T>

namespace BusinessLogicUnitTests;

internal class AddingNumbersData : IEnumerable<object[]>
{
    public IEnumerator<object[]> GetEnumerator()
    {
        // Проверка сложения 2 и 2 с результатом 4
        yield return new object[] { 4, 2, 2 };
    }
}
```

```

    // Проверка сложения 2 и 3 с результатом 5
    yield return new object[] { 5, 2, 3 };
}

// Необобщенная версия интерфейса вызывает обобщенную реализацию
IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}

```



**Дельный совет:** вместо жесткого кодирования всех наборов параметров через `yield return` можно считывать данные из файла или базы данных и кэшировать их. Так вы сможете изменять или добавлять параметры без необходимости перекомпиляции проекта тестов при каждом обновлении данных.

3. В файле `CalculatorUnitTests.cs` закомментируйте атрибуты `[InlineData]` в тестовом методе и добавьте атрибут `[ClassData]`, ссылаясь на только что созданный класс:

```

[Theory]
// [InlineData(4, 2, 2)]
// [InlineData(5, 2, 3)]
[ClassData(typeof(AddingNumbersData))]
public void TestAdding(double expected,
    double number1, double number2)

```

4. Выполните сборку проекта `BusinessLogicUnitTests`.
5. Запустите модульные тесты. Результаты должны совпасть с теми, которые были получены ранее при использовании `[InlineData]`.

## Теоретическое тестирование методов с помощью сильно типизированного атрибута `ClassData`

Теперь используем сильно типизированный класс для передачи параметров в тестовый метод.

1. В проекте `BusinessLogicUnitTests` создайте файл класса `AddingNumbersDataTyped.cs`.
2. В созданном файле определите класс, который наследуется от `TheoryData<T1, T2, T3>`, где `T1` — ожидаемое значение, а `T2` и `T3` — входные значения. Добавьте в коллекцию два набора параметров, которые будут передаваться тестовому методу:

```

namespace BusinessLogicUnitTests;

internal class AddingNumbersDataTyped : TheoryData<double, double,
    double>
{
    public AddingNumbersDataTyped()
    {
        // Проверка сложения 2 и 2 с результатом 4
        Add(4, 2, 2);
    }
}

```

```

    // Проверка сложения 2 и 3 с результатом 5
    Add(5, 2, 3);
}
}

```



Порядок указанных типов ( $T_1, T_2, T_3$ ) должен соответствовать порядку параметров в тестовом методе. В нашем примере сначала передается ожидаемое значение, затем два входных параметра. При желании можно изменить порядок, например, передавать ожидаемое значение последним — тогда типы будут распределены иначе:  $T_1$  и  $T_2$  для входов,  $T_3$  для результата.

3. В файле `CalculatorUnitTests.cs` закомментируйте ранее использованный атрибут `[ClassData]` и добавьте новый атрибут, ссылающийся на только что созданный класс:

```

[Theory]
// [InlineData(4, 2, 2)]
// [InlineData(5, 2, 3)]
// [ClassData(typeof(AddingNumbersData))]
[ClassData(typeof(AddingNumbersDataTyped))]
public void TestAdding(double expected,
    double number1, double number2)

```

4. Выполните сборку проекта `BusinessLogicUnitTests`.
5. Запустите модульные тесты. Результаты должны быть такими же, как и при использовании обычного атрибута `[ClassData]`.

## Теоретическое тестирование методов с помощью атрибута `MethodData`

Теперь для предоставления значений параметров воспользуемся методом.

1. В файле `CalculatorUnitTests.cs` создайте статический метод, который возвращает `IEnumerable<object[]>`. Метод должен возвращать последовательность массивов объектов, каждый из которых представляет набор параметров для тестируемого метода:

```

public static IEnumerable<object[]> GetTestData()
{
    // Проверка сложения 2 и 2 с результатом 4
    yield return new object[] { 4, 2, 2 };

    // Проверка сложения 2 и 3 с результатом 5
    yield return new object[] { 5, 2, 3 };
}

```



Метод должен быть статическим и возвращать `IEnumerable<object[]>`. Вместо инструкции `yield return` можно вернуть список (`List<object[]>`) или массив. К примеру, в версии не ниже C# 12 можно воспользоваться синтаксисом коллекции `return [ [4, 2, 2], [5, 2, 3] ];`.

2. В файле `CalculatorUnitTests.cs` закомментируйте атрибут `[ClassData]` и добавьте атрибут `[MemberData]`, ссылаясь на только что созданный метод:

```
[Theory]
// [InlineData(4, 2, 2)]
// [InlineData(5, 2, 3)]
// [ClassData(typeof(AddingNumbersData))]
[MemberData(memberName: nameof(GetTestData))]
public void TestAdding(double expected,
    double number1, double number2)
```

3. Выполните сборку проекта `BusinessLogicUnitTests`.  
4. Запустите модульные тесты. Результаты должны быть такими же, как и при использовании атрибута `[ClassData]`.

## Положительные и отрицательные результаты теста

Параметризованные тесты позволяют существенно сократить объем кода, который нужно написать для покрытия разных сценариев. Однако при этом может усложниться выявление важных различий между тестами.

Одни тесты проверяют положительные результаты — например, успешное оформление заказа покупателем. Другие тесты проверяют отрицательные результаты — например, невозможность оформления заказа из-за недостаточного количества товара на складе. Хотя технически все эти случаи можно тестировать в одном методе с разными параметрами, на практике рекомендуется разделять положительные и отрицательные сценарии в разные тестовые методы.



**Дельный совет:** храните положительные и отрицательные сценарии в отдельных методах тестирования с соответствующими наборами параметров. Если какой-то сценарий особенно сложный и его трудно понять, глядя только на значения параметров, создайте для него отдельный тестовый метод с атрибутом `[Fact]`.

Теперь рассмотрим предупреждающие признаки, которые могут указывать на проблемы в модульных тестах, а также способы выполнения дополнительного кода при подготовке и завершении тестирования.

## Предупреждающие признаки в модульных тестах

При написании модульных тестов стоит обращать внимание на следующие предупреждающие признаки.

- Если в тесте используются конструкции `if` или `switch`, это антипаттерн. Модульный тест не должен содержать разветвлений — скорее всего, вы пытаетесь проверить слишком много функций в одном тесте. Рекомендуется выделить каждое ветвление в отдельный тест.

- Секция подготовки (`Arrange`) обычно самая большая из трех частей теста. Иногда она содержит больше кода, чем секции действия (`Act`) и проверки (`Assert`), вместе взятые. Если подготовительная часть становится громоздкой, стоит выделить приватные вспомогательные методы и вызывать их внутри теста.
- Секция действия обычно должна содержать один вызов. Если для проверки одной операции требуется вызвать несколько методов, это может свидетельствовать о неудачном проектировании публичного API. Например, если для оформления покупки на сайте требуется сначала вызвать метод `ProcessOrder`, а затем отдельно метод `ReduceInventory`, стоит задуматься о перепроектировании. Такой случай хорошо иллюстрирует пользу подхода TDD, который помогает улучшить дизайн публичных API и сервисов еще на ранних этапах, пока изменения не стали сложными и затратными.
- В секции проверки может быть несколько утверждений, поскольку одна операция может иметь несколько выходных значений. Если необходимо проверить, например, результат типа `bool`, `int` и `string`, стоит рассмотреть возможность перепроектировать API так, чтобы метод возвращал запись, объединяющую все эти значения. Благодаря этому можно будет сделать одно целостное утверждение вместо нескольких отдельных.

## Просмотр результатов во время выполнения теста

По умолчанию в xUnit вывод, отправляемый через `Console.WriteLine`, не отображается во время выполнения тестов. Это сделано намеренно, поскольку тесты обычно выполняются параллельно и вывод в консоль мог бы стать хаотичным. Для записи вывода в тестах xUnit следует использовать интерфейс `ITestOutputHelper`. Его экземпляр внедряется через конструктор тестового класса, обеспечивая привязку вывода именно к тому тесту, который его сгенерировал. Такой подход позволяет сохранять вывод организованным и избежать путаницы между результатами разных тестов.

Разработчику может понадобиться записывать вывод во время выполнения тестов по нескольким причинам.

- **Отладка.** Вывод данных во время теста помогает определить, на каком этапе возникает сбой. Просматривая значения переменных, последовательность выполнения или другие важные детали, разработчик быстрее поймет, что именно идет не так.
- **Промежуточные результаты.** Вывод промежуточных данных позволяет проследить, как изменяются значения в процессе тестирования. Это особенно полезно для проверки корректности работы тестируемого кода.
- **Проверка утверждений.** Тесты содержат утверждения о состоянии системы или данных. Вывод на экран можно использовать для дополнительной проверки соблюдений условий теста.

- **Длительные тесты.** При длительных тестах — например, связанных с производительностью, интеграцией или сквозным тестированием — вывод позволяет отслеживать ход выполнения. В таких случаях сообщения помогают убедиться, что тест действительно выполняется, и дают контрольные точки для оценки прогресса. Обратная связь в реальном времени особенно важна, когда тестирование занимает значительное время.
- **Сложная настройка и завершение работы.** Когда тесты включают сложные процессы настройки и освобождения ресурсов, вывод помогает подтвердить, что все этапы выполняются корректно. Это особенно важно при взаимодействии с внешними системами, базами данных или специфическими конфигурациями. Вывод позволяет убедиться, что ресурсы правильно выделяются и освобождаются, что критично для предотвращения утечек и обеспечения надежности тестов.
- **Документация и отчетность.** Вывод может служить дополнительной формой документации, фиксируя действия, предпринятые в процессе тестирования. Это полезно для аудита, проверки соответствия требованиям или просто для сохранения наглядной истории проведения тестов. Вывод можно сохранять и включать в отчеты о тестировании, предоставляя дополнительный контекст для заинтересованных сторон.
- **Понимание поведения тестов.** Иногда вывод помогает лучше понять, что именно делает тест и почему. Это особенно полезно для других разработчиков, просматривающих тесты, а также для новых членов команды, которые еще не знакомы с кодовой базой. Результаты выполнения тестов могут стать образовательным инструментом, показывая, как система ведет себя в различных условиях, и помогая быстрее освоить используемые подходы и фреймворки тестирования.

В следующем разделе мы рассмотрим пример записи вывода во время выполнения теста.

## Настройка и завершение работы

Перед запуском теста иногда требуется выполнить несколько операторов для подготовки состояния или данных, чтобы обеспечить известное начальное состояние системы. А после выполнения теста может понадобиться очистить данные или освободить ресурсы.

Такая необходимость чаще возникает при интеграционном тестировании, где есть внешние зависимости — например, файлы, сервисы или базы данных. Тем не менее иногда и модульные тесты требуют настройки и очистки, поэтому эти приемы рассматриваются уже сейчас, а более подробно они будут описаны в главе 12.

Если нужно выполнять подготовительные действия перед каждым тестом, их можно разместить в конструкторе тестового класса. Если требуется освобождать ресурсы после тестов, тестовый класс должен реализовать интерфейс `IDisposable`, а код очистки поместить в метод `Dispose`.

Посмотрим, как применить эту технику на примере тестирования класса калькулятора.

1. В файле `CalculatorUnitTests.cs` импортируйте пространство имен для использования интерфейса `ITestOutputHelper`, который применяется для вывода сообщений в тестах:

```
using Xunit.Abstractions; // Подключение интерфейса ITestOutputHelper
```

2. Добавьте конструктор в класс тестов для инициализации приватных полей — экземпляра SUT и объекта для записи вывода. Реализуйте интерфейс `IDisposable` для освобождения ресурсов. Добавьте вывод сообщений, чтобы видеть, когда вызывается конструктор, метод `Dispose` и сами тесты, как показано в следующем примере:

```
public class CalculatorUnitTests : IDisposable
{
    private readonly Calculator _sut;
    private readonly ITestOutputHelper _output;

    public CalculatorUnitTests(ITestOutputHelper output)
    {
        _sut = new();
        _output = output;

        _output.WriteLine("Constructor runs before each test.");
    }

    [Fact]
    public void TestAdding2And2()
    {
        _output.WriteLine($"Running {nameof(TestAdding2And2)}.");
        ...
    }

    [Fact]
    public void TestAdding2And3()
    {
        _output.WriteLine($"Running {nameof(TestAdding2And3)}.");
        ...
    }

    [Theory]
    // [InlineData(4, 2, 2)]
    // [InlineData(5, 2, 3)]
    // [ClassData(typeof(AddingNumbersData))]
    [MemberData(memberName: nameof(GetTestData))]
    public void TestAdding(double expected,
        double number1, double number2)
    {
        _output.WriteLine($"Running {nameof(TestAdding)}.");
        _output.WriteLine($" {nameof(number1)}: {number1}");
    }
}
```

```

    _output.WriteLine($" {nameof(number2)}: {number2}");
    _output.WriteLine($" {nameof(expected)}: {expected}");
    ...
}

...
public void Dispose()
{
    // При необходимости переменная _sut очищается

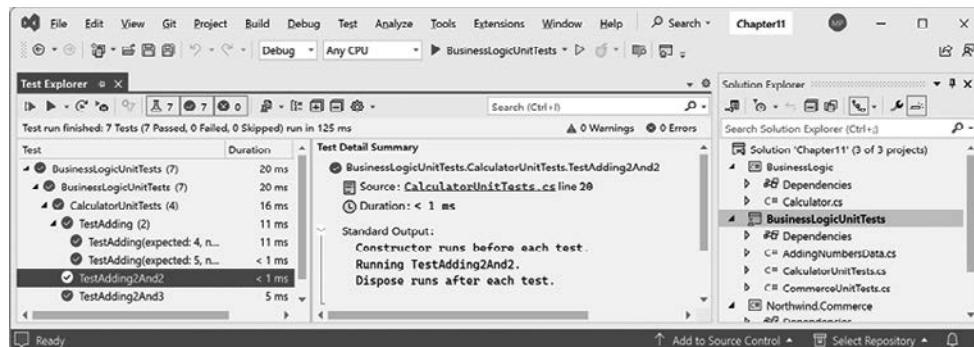
    _output.WriteLine("Dispose runs after each test.");
}
}

```

- В каждом методе тестирования используйте переменную `_sut` вместо `sut`, а выражения для создания экземпляра `Calculator` закомментируйте или удалите.
- Повторно запустите тесты. Если вы используете CLI, в терминале перейдите в папку проекта и введите команду. Таким образом вы сможете увидеть стандартный вывод для каждого теста:

```
dotnet test --logger "console;verbosity=detailed"
```

- Обратите внимание, что поведение тестов останется прежним, но теперь дополнительная информация в Visual Studio будет выводиться на панели **Test Detail Summary** (Краткое описание подробностей теста) в разделе **Standard Output** (Стандартный вывод) (рис. 11.3).



**Рис. 11.3.** Стандартный вывод на панели Test Detail Summary



Чтобы увидеть стандартный вывод, записанный с помощью интерфейса `ITestOutputHelper`, при выполнении команды `dotnet test`, используйте параметр командной строки `--logger "console;verbosity=detailed"`.

Хотя перемещение общего кода в одну локацию, например в конструктор, иногда и приемлемо, у такого подхода есть определенные недостатки.

- При чтении теста вам придется дополнительно изучать конструктор, чтобы понять, как инициализируются SUT и зависимости, используемые в teste.
- Конструктор инициализирует переменные-члены для всех тестов без разбора, что делает его хрупким и нарушает принцип единственной ответственности.

На практике лучше, когда каждый тест самостоятельно инициализирует все необходимое для своей работы. Так тест легче читать как одно целое, проще оценить его сложность, а сами тесты остаются полностью изолированными друг от друга. Во многих случаях стоит вообще отказаться от общих методов настройки и завершения работы.

## Управление фикстурами тестов

Распространено заблуждение, что тестовые фикстуры всегда нужно настраивать в конструкторе тестового класса. В результате тестовые методы могут вообще не содержать кода для фазы подготовки, поскольку вся подготовка будет спрятана в конструкторе. Такой подход действительно сокращает объем кода, но при этом усиливает связь между тестами и ухудшает их читаемость и изоляцию.

Сначала разберем практический пример, а потом улучшим его.

1. В редакторе кода создайте проект Class Library (Библиотека классов) / `classlib`, присвойте ему имя `Northwind.Commerce` и добавьте в решение `Chapter11`.
2. В файле `Northwind.Commerce.csproj` проекта настройте обработку предупреждений как ошибок.
3. Переименуйте файл `Class1.cs` в `Store.cs`.
4. В файле `Store.cs` определите класс с методами для управления запасами товаров, хранящихся в словаре, где ключом выступает идентификатор товара, а значением — количество единиц на складе:

```
namespace Northwind.Commerce;

public class Store
{
    // Ключ – ProductId, значение – Quantity
    public Dictionary<int, int> Inventory { get; } = new();

    public void AddInventory(int productId, int quantity)
    {
        // Добавление указанного количества товара
        if (Inventory.ContainsKey(productId))
            Inventory[productId] += quantity;
        else
            Inventory.Add(productId, quantity);
    }
}
```

```
public bool RemoveInventory(int productId, int quantity)
{
    if (Inventory[productId] >= quantity)
    {
        // Удаление указанного количества товара
        Inventory[productId] -= quantity;
        return true;
    }
    return false;
}
```

5. Создайте новый файл класса `Cart.cs`.
6. В файле `Cart.cs` определите класс для отслеживания товаров, которые покупатель добавил в корзину и в дальнейшем предполагает приобрести:

```
namespace Northwind.Commerce;

public class Cart
{
    private readonly Store _store;

    public Cart(Store store)
    {
        _store = store;
    }

    // Ключ – ProductId, значение – Quantity
    public Dictionary<int, int> Items { get; } = new();

    public void AddItems(int productId, int quantity)
    {
        // Добавление указанного количества товара
        if (Items.ContainsKey(productId))
            Items[productId] += quantity;
        else
            Items.Add(productId, quantity);
    }

    public void RemoveItems(int productId, int quantity)
    {
        // Удаление указанного количества товара
        Items[productId] -= quantity;
    }

    public bool Checkout()
    {
        // Обработка оформления заказа и прием оплаты
        foreach (var item in Items)
        {
            if (! _store.RemoveInventory(item.Key, item.Value))
            {
                // Ошибка при удалении товара, если его нет в наличии
            }
        }
    }
}
```

```
        return false;
    }
}
return true;
}
}
```

7. Выполните сборку проекта `Northwind.Commerce`.
8. В проекте `BusinessLogicUnitTests` добавьте ссылку на библиотеку классов `Northwind.Commerce`:

```
<ItemGroup>
  <ProjectReference Include="..\BusinessLogic\BusinessLogic.csproj" />
  <ProjectReference Include="..\Northwind.Commerce\Northwind.Commerce.csproj" />
</ItemGroup>
```

9. В проекте `BusinessLogicUnitTests` создайте файл класса `CommerceUnitTests.cs`.
10. В файле `CommerceUnitTests.cs` определите класс с тестами, в котором вместо конструктора используется фабричный метод для создания тестовых фикстур, таких как экземпляр магазина:

```
using Northwind.Commerce; // Подключение классов Store и Cart

namespace BusinessLogicUnitTests;

public class CommerceUnitTests
{
    // Фабричный метод для создания тестовых фикстур
    private Store CreateStore(int productId, int initialInventory)
    {
        Store store = new();
        store.AddInventory(productId, initialInventory);
        return store;
    }

    [Fact]
    public void Checkout_ShouldReduceInventoryLevel()
    {
        #region Arrange
        int productId = 1;
        int initialInventory = 10;
        Store store = CreateStore(productId, initialInventory);
        Cart sut = new(store);
        int quantityToBuy = 5;
        #endregion

        #region Act
        sut.AddItems(productId, quantityToBuy);
        bool success = sut.Checkout();
        #endregion

        #region Assert
        Assert.True(success);
    }
}
```

```

int updatedInventory = store.Inventory[productId];
Assert.Equal(initialInventory - quantityToBuy, updatedInventory);
#endregion
}

[Fact]
public void Checkout_ShouldFailWhenLowInventory()
{
    // Подготовка
    int productId = 1;
    int initialInventory = 10;
    Store store = CreateStore(productId, initialInventory);
    Cart sut = new(store);
    int quantityToBuy = 15;

    // Действие
    sut.AddItems(productId, quantityToBuy);
    bool success = sut.Checkout();

    // Проверка
    Assert.False(success);
    int updatedInventory = store.Inventory[productId];
    Assert.Equal(initialInventory, updatedInventory);
}
}

```



Чтобы продемонстрировать разные способы документирования тестов, в приведенном примере один тест снабжен простыми комментариями, а другой оформлен с использованием сворачиваемых блоков `#region`.

11. Запустите тесты и убедитесь, что все они выполняются успешно.



**Дельный совет:** изменение одного теста или конструктора не должно оказывать влияние на выполнение других тестов. Вместо приватных полей для хранения общего состояния (например, для SUT и фикстур типа `Store`) лучше определить фабричные методы. Они упрощают код в секции подготовки, сохраняя при этом явный контекст создания зависимостей.

## Имитация сущностей в тестах

Создание имитаций в модульных тестах — это техника, которая помогает изолировать тестируемый блок кода, заменяя его зависимости контролируемыми, заранее настроенными тестовыми дублерами, известными как *имитации* (или *mock-объекты*). Благодаря этому можно сосредоточиться на тестировании конкретной функциональности блока кода без влияния со стороны его зависимостей.

В процессе работы с имитациями в тестах важно понимать несколько ключевых понятий.

- **Изоляция.** Имитации помогают изолировать тестируемый модуль, чтобы тест сосредоточивался только на его поведении, без влияния со стороны

зависимостей. Благодаря этому можно создать контролируемую среду, где легко управлять входными и выходными данными.

- **Тестовые дублеры.** Для тестов вместо настоящих зависимостей используют разные виды дублеров.
  - **Имитации (mock-объекты)** подменяют настоящие объекты и настраивают так, чтобы возвращать определенные значения или выполнять заданные действия.
  - **Фальшивки (fakes)** — упрощенные реализации интерфейсов или классов. Они работают, но обычно проще устроены и менее гибкие, чем имитации.
  - **Заглушки (stubs)** возвращают фиксированные ответы на вызовы, не реализуя полное поведение зависимости.
  - **Шпионы (spies)** похожи на имитации, но дополнительно фиксируют, как именно их вызвали, что удобно для проверки взаимодействий.
  - **Пустышки (dummies)** — просто объекты-заполнители, которые передаются в методы, но на самом деле не используются.
- **Проверка взаимодействий.** Имитации можно настраивать так, чтобы проверять, какие методы были вызваны, с какими аргументами и сколько раз. После выполнения теста фреймворк проверит, соответствуют ли фактические вызовы ожиданиям. Это помогает убедиться, что модуль взаимодействует с зависимостями так, как задумано.

У применения имитаций множество преимуществ.

- **Изоляция и фокус.** Благодаря изоляции тестируемого модуля от зависимостей можно сосредоточиться только на его поведении, без лишних отвлечений.
- **Контроль над тестовой средой.** Имитации помогают смоделировать разные сценарии, управляя входными данными и ожидаемыми результатами.
- **Более надежные тесты.** Исключая реальные зависимости, вы снижаете риск нестабильной работы тестов из-за внешних факторов — например, сбоев сети или ошибок базы данных.
- **Ускорение тестирования.** Имитации обычно выполняются в памяти и не требуют долгих операций вроде сетевых запросов или работы с базой данных, что ускоряет тестирование.
- **Большее покрытие.** С помощью имитаций можно легко смоделировать редкие случаи и условия ошибок, которые сложно воспроизвести при использовании настоящих зависимостей.

Например, можно создать имитацию интерфейса, чтобы протестировать метод сервиса, который зависит от слоя хранения данных, — при этом обращаться к реальной базе данных не придется. Или сделать имитацию сервиса журналирования и проверить, что ошибка действительно записывается в журнал при определенном вводе, без необходимости создавать настоящий файл.

## Библиотеки для имитаций

Выбор библиотеки для создания имитаций часто зависит от личных предпочтений, потребностей команды, особенностей проекта и используемого стека технологий. Библиотеки Moq, NSubstitute и FakeItEasy предлагают современный, удобный для разработчиков подход к созданию имитаций. У всех этих библиотек активные сообщества и регулярные обновления, поэтому это отличный выбор для большинства проектов на платформе .NET. Выбирая библиотеку для имитаций, стоит попробовать несколько вариантов на практике, чтобы понять, какая из них лучше всего подходит вашему стилю работы и требованиям проекта.

Moq — одна из простейших в использовании библиотек для создания имитаций в .NET. Она известна своим понятным синтаксисом и возможностью быстро создавать имитации без необходимости в сложных настройках. Библиотека поддерживает создание имитаций для интерфейсов, абстрактных и конкретных классов, настройку возвращаемых значений, проверку вызовов методов и работу со свойствами. Для создания имитаций в Moq активно используются лямбда-выражения, такой код понятнее и проще рефакторить.



**Внимание!** В августе 2023 года разработчик библиотеки Moq, известный под псевдонимом kzu, вызвал споры, добавив в версию 4.20.0 компонент под названием SponsorLink. Подробнее об изменениях можно прочитать в примечаниях к релизу по адресу [github.com/devlooped/moq/pull/1363](https://github.com/devlooped/moq/pull/1363). Код этого компонента был закрыт и обfuscирован, а также собирали адреса электронной почты, привязанные к учетным записям Git, и отправляли их на удаленный сервер. Многие разработчики сочли такие действия неприемлемыми, что обсуждалось, в частности, в ветке [www.reddit.com/r/programming/comments/15m2q0o/moq\\_a\\_net\\_mocking\\_library\\_now\\_ships\\_with\\_a/](https://www.reddit.com/r/programming/comments/15m2q0o/moq_a_net_mocking_library_now_ships_with_a/). Через несколько дней в версии 4.20.69 компонент SponsorLink был удален, но осадочек остался, и многие организации после этого предпочли перейти на альтернативные библиотеки, такие как NSubstitute. Эту тему также обсуждали по адресу [www.reddit.com/r/dotnet/comments/173ddyk/now\\_that\\_the\\_controversy\\_from\\_moqs\\_dependencies/](https://www.reddit.com/r/dotnet/comments/173ddyk/now_that_the_controversy_from_moqs_dependencies/).

NSubstitute — это библиотека для создания имитаций, в которой особое внимание уделено простоте и удобству. С ней доступен лаконичный и понятный API, благодаря которому код тестов упрощается и читается легче. Эта библиотека прекрасно подойдет тем, кто ценит чистоту кода и ясную структуру тестов. Как и Moq, библиотека NSubstitute позволяет имитировать интерфейсы и классы, проверять вызовы методов и задавать поведение в зависимости от аргументов. Кроме того, у нее есть удобная возможность автоматически создавать замещающие зависимости при создании объекта — это помогает сократить количество шаблонного кода.



**В копилку знаний:** больше информации о NSubstitute можно найти на сайте GitHub по адресу [github.com/nsubstitute/NSubstitute](https://github.com/nsubstitute/NSubstitute).

Разработчики FakeItEasy стремятся создать одну из самых удобных библиотек имитаций для .NET. Ее синтаксис создавался с упором на простоту — так, чтобы код было приятно читать и легко писать. Эта библиотека подойдет тем, кто ищет сбалансированное решение: функциональное, но при этом неперегруженное. С ее помощью можно быстро создать тестовые дублеры интерфейсов и классов, настроить поведение, задать возвращаемые значения или генерировать исключения. Поддерживаются и более сложные сценарии, например проверка вызовов методов с конкретными аргументами.



**В копилку знаний:** подробнее о FakeItEasy можно узнать на официальном сайте по адресу [fakeiteeasy.github.io](https://fakeiteeasy.github.io).

## Создание тестовых дублеров с помощью NSubstitute

Библиотека NSubstitute создана в целях упрощения работы с тестовыми дублерами, такими как имитации, заглушки и фальшивки, при модульном тестировании. С ее помощью вы можете протестировать компонент отдельно от настоящих зависимостей и при этом сохранить ясность и предсказуемость поведения.

Чтобы создать имитацию, вызовите метод `Substitute.For<T>()`, где `T` — это интерфейс или класс, который вы хотите имитировать:

```
ICalculator calc = Substitute.For<ICalculator>();
```

Если у класса есть конструктор, параметры можно передать, как показано ниже:

```
var substitute = Substitute.For<ClassWithConstructor>(5, "Error");
```

Объект `substitute` будет содержать не только члены заменяемого типа, например метод `Add` интерфейса `ICalculator`. В нем также будут доступны методы расширения, которые позволяют настраивать поведение подстановки. Эти методы перечислены в табл. 11.5.

**Таблица 11.5.** Методы расширения NSubstitute

Метод расширения	Описание
<code>Returns</code>	Позволяет задать возвращаемое значение. Вызовите метод, как обычно, и добавьте <code>Returns()</code> : например, <code>calc.Add(2, 3).Returns(5)</code> . Вместо фиксированного значения можно передать лямбда-выражение — например, чтобы выбросить исключение
<code>Received</code>	Проверяет, что метод действительно был вызван. Например: <code>calc.Received().Add(2, 3)</code> — подтверждает, что метод <code>Add</code> вызывался с этими аргументами

Если нужно проверить вызовы с любыми аргументами, можно воспользоваться методом `Args.Any<T>`. Например, можно задать поведение так, чтобы при сложении любого целого числа с 5 возвращалось значение 7:

```
calc.Add(Arg.Any<int>(), 5).Returns(7);
```

С помощью метода `Args.Is<T>()` можно сопоставить конкретные аргументы. Например, можно настроить поведение так, чтобы при сложении любого целого числа больше 3 с числом 5 возвращалось значение 9:

```
calc.Add(Arg.Is<int>(x => x > 3), 5).Returns(9);
```

И конечно, вы можете настроить тестовый дублер так, чтобы он выбрасывал исключение:

```
calc.Add(-1, -1).Returns(x => { throw new Exception(); });
```

## Создание имитаций с помощью NSubstitute на практике

Рассмотрим, как это можно сделать.

1. В проекте `BusinessLogic` создайте файл интерфейса `IEmailSender.cs`.
2. В созданном файле определите интерфейс для отправки электронных писем:

```
namespace Packt.Shared;

public interface IEmailSender
{
    bool SendEmail(string to, string subject, string body);
}
```

3. В проекте `BusinessLogic` создайте файл класса `UserService.cs`.
4. В этом файле определите класс для создания пользователя, который при регистрации отправляет электронное письмо:

```
namespace Packt.Shared;

public class UserService
{
    private readonly IEmailSender _emailSender;

    public UserService(IEmailSender emailSender)
    {
        _emailSender = emailSender;
    }

    public bool CreateUser(string email, string password)
    {
        // Создание пользователя
        bool successfulUserCreation = true;
```

```
// Отправка письма пользователю
bool successfulEmailSend = _emailSender.SendEmail(
    to: email,
    subject: "Welcome!",
    body: "Your account is created.");

    return successfulEmailSend && successfulUserCreation;
}
}
```

5. Выполните сборку проекта **BusinessLogic**.
6. В проекте **BusinessLogicUnitTests** добавьте ссылку на пакет **NSubstitute**:

```
<PackageReference Include="NSubstitute" Version="5.1.0" />
```



Актуальную версию пакета можно найти на сайте [www.nuget.org/packages/NSubstitute](http://www.nuget.org/packages/NSubstitute).

7. Выполните сборку проекта **BusinessLogicUnitTests**, чтобы восстановить зависимости.
8. В проекте **BusinessLogicUnitTests** создайте файл класса **EmailSenderUnitTests.cs**.
9. В созданном файле добавьте код для тестового класса, имитирующего интерфейс **IEmailSender** с помощью библиотеки **NSubstitute**:

```
using NSubstitute; // Подключение метода Substitute
using Packt.Shared; // Подключение интерфейса IEmailSender
using Xunit.Abstractions; // Подключение интерфейса ITestOutputHelper

namespace BusinessLogicUnitTests;

public class EmailSenderUnitTests
{
    private readonly ITestOutputHelper _output;

    public EmailSenderUnitTests(ITestOutputHelper output)
    {
        _output = output;
    }

    [Fact]
    public void SendEmailTest()
    {
        #region Arrange
        IEmailSender emailSender = Substitute.For<IEmailSender>();

        emailSender.SendEmail(
            to: Arg.Any<string>(),
```

```
subject: Arg.Any<string>(),
body: Arg.Any<string>()
.Returns(true);

emailSender.When(x => x.SendEmail(
    to: Arg.Is<string>(s => s.EndsWith("example.com")),
    subject: Arg.Any<string>(),
    body: Arg.Any<string>())
.Do(x => _output.WriteLine("Email sent to example domain."));

UserService sut = new(emailSender);
#endregion

#region Act
bool result = sut.CreateUser("user@example.com", "password");
#endregion

#region Assert
Assert.True(result);
emailSender.Received(requiredNumberOfCalls: 1)
    .SendEmail(to: "user@example.com",
        subject: Arg.Any<string>(), body: Arg.Any<string>());
#endregion
}
}
```

10. Запустите тест и убедитесь, что он прошел успешно.

Обратите внимание, как устроен код метода тестирования.

- Метод `Substitute.For<IEEmailSender>()` создает объект-имитацию интерфейса `IEEmailSender`.
- Метод `Returns` используется для задания возвращаемого значения при вызове метода `SendEmail` с любыми аргументами.
- Методы `When` и `Do` позволяют настроить действие, которое будет выполняться только в том случае, если письмо отправляется на адрес в домене `example.com`. При этом сообщение будет записано в тестовый вывод.
- Метод `Received` проверяет, что метод `SendEmail` действительно был вызван одинократно с ожидаемыми аргументами.

## Произвольные утверждения в модульном тестировании

Библиотека Fluent Assertions содержит набор удобных методов расширения, которые помогают писать и читать код в модульных тестах, а также разбираться в сообщениях об ошибках в случае неудачных тестов. Формулировки становятся ближе к естественному человеческому языку, в частности английскому.

Библиотека совместима с большинством фреймворков модульного тестирования, включая xUnit. Когда вы подключаете пакет FluentAssertions, библиотека автоматически находит используемый вами фреймворк и адаптируется для работы с ним, в том числе для выбрасывания исключений.

Чтобы использовать FluentAssertions, достаточно импортировать соответствующее пространство имен. После этого можно вызывать метод расширения `Should()` для переменной, а затем добавлять один из множества доступных методов для создания понятных утверждений. При необходимости можно объединять несколько утверждений в цепочку с помощью метода `And()` либо оставлять их отдельными, каждое со своим вызовом `Should()`.

## Утверждения для строковых значений

Начнем с создания утверждения об одном значении типа `string`.

1. В редакторе кода создайте C#-проект `xUnit Test Project` (Тестовый проект xUnit / `xunit`, присвойте ему имя `FluentTests` и добавьте в решение `Chapter11`).
2. Откройте файл `FluentTests.csproj` проекта и добавьте в него ссылку на пакет `FluentAssertions`:

```
<ItemGroup>
<PackageReference Include="FluentAssertions" Version="6.12.0" />
<PackageReference Include="coverlet.collector" Version="6.0.0" />
```



На момент публикации книги может быть доступна новая версия `FluentAssertions`. Проверить актуальную версию можно на сайте [www.nuget.org/packages/FluentAssertions/](http://www.nuget.org/packages/FluentAssertions/).

3. Выполните сборку проекта `FluentTests`.
4. Переименуйте файл `UnitTest1.cs` в `FluentExamples.cs`.
5. В файле `FluentExamples.cs` импортируйте пространство имен `FluentAssertions`, чтобы использовать методы расширения библиотеки, и добавьте метод тестирования для строки:

```
using FluentAssertions; // Подключение методов расширения библиотеки

namespace FluentTests;

public class FluentExamples
{
    [Fact]
    public void TestString()
    {
        string city = "London";
        string expectedCity = "London";
```

```

    city.Should().StartWith("Lo")
        .And.Endwith("on")
        .And.Contain("do")
        .And.HaveLength(6);

    city.Should().NotBeNull()
        .And.Be("London")
        .And.BeSameAs(expectedCity)
        .And.BeOfType<string>();

    city.Length.Should().Be(6);
}
}

```

6. Запустите тесты:

- в Visual Studio выберите команду меню **Test ▶ Run All Tests** (Тестирование ▶ Запустить все тесты);
- в Visual Studio Code на панели **Terminal** (Терминал) выполните команду **dotnet test**.

7. Убедитесь, что тест прошел успешно.

8. В методе **TestString** удалите последнюю букву **n** в слове **London**.

9. Перезапустите тест и обратите внимание, что он завершится с ошибкой. Вывод будет содержать подробности о несоответствии:

```
Expected city "Londo" to end with "on".
```

10. Верните букву **n** в слово **London**.

11. Выполните тест еще раз, чтобы убедиться, что ошибка не проявляется.

## Создание утверждений для коллекций и массивов

Теперь потренируемся создавать утверждения для коллекций и массивов.

1. В файле **FluentExamples.cs** добавьте еще один метод тестирования, который будет проверять содержимое коллекции:

```

[Fact]
public void TestCollections()
{
    string[] names = { "Alice", "Bob", "Charlie" };

    names.Should().HaveCountLessThan(4,
        "because the maximum items should be 3 or fewer");

    names.Should().OnlyContain(name => name.Length <= 6);
}

```

2. Запустите тесты. Проверка коллекций завершится неудачно — в отчете будет указано, какие элементы не соответствуют ожиданиям:

```
Expected names to contain only items matching (name.Length <= 6), but
{"Charlie"} do(es) not match.
```

3. В коде метода `TestCollection` исправьте слово `Charlie` на `Charly`.  
 4. Перезапустите тесты и убедитесь, что теперь они проходят успешно.

## Создание утверждений для значений даты и времени

Теперь потренируемся писать утверждения для значений даты и времени.

1. В файле `FluentExamples.cs` импортируйте пространство имен, добавляющее методы расширения для работы с именованными месяцами и другими удобными функциями, связанными с датой и временем:

```
using FluentAssertions.Extensions; // Подключение элемента February
// и других вспомогательных методов
```

2. В этом же файле добавьте метод тестирования, который проверяет утверждения о дате и времени:

```
[Fact]
public void TestDateTimes()
{
    DateTime when = new(
        hour: 9, minute: 30, second: 0,
        day: 25, month: 3, year: 2024);

    when.Should().Be(25.March(2024).At(9, 30));

    when.Should().BeOnOrAfter(23.March(2024));

    when.Should().NotBeSameDateAs(12.February(2024));

    when.Should().HaveYear(2024);

    DateTime due = new(
        hour: 11, minute: 0, second: 0,
        day: 25, month: 3, year: 2024);

    when.Should().BeAtLeast(2.Hours()).Before(due);
}
```

3. Запустите тесты. Тест значений даты и времени завершится с ошибкой:

```
Expected when <2024-03-25 09:30:00> to be at least 2h before <2024-03-25
11:00:00>, but it is behind by 1h and 30m.
```

4. Установите для переменной `due` значение часа (`hour`) равным 13 вместо 11.
5. Снова запустите тесты. Проверка даты и времени завершится успешно.



**В копилку знаний:** более подробно о возможностях библиотеки `FluentAssertions` можно узнать на сайте [fluentassertions.com](http://fluentassertions.com).

## Генерация тестовых данных с библиотекой Bogus

Bogus – популярная библиотека для .NET, предназначенная для создания фальшивых данных, которые удобно использовать в тестах. Она особенно полезна, когда нужно быстро сгенерировать реалистичные данные без необходимости вручную собирать каждую запись.

Библиотека предоставляет класс `Faker<T>`, который отличается гибкостью и позволяет настроить генерацию значений для свойств вашего класса через цепочку методов.

Наиболее используемые члены класса `Faker<T>` приведены в табл. 11.6.

**Таблица 11.6.** Наиболее популярные члены класса `Faker<T>`

Член	Описание
<code>RuleFor(property, generator)</code>	Позволяет задать правило для свойства, используя лямбда-выражение для указания способа генерации значения
<code>Clone()</code>	Клонирует экземпляр <code>Faker&lt;T&gt;</code> , чтобы можно было вносить изменения, не затрагивая оригинал. Особенно удобно, если нужно временно изменить некоторые правила
<code>StrictMode(true)</code>	При значении <code>true</code> проверяет, что для всех свойств заданы правила, и выбрасывает исключение, если какое-то правило отсутствует
<code>Generate()</code>	Создает один экземпляр объекта на основе заданных правил
<code>Generate(count)</code>	Создает <code>count</code> экземпляров объекта и возвращает их в списке <code>List&lt;T&gt;</code>
<code>GenerateBetween(min, max)</code>	Генерирует случайное количество экземпляров от <code>min</code> до <code>max</code> и возвращает их в списке <code>List&lt;T&gt;</code>

*Продолжение ↗*

**Таблица 11.6** (окончание)

Член	Описание
GenerateLazy(count)	Создает последовательность <code>IEnumerable&lt;T&gt;</code> из <code>count</code> объектов, используя отложенную оценку
GenerateForever()	Создает бесконечную последовательность объектов с немедленным выполнением

В библиотеке Bogus встроено много генераторов для создания тестовых данных. Некоторые из них перечислены в табл. 11.7.

**Таблица 11.7.** Встроенные генераторы значений в библиотеке Bogus

Член	Описание
<code>IndexFaker</code>	Автоматически увеличивается для каждого нового объекта, созданного экземпляром <code>Faker&lt;T&gt;</code> на протяжении его жизни
<code>Name.FirstName()</code> <code>Name.LastName()</code> <code>Name.FullName()</code>	Генерирует имена и связанные с ними значения
<code>Internet.Email()</code> <code>Internet.Url()</code> <code>Internet.Ip()</code>	Создает адреса электронной почты, URL-адреса и IP-адреса
<code>Address.StreetAddress()</code> <code>Address.City()</code> <code>Address.Country()</code>	Генерирует данные, связанные с почтовыми адресами
<code>Phone.PhoneNumber()</code>	Создает телефонные номера
<code>Date.Between(start, end)</code> <code>Date.Recent([days])</code> <code>Date.Soon([days])</code> <code>Date.Future(years, refDate)</code> <code>Date.Past(years, refDate)</code>	Генерирует даты и время. Можно задавать недавние, ближайшие или отстоящие на несколько лет даты
<code>Commerce.ProductName()</code> <code>Commerce.Price()</code> <code>Commerce.Categories(1).First()</code>	Создает наименования товаров, цены и категории для коммерческих данных
<code>Finance.CreditCardNumber()</code> <code>Finance.Currency().Code</code>	Генерирует финансовые данные, такие как номера банковских карт и коды валют

Член	Описание
Random.Int(18, 65) Random.Bool()	Генерирует случайные целые числа, логические значения и другие подобные данные
Image.DataUri(width, height) Image.PicsumUrl(width, height) Image.PlaceholderUrl(width, height)	Создает ссылки на изображения с использованием различных сервисов
Company.CompanyName() Company.Bs() Company.CatchPhrase()	Генерирует наименования компаний и слоганы
Vehicle.Fuel() Vehicle.Manufacturer() Vehicle.Model()	Создает данные о транспорте, такие как тип топлива, производитель и модель
Database.Column() Database.Engine()	Генерирует имена столбцов и типы движков баз данных и другие подобные значения

Представьте, что у вас есть класс User:

```
public class User
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string PhoneNumber { get; set; }
    public string StreetAddress { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
    public decimal AccountBalance { get; set; }
}
```

Для генерации поддельного экземпляра пользователя можно настроить объект Faker<User>, определяя правила для свойств:

```
Faker<User> userFaker = new Faker<User>()
    .RuleFor(u => u.Id, f => f.IndexFaker + 1)
    .RuleFor(u => u.FirstName, f => f.Name.FirstName())
    .RuleFor(u => u.LastName, f => f.Name.LastName())
    .RuleFor(u => u.Email, f => f.Internet.Email())
    .RuleFor(u => u.DateOfBirth, f => f.Date.Past(40, DateTime.Now.AddYears(-18)))
    .RuleFor(u => u.PhoneNumber, f => f.Phone.PhoneNumber())
    .RuleFor(u => u.StreetAddress, f => f.Address.StreetAddress())
    .RuleFor(u => u.City, f => f.Address.City())
    .RuleFor(u => u.Country, f => f.Address.Country())
    .RuleFor(u => u.AccountBalance, f => f.Finance.Amount());

User user = userFaker.Generate();
```

Метод `faker.Generate()` будет создавать объект `User` с реалистичными тестовыми данными для имени, фамилии и адреса электронной почты.

Теперь, когда вы познакомились с основами работы с библиотекой `Bogus`, давайте перейдем к практическому примеру.

## Тестовый проект с фальшивыми данными

Создадим тестовый проект .NET, в котором будет использоваться библиотека `Bogus` для генерации фальшивых данных.

1. В проекте `BusinessLogic` создайте файл `User.cs`.

2. В этом файле определите класс для представления пользователя:

```
namespace Packt.Shared;

public class User
{
    public int Id { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public string? Email { get; set; }
    public DateTime DateOfBirth { get; set; }

    public override string ToString()
    {
        return $"{Id}, {FirstName} {LastName}, {Email}, {DateOfBirth:yyyy-MM-dd}";
    }
}
```

3. В файле `UserService.cs` добавьте метод, который принимает объект `User` и проверяет, достиг ли пользователь совершеннолетия:

```
namespace Packt.Shared;

public class UserService
{
    ...

    public bool IsAdult(User user)
    {
        return user.DateOfBirth <= DateTime.Now.AddYears(-18);
    }
}
```

4. Выполните сборку проекта `BusinessLogic`.

5. В редакторе кода создайте C#-проект `xUnit Test Project` (Тестовый проект `xUnit`) / `xunit`, присвойте ему имя `BogusTests` и добавьте в решение `Chapter11`.

6. В файле `BogusTests.csproj` проекта укажите ссылки на пакеты `NSubstitute` и `Bogus`, а также на проект `BusinessLogic`:

```
<ItemGroup>
  <ProjectReference Include="..\BusinessLogic\BusinessLogic.csproj" />
</ItemGroup>

<ItemGroup>
  <PackageReference Include="NSubstitute" Version="5.1.0" />
  <PackageReference Include="Bogus" Version="35.5.1" />
  <PackageReference Include="coverlet.collector" Version="6.0.0" />
```



Проверить последнюю доступную версию библиотеки Bogus можно на сайте [www.nuget.org/packages/Bogus/](http://www.nuget.org/packages/Bogus/).

7. Выполните сборку проекта BogusTests.

## Создание теста с фальшивыми данными

Теперь создадим тест, в котором фальшивые данные будут использоваться для проверки работы метода `IsAdult` в сервисе пользователя.

1. Переименуйте файл `UnitTest1.cs` в `BogusExamples.cs`.
2. В этом файле подключите пространство имен для работы с библиотекой Bogus и добавьте два тестовых метода. Они будут создавать поддельные объекты `User` и проверять, правильно ли метод `IsAdult` определяет, совершеннолетний ли пользователь:

```
using NSubstitute; // Подключение метода Substitute
using Bogus; // Подключение класса Faker<T>
using Packt.Shared; // Подключение классов User и UserService
using Xunit.Abstractions; // Подключение интерфейса ITestOutputHelper

namespace BogusTests;

public class BogusExamples
{
    private readonly ITestOutputHelper _output;
    private readonly Faker<User> _userFaker;

    public BogusExamples(ITestOutputHelper output)
    {
        _output = output;

        _userFaker = new Faker<User>()
            // Настройка инкрементирующего индекса для свойства Id

            .RuleFor(u => u.Id, f => f.IndexFaker + 1)
            // Настройка случайного значения для свойства FirstName

            .RuleFor(u => u.FirstName, f => f.Name.FirstName())
            // Настройка случайного значения для свойства LastName
```

```

    .RuleFor(u => u.LastName, f => f.Name.LastName())
    // Настройка случайного адреса электронной почты для свойства Email

    .RuleFor(u => u.Email, f => f.Internet.Email())
    // Настройка случайной даты рождения на 30 лет раньше, чем 18 лет назад

    .RuleFor(u => u.DateOfBirth, f => f.Date.Past(yearsToGoBack: 30,
        refDate: DateTime.Now.AddYears(-18)));
}

[Fact]
public void IsAdult_ShouldReturnTrue_WhenUserIs180rOlder()
{
    // Подготовка
    IEmailSender emailSender = Substitute.For<IEmailSender>();
    UserService userService = new(emailSender);
    User user = _userFaker.Generate();

    _output.WriteLine($"{user}");

    // Выполнение
    bool result = userService.IsAdult(user);

    // Проверка
    Assert.True(result);
}

[Fact]
public void IsAdult_ShouldReturnFalse_WhenUserIsUnder18()
{
    // Подготовка
    IEmailSender emailSender = Substitute.For<IEmailSender>();
    UserService userService = new(emailSender);

    User user = _userFaker.Clone()
    // Переопределение свойства DateOfBirth для случайной даты рождения
    // 10 лет назад (то есть моложе 18 лет).
    .RuleFor(u => u.DateOfBirth, f => f.Date.Past(yearsToGoBack: 10,
        refDate: DateTime.Now.AddYears(-8)))
    .Generate();

    _output.WriteLine($"{user}");

    // Выполнение
    bool result = userService.IsAdult(user);

    // Проверка
    Assert.False(result);
}
}

```

3. Запустите тесты Bogus. Оба теста должны пройти успешно, подтвердив, что метод `IsAdult` правильно определяет взрослых и несовершеннолетних пользователей.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 11.1. Дополнительные онлайн-материалы

Официальную документацию по xUnit можно прочитать на сайте [xunit.net](http://xunit.net).

Подробнее о пакетах xUnit и рекомендациях по их использованию рассказано на странице [xunit.net/docs/nuget-packages](http://xunit.net/docs/nuget-packages).

Если вы работаете с MSTest, советую обратить внимание на статью Introducing MSTest SDK – Improved Configuration & Flexibility на сайте [devblogs.microsoft.com/dotnet/introducing-mstest-sdk/](http://devblogs.microsoft.com/dotnet/introducing-mstest-sdk/). В ней описаны последние улучшения MSTest и возможности конфигурации.

### Упражнение 11.2. Практические задания

Рассмотрим интерфейс `IProductService`, который отвечает за управление информацией о товарах в системе электронной коммерции. В его состав входят методы для получения данных о товаре, проверки наличия товара на складе, обновления информации, обработки уведомлений об изменениях и выполнения асинхронных операций:

```
public interface IProductService
{
    Product GetProductById(int productId);
    Task<bool> IsProductAvailableAsync(int productId);
    Product UpdateProductPrice(int productId, decimal newPrice);
    string DefaultCurrency { get; set; }
    event EventHandler<ProductUpdatedEventArgs> ProductUpdated;
    IEnumerable<Product> GetProducts(string category = null);
}

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
}

public class ProductUpdatedEventArgs : EventArgs
{
    public int ProductId { get; set; }
    public decimal NewPrice { get; set; }
}
```

Напишем модульные тесты для этого интерфейса с помощью xUnit и NSubstitute.

1. Настройте имитацию метода `GetProducts`, чтобы он возвращал разные наборы товаров в зависимости от указанной категории.
2. Настройте имитацию свойства `DefaultCurrency`, чтобы при обращении оно возвращало заданный код валюты. Так вы сможете протестировать, как система обрабатывает операции с разными валютами.
3. Настройте имитацию события `ProductUpdated`, чтобы отправить уведомление об изменении товара и проверить, как подписчики реагируют на обновления.
4. Настройте имитацию метода `IsProductAvailableAsync`, чтобы он асинхронно возвращал `true` или `false`. Это позволит проверить, как пользовательский интерфейс или другие сервисы реагируют на наличие товара.
5. Настройте имитацию методов `GetProductById` и `UpdateProductPrice`, чтобы возвращать заранее определенные экземпляры класса `Product`. Так можно будет проверить, как система обрабатывает получение и обновление информации о товарах.

### Упражнение 11.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Каким требованиям должен соответствовать качественный модульный тест?
2. Что такое MUT и какую роль он играет в модульном тестировании?
3. Что означает формула «красный – зеленый – рефакторинг» в подходе TDD?
4. Как выполняется настройка тестов в библиотеке xUnit?
5. Какой параметр является общим для атрибутов `[Fact]` и `[Theory]` в xUnit и зачем его указывать?
6. При использовании атрибутов `[Theory]` и `[ClassData]` какие требования предъявляются к предоставляющему данные классу?
7. Как правильно обрабатывать как успешные, так и провальные результаты модульного теста?
8. На какие настораживающие признаки стоит обратить внимание при написании или проверке модульных тестов?
9. Каким образом в xUnit можно выводить сообщения во время выполнения теста?
10. Как с помощью NSubstitute задать возвращаемое значение у подставного метода?

## Упражнение 11.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-11---unit-testing-and-mocking](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-11---unit-testing-and-mocking).

## Резюме

В этой главе мы разобрали:

- разные виды тестирования — от модульного до функционального;
- преимущества и ограничения разработки через тестирование (TDD);
- как организовать модульное тестирование с помощью xUnit;
- как передавать значения параметров в тестовые методы;
- как настраивать и завершать работу тестовых фикстур;
- как применять библиотеку NSubstitute для создания тестовых дублеров;
- как использовать библиотеку Bogus для генерации фальшивых данных.

В следующей главе мы перейдем к интеграционному тестированию и вопросам безопасности.

# 12

## Интеграционное тестирование и проверка безопасности

В этой главе речь пойдет об интеграционном тестировании и проверке безопасности — двух типах тестирования более высокого уровня. Модульные тесты отлично выявляют ошибки в бизнес-логике, но важно также убедиться, что все части кодовой базы корректно взаимодействуют друг с другом и с внешними системами. Именно на этом этапе становится необходимым интеграционное тестирование.

Одни внешние системы следует использовать напрямую в интеграционных тестах, а другие — заменять тестовыми дублерами. Интеграционные тесты часто взаимодействуют с внешними процессами, такими как базы данных, шины событий и очереди сообщений. Поэтому они медленнее по сравнению с модульными тестами, но при этом охватывают больший объем кода — как вашего собственного, так и сторонних библиотек. Интеграционные тесты также лучше выявляют случаи регрессии.



Реальный пример регрессии в пакете для журналирования в .NET с описанием проблемы приведен на сайте GitHub по адресу [github.com/markjprice/cs11dotnet7/blob/main/docs/errata/errata.md#page-178---reviewing-project-packages](https://github.com/markjprice/cs11dotnet7/blob/main/docs/errata/errata.md#page-178---reviewing-project-packages).

Один из инструментов, доступных как в терминале, так и в Visual Studio, который помогает упростить интеграционное тестирование веб-сервисов, — это туннели разработки. В этой главе мы рассмотрим, как использовать их для упрощения тестирования сервисов.

Проверка безопасности становится все важнее с каждым днем. Это один из самых сложных видов тестирования, поскольку надежная защита начинается с подхода Security by Design — и автоматизировать такой подход полностью невозможно.

### Основы интеграционного тестирования

*Интеграционное тестирование* — это этап проверки ПО, во время которого отдельные модули или компоненты приложения объединяются и тестируются как единое целое. Задача — убедиться, что модули корректно взаимодействуют между собой. Такой тип тестирования помогает выявить ошибки, возникающие при

интеграции компонентов: сбои передачи данных, несоответствие интерфейсов, проблемы в связях между системами.

Проверяя совместную работу взаимосвязанных компонентов, интеграционное тестирование позволяет убедиться, что вся система функционирует надежно и соответствует заданным требованиям. Обычно оно проводится после модульного тестирования и предшествует другим видам высокогоуровневой проверки, таким как тестирование безопасности и производительности.

Как уже упоминалось в главе 11, для интеграционного тестирования можно использовать те же инструменты, что и для модульного. Например, интеграционные тесты удобно писать с помощью xUnit.

Напомню, что модульные тесты должны:

- проверять отдельную единицу поведения;
- выполняться быстро;
- быть изолированными от других тестов.

Простейшее определение интеграционного теста — это автоматизированный тест, который не соответствует критериям модульного теста.

Есть еще одно важное различие между этими типами тестирования — это количество создаваемых тестов. Один из способов наглядно показать соотношение между модульным, интеграционным и сквозным тестированием — пирамида тестов (рис. 12.1).



**Рис. 12.1.** Пирамида тестов, отражающая соотношение уровней тестирования

В основании пирамиды должно быть больше модульных тестов, а количество сквозных тестов — минимальным. Интеграционные тесты занимают промежуточное положение.



**Дельный совет:** старайтесь определять интеграционные тесты для наиболее распространенных сценариев реального использования ваших приложений. Пишите достаточно большое количество интеграционных тестов для проверки взаимодействия с каждой используемой внешней системой. Крайние случаи, ведущие к ошибкам, лучше всего покрывать модульными тестами. Интеграционные тесты для крайних сценариев стоит создавать только тогда, когда их невозможно проверить на уровне модульных тестов.

## Какие внешние системы включать в интеграционное тестирование

Внешние системы можно условно разделить на два типа: находящиеся под вашим контролем и вне вашего контроля. К системам под вашим контролем относятся, например, хранилища данных, доступ к которым имеет только ваш проект. Никакие другие системы не обновляют эти данные напрямую. К системам вне вашего контроля относятся такие сервисы, как электронная почта, погодные API или государственные сервисы.

Как можно догадаться, внешние системы, находящиеся под вашим контролем, следует использовать напрямую в интеграционных тестах. Те системы, которые вы не контролируете, рекомендуется имитировать.

А что делать, если база данных используется не только вашим проектом? На старте система может быть полностью под вашим контролем, но со временем другие проекты могут получить прямой доступ к той же базе данных. Такая ситуация нарушает принципы проектирования микросервисов и может привести к проблемам в будущем. Тем не менее в реальных проектах это случается довольно часто. В таких случаях базу данных стоит рассматривать как неконтролируемую систему и применять ее имитацию для тестирования.

## Совместное использование фикстур в интеграционных тестах

Иногда возникает ситуация, когда фиксттуру действительно нужно сделать общей для всех тестов. Чаще всего это требуется при работе с базой данных или моделью EF Core, особенно в интеграционных тестах.

В таких случаях стоит использовать наследование, чтобы тестовому классу не требовался собственный конструктор:

```
public class NorthwindStoreTests : DatabaseIntegrationTests
{
    [Fact]
    public void Checkout_ShouldFailWhenLowInventory()
    {
        // Здесь используется фикстура _db
    }
    ...
}
```

```

public abstract class DatabaseIntegrationTests : IDisposable
{
    protected readonly NorthwindContext _db;

    protected DatabaseIntegrationTests(NorthwindContext db)
    {
        _db = db;
    }

    public void Dispose()
    {
        _db.Dispose();
    }
}

```

## Обзор примера интеграционного теста

Рассмотрим пример интеграционного теста, который проверяет получение товара из каталога по его идентификатору:

```

public class GetById
{
    private readonly CatalogContext _catalogContext;
    private readonly EfRepository<Order> _orderRepository;

    private OrderBuilder OrderBuilder { get; } = new OrderBuilder();

    private readonly ITestOutputHelper _output;

    public GetById(ITestOutputHelper output)
    {
        _output = output;
        DbContextOptionsBuilder<CatalogContext> dbOptions = new()
            .UseInMemoryDatabase(databaseName: "TestCatalog")
            .Options;
        _catalogContext = new CatalogContext(dbOptions);
        _orderRepository = new EfRepository<Order>(_catalogContext);
    }

    [Fact]
    public async Task GetsExistingOrder()
    {
        var existingOrder = OrderBuilder.WithDefaultValues();
        _catalogContext.Orders.Add(existingOrder);
        _catalogContext.SaveChanges();
        int orderId = existingOrder.Id;
        _output.WriteLine($"OrderId: {orderId}");

        var orderFromRepo = await _orderRepository.GetByIdAsync(orderId);
        Assert.Equal(OrderBuilder.TestBuyerId, orderFromRepo.BuyerId);

        // Примечание: при использовании базы данных InMemoryDatabase объект
        // OrderItems доступен.
        // При использовании SQL-базы данных он будет иметь значение null.
    }
}

```

```
// Вместо простого GetById лучше использовать класс OrderWithItemsByIdSpec,  
// чтобы получить полный агрегированный результат.  
var firstItem = orderFromRepo.OrderItems.FirstOrDefault();  
Assert.Equal(OrderBuilder.TestUnits, firstItem.Units);  
}  
}
```

Этот код представляет собой тестовый класс с именем `GetById`, который проверяет функциональность извлечения существующего заказа из хранилища.

Ключевые моменты теста.

- Для проверки используется база данных в оперативной памяти.
- Присутствуют зависимости от класса `CatalogContext` (пользовательский `DbContext`) и репозитория `EfRepository<Order>`.
- Конструктор тестового класса инициализирует базу данных в оперативной памяти и репозиторий. Он принимает экземпляр `ITestOutputHelper` для записи дополнительной информации о ходе тестирования.

Метод тестирования `GetsExistingOrder`.

- Использует класс `OrderBuilder` для создания экземпляра сущности `Order` со значениями по умолчанию.
- Добавляет созданный заказ в контекст `CatalogContext` и сохраняет его в базе данных в оперативной памяти.
- Извлекает идентификатор заказа и выводит его через `_output`.
- Загружает заказ из репозитория с помощью метода  `GetByIdAsync`.
- Проверяет, что значение `BuyerId` загруженного заказа совпадает с ожидаемым значением из `OrderBuilder`.
- Дополнительно проверяет, что первый элемент коллекции `OrderItems` содержит нужное количество единиц.

Такой тест позволяет убедиться, что хранилище корректно извлекает существующий заказ из базы данных в оперативной памяти, а свойства заказа соответствуют ожидаемым значениям.

## Интеграционное тестирование с хранилищами данных

Одни из наиболее распространенных систем, с которыми приходится работать в интеграционных тестах, — хранилища данных. Они требуют особого подхода.

Схему хранилища данных стоит рассматривать как часть кода проекта и хранить под управлением системы контроля версий, например Git. Благодаря этому все изменения схемы будут синхронизированы с изменениями кода, который использует эти данные.

Схема базы данных включает структуру таблиц, определения индексов, представления и хранимые процедуры. Для баз данных на основе SQL такие элементы обычно описываются в отдельных SQL-сценариях. Стоит также учитывать ссылочные данные — те, которые нужно заранее вставить в базу для корректной работы приложения. Например, в проекте может понадобиться заполнить таблицу CountryRegion около 200 строками, чтобы обеспечить поиск по регионам.

## Разработка экземпляров базы данных и миграции

Каждому разработчику в команде нужно работать с собственным экземпляром базы данных. Это важно, чтобы запущенные тесты не мешали друг другу и не возникало конфликтов данных. Такой подход также помогает повысить производительность при локальном выполнении тестов.

Со временем структура базы данных будет меняться: появятся новые таблицы, в существующие таблицы будут добавляться новые столбцы. Лучший способ управлять такими изменениями — использовать миграции — изменения схемы базы данных, оформленные в виде SQL-операторов, таких как `CREATE TABLE` или `ALTER TABLE`. Объектно-реляционные отображения (Object-Relational Mapping, ORM), например EF Core, поддерживают миграции через специальные классы с командами, эквивалентными SQL-операторам:

```
using Microsoft.EntityFrameworkCore.Migrations; // Подключение класса Migration

namespace YourApp.Migrations;

public partial class AddPosts : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(name: "Posts", columns: table => new
        {
            PostId = table.Column<int>(nullable: false)
                .Annotation("SqlServer:ValueGenerationStrategy", "IdentityColumn"),
            Title = table.Column<string>(nullable: true),
            Content = table.Column<string>(nullable: true),
            DateCreated = table.Column<DateTime>(nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Posts", x => x.PostId);
        });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(name: "Posts");
    }
}
```

Обратите внимание на следующие нюансы приведенного ранее кода.

- Метод `Up` создает таблицу `Posts` с четырьмя столбцами: `PostId`, `Title`, `Content` и `DateCreated`. Столбец `PostId` настроен как идентифицирующий — его значение будет автоматически генерироваться базой данных при вставке новой строки. Метод `table.PrimaryKey` указывает `PostId` в качестве первичного ключа таблицы `Posts`.
- Метод `Down` отменяет изменения, внесенные методом `Up`. В данном случае он просто удаляет таблицу `Posts`. Наличие метода `Down` позволяет при необходимости откатить изменения схемы базы данных.



Фактические SQL-команды, которые выполняются этими методами, зависят от используемого поставщика базы данных. EF Core автоматически преобразует методы миграции в подходящие SQL-команды для выбранного поставщика, например SQL Server, SQLite или PostgreSQL.

Класс миграции может быть автоматически сгенерирован с помощью инструментов EF Core на основе определений моделей и конфигурации класса `DbContext`. При каждом изменении модели, затрагивающем схему базы данных, рекомендуется создавать новую миграцию:

```
dotnet ef migrations add <MigrationName>
```

где `<MigrationName>` — это описательное название, например `AddPosts`.

Такой подход позволяет версионировать схему базы данных вместе с кодом приложения, упрощая управление изменениями и развертывание.

Чтобы применить все созданные миграции (то есть запустить методы `Up`), используется команда:

```
dotnet ef database update
```

Если нужно вернуться к конкретной миграции, откатив изменения, примененные после нее (то есть запустить методы `Down` для последующих миграций), используется команда:

```
dotnet ef database update <MigrationName>
```

Таким образом база данных будет приведена к состоянию, зафиксированному в указанной миграции. Если требуется откатить все миграции и вернуть базу данных в исходное состояние, можно выполнить следующую команду:

```
dotnet ef database update 0
```

Эта команда последовательно вызовет методы `Down` всех миграций.

## Жизненный цикл данных

Тесты не должны зависеть от состояния базы данных, оставленного предыдущими прогонами. Каждый тест должен самостоятельно инициализировать необходимое состояние базы данных, чтобы обеспечить согласованность и избежать влияния внешних факторов.

Если полностью сбросить состояние базы данных невозможно, тесты следует выполнять последовательно, чтобы текущее состояние оставалось предсказуемым. При параллельном выполнении тестов возрастает риск рассинхронизации данных.

Существует несколько распространенных способов сброса состояния базы данных между тестами.

- **Восстановление резервной копии базы перед каждым тестом.** Этот способ может быть медленным, особенно при работе с большими базами данных.
- **Откат транзакции, созданной в начале теста.** Подход эффективен, но важно учитывать, что поведение в тестах будет отличаться от поведения в реальной эксплуатации.
- **Очистка данных после выполнения теста.** Это относительно быстро, но если тест завершится с ошибкой и очистка не произойдет, это может повлиять на результаты других тестов.
- **Очистка данных перед выполнением каждого теста.** Этот способ также работает быстро и снижает вероятность, что база данных останется в непредсказуемом состоянии в случае сбоя теста.

Для интеграционных тестов, которые используют общую базу данных и однотипное начальное состояние, рекомендуется создать базовый тестовый класс. В этом классе можно вызвать SQL-сценарий для очистки и повторной инициализации базы данных перед каждым тестом:

```
using Microsoft.Data.SqlClient; // Подключение метода SqlConnection
                                // и других необходимых классов
using System.Data; // Подключение свойства CommandType

public abstract class DatabaseIntegrationTests
{
    private const string _connectionString;

    protected DatabaseIntegrationTests()
    {
        ResetDatabase();
    }

    public void ResetDatabase()
    {
        string sql = "DELETE FROM ...;" +
                    "DELETE FROM ...;" +
                    "INSERT INTO ...;" +
```

```
// Либо загрузка операторов SQL из внешнего файла сценария

using SqlConnection con = new(_connectionString);

SqlCommand cmd = new(sql, con);
cmd.CommandType = CommandType.Text;

con.Open();
cmd.ExecuteNonQuery();
}

}
```



**Дельный совет:** лучше избегать замены настоящих баз данных на БД в оперативной памяти. Хотя такой прием может ускорить тесты, современные системы хранения данных тоже обеспечивают схожую высокую производительность. По-настоящему надежным считается интеграционный тест, который работает с той же базой данных, что и в эксплуатации.

Чтобы упростить создание данных для тестов, удобно определить фабричные методы в отдельном вспомогательном классе:

```
public static class ObjectMother
{
    public static Category CreateCategory(
        int categoryId = 1,
        string categoryName = "Beverages",
        string description = "...")
    {
        using NorthwindContext db = new();
        Category category = new Category()
        {
            CategoryId = categoryId,
            CategoryName = categoryName,
            Description = description
        };
        db.Categories.Add(category);
        db.SaveChanges();
        return category;
    }
}
```

В самих тестах такие фабричные методы можно использовать для быстрой и чистой подготовки данных:

```
using static ObjectMother;

Category c1 = CreateCategory(); // Создание категории Beverages
Category c2 = CreateCategory(2, "Condiments", "..."); // Создание категории Condiments
```

## Тестирование сервисов с помощью туннелей разработки

*Туннели разработки* (*Dev tunnels*) становятся все более популярным инструментом среди разработчиков веб-сервисов. Их основная задача — создать безопасный публичный URL-адрес, который напрямую привязывается к локальному серверу на вашем компьютере. Благодаря этому вы можете быстро и безопасно поделиться своей локальной средой разработки, не развертывая веб-сервис на удаленном сервере.

Представьте, что вы работаете над веб-приложением на своем ноутбуке. Обычно такое приложение доступно только вам через `localhost`. Если возникает необходимость показать свою работу кому-то еще, вам, как правило, нужно опубликовать приложение на внешнем сервере — а это может быть трудоемким процессом, особенно если важно быстро получить обратную связь.

Туннели разработки упрощают эту задачу: они создают защищенное соединение между вашим локальным сервером и специальным сервисом, который предоставляет публичный URL-адрес. Когда пользователь открывает этот URL-адрес, сервис туннелей перенаправляет трафик на его компьютер и пользователь видит приложение так, словно оно уже размещено в интернете.

Туннели разработки открывают множество возможностей, среди которых особенно важны следующие.

- **Упрощенная совместная работа.** Туннели позволяют легко делиться своими наработками с клиентами, тестировщиками или коллегами, не развертывая их в публичной инфраструктуре. Это особенно удобно для быстрой демонстрации результатов или совместной отладки.
- **Тестирование в реальных условиях.** Туннели позволяют тестировать веб-хуки, сторонние интеграции и мобильные приложения, которым необходим публичный URL-адрес, — прямо из вашей локальной среды.
- **Обучение и эксперименты.** Для студентов и преподавателей туннели разработки становятся отличным инструментом: они упрощают работу над проектами, позволяют безопасно экспериментировать с новыми технологиями и быстро делиться результатами без необходимости настраивать полноценные серверы.

Прежде чем начать использовать туннели разработки, полезно ознакомиться с ключевыми терминами, приведенными в табл. 12.1.

Хотя туннели разработки создаются с учетом требований безопасности, открытие локальной среды для публичного доступа всегда связано с определенными рисками. Очень важно использовать такие инструменты осознанно: защищать конфиденциальные данные, а также при необходимости ограничивать доступ с помощью проверки подлинности или настройки белых списков IP-адресов.

**Таблица 12.1.** Важные термины, касающиеся туннелей разработки

Термин	Описание
<b>Туннель</b>	Обеспечивает безопасный удаленный доступ к одному узлу через сервис ретрансляции. Каждый туннель разработки имеет уникальное DNS-имя, поддерживает несколько портов, настройки управления доступом и содержит связанные с ним метаданные
<b>Сервис ретрансляции туннеля</b>	Обеспечивает безопасные соединения между узлом туннеля разработки и клиентами через облачный сервис. Это особенно важно, если узел находится за брандмауэром и не принимает прямые входящие соединения
<b>Туннельный узел</b>	Принимает клиентские подключения через сервис ретрансляции туннеля и перенаправляет их на локальные порты компьютера разработчика
<b>Туннельный порт</b>	Определенный IP-порт (в диапазоне от 1 до 65535), через который разрешено подключение к туннелю разработки. В одном туннеле можно настроить несколько портов, причем каждый порт может использовать разные протоколы, такие как HTTP или HTTPS, и иметь свои настройки доступа

Еще один момент, который стоит учитывать: производительность приложения, работающего через туннель, может отличаться от производительности полностью развернутой версии. Накладные расходы на установку туннеля и особенности локальной среды разработки могут влиять на скорость отклика и стабильность соединения.

## Установка CLI для туннелей разработки

Прежде чем начать использовать туннели разработки, потребуется установить соответствующий интерфейс командной строки (CLI).

- В операционной системе Windows можно воспользоваться диспетчером пакетов winget:

```
winget install Microsoft.dev tunnel
```

- В macOS установка выполняется с помощью утилиты Homebrew:

```
brew install --cask dev tunnel
```

- В среде Linux установка производится через инструмент curl:

```
curl -sL https://aka.ms/DevTunnelCliInstall | bash
```



После установки необходимо перезапустить терминал, чтобы команды CLI для туннелей разработки стали доступны в системе.

## Создание туннеля разработки с помощью CLI и эхо-сервиса

Эхо-сервис — это простой сервер или конечная точка, принимающая HTTP-запросы и возвращающая отправленные данные в ответ. Такие сервисы часто используются для тестирования, отладки и обучения, поскольку позволяют увидеть, какие данные передаются на сервер, и убедиться, что они обрабатываются корректно.

Создадим туннель разработки вместе с эхо-сервисом для проведения базовой проверки.

1. Авторизуйтесь в системе с учетной записью Microsoft Entra ID, Microsoft или GitHub:

```
devtunnel user login
```

2. Выберите свою учетную запись и посмотрите результат команды — вы увидите подтверждение успешного входа:

```
Logged in as <your-email-account> using Microsoft.
```

3. Запустите простой эхо-сервис на порту **8080**, чтобы принимать HTTP-запросы и возвращать их содержимое:

```
devtunnel echo http -p 8080
```

4. Откройте новое окно терминала и создайте туннель разработки для порта **8080**:

```
devtunnel host -p 8080
```

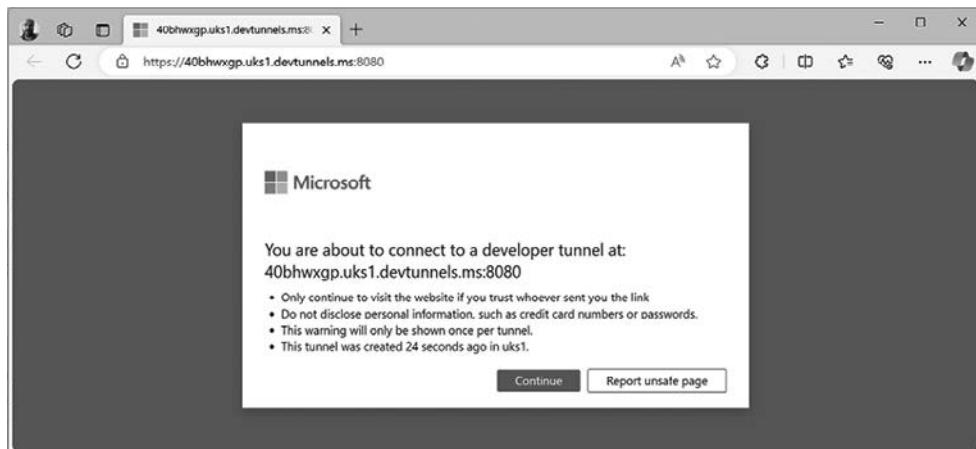
5. В выводе отобразится URL-адрес туннеля. Например, он может выглядеть так:

```
Hosting port: 8080
Connect via browser: https://40bhwxgp.uks1.devtunnels.ms:8080,
https://40bhwxgp-8080.uks1.devtunnels.ms
Inspect network activity: https://40bhwxgp-8080-inspect.uks1.devtunnels.ms

Ready to accept connections for tunnel: happy-hill-zw1k7n8
```

6. Откройте любой браузер и перейдите по указанному адресу. Например: **40bhwxgp.uks1.devtunnels.ms:8080**.

7. Откройте любой браузер и перейдите по указанному адресу. По умолчанию туннель доступен только вам.
8. В браузере отобразится предупреждение о подключении к туннелю разработки (рис. 12.2).



**Рис. 12.2.** Подключение к туннелю разработки

9. После проверки можно закрыть браузер. На этом этапе подключение к туннелю нам не требуется.
10. Вернитесь в терминал, где запущен узел туннеля, и нажмите сочетание клавиш **Ctrl+C**, чтобы остановить его.
11. Аналогично завершите работу эхо-сервиса — снова нажмите **Ctrl+C** в соответствующем терминале.

Теперь разберемся, как использовать туннель разработки в проекте ASP.NET Core.

## Исследование туннеля разработки с проектом ASP.NET Core

Теперь рассмотрим практический пример использования туннеля разработки в проекте ASP.NET Core.

1. В редакторе кода в решении **Chapter12** создайте проект веб-API, указав следующие настройки:
  - выберите шаблон проекта **ASP.NET Core Web API** (**Веб-API ASP.NET Core**) / **webapi**;
  - в поле **Solution name** (**Имя решения**) укажите значение **Chapter12**;

- в поле Project name (Имя проекта) укажите значение `Northwind.WebApi`;
- в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт `None` (Отсутствует);
- установите флажок `Configure for HTTPS` (Настроить для HTTPS);
- снимите флажок `Enable Docker` (Включить поддержку контейнера);
- установите флажок `Enable OpenAPI support` (Включить поддержку OpenAPI);
- снимите флажок `Do not use top-level statements` (Не использовать операторы верхнего уровня);
- снимите флажок `Use controllers` (Использовать контроллеры).



**Внимание!** Программа Rider на момент чтения книги может не поддерживать создание проектов веб-API на базе минимальных API. В этом случае рекомендуется создать проект с помощью команды `dotnet new`, а затем вручную добавить его в решение Rider.

2. Откройте файл `launchSettings.json`, расположенный в папке `Properties`. Измените параметр `applicationUrl` для профиля `https`, указав порт 5121 для протокола HTTPS и 5122 — для HTTP:

```
"profiles": {
  ...
  "https": {
    "commandName": "Project",
    "dotnetRunMessages": true,
    "launchBrowser": true,
    "launchUrl": "swagger",
    "applicationUrl": "https://localhost:5121;http://localhost:5122",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    }
}
```

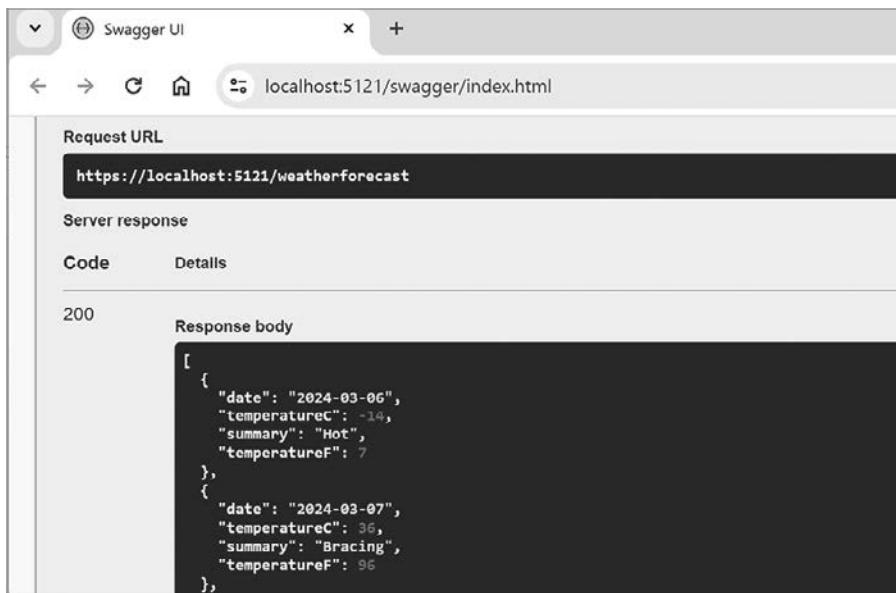


Visual Studio автоматически считывает файл настроек проекта, запускает браузер и переходит по адресу, указанному в полях `applicationUrl` и `launchUrl`, если параметру `launchBrowser` присвоено значение `true`. В Code и при выполнении команды `dotnet run` браузер не запускается, поэтому адрес `localhost:5121/swagger` нужно будет ввести вручную.

3. В файле `Program.cs` перед вызовом метода `app.Run()` добавьте код для вывода URL-адреса туннеля разработки:

```
string? tunnelUrl = Environment.GetEnvironmentVariable("VS_TUNNEL_URL");
if (tunnelUrl is not null)
{
  Console.WriteLine($"Tunnel URL: {tunnelUrl}");
}
```

4. Запустите проект без отладки.
5. Проверьте, что веб-сервис погоды автоматически открылся в браузере и отобразилась страница Swagger.
6. Воспользуйтесь сервисом погоды и посмотрите, как он возвращает случайный прогноз (рис. 12.3).

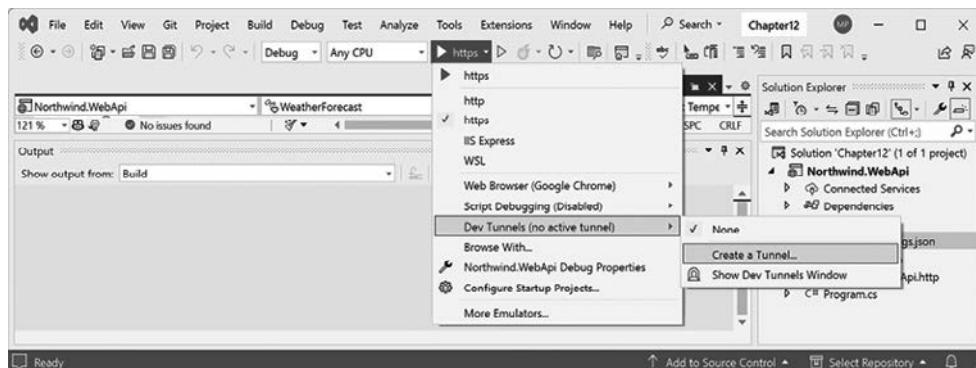


**Рис. 12.3.** Случайный прогноз, извлеченный из сервиса на сервере localhost

7. В терминале проверьте вывод: сервис должен быть размещен на localhost и слушать порты 5121 и 5122:

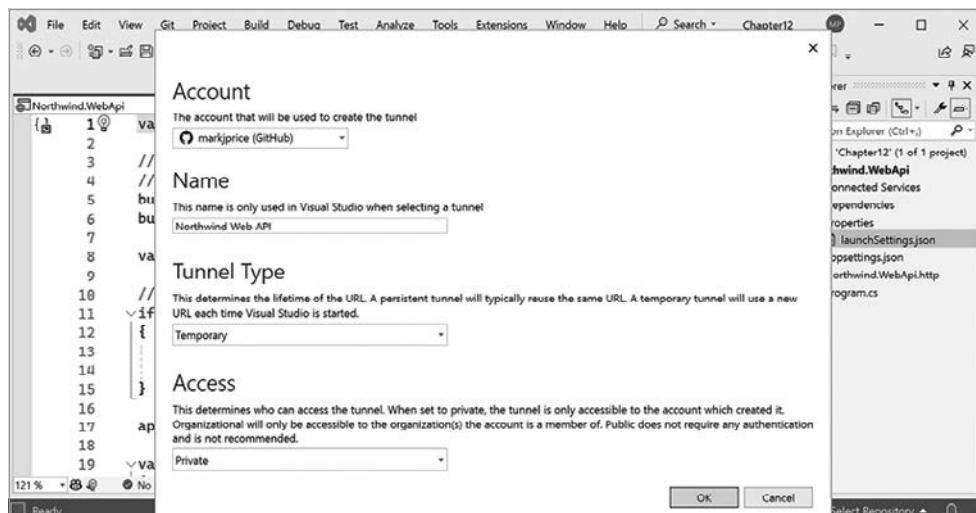
```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5121
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5122
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\tools-skills-net8\Chapter12\Northwind.WebApi
```

8. Закройте браузер и остановите веб-сервис.
9. В Visual Studio на панели инструментов откройте раздел **https ▶ Dev Tunnels (no active tunnel) ▶ Create a Tunnel** (**https ▶ Туннели разработки (нет активного туннеля) ▶ Создать туннель**) (рис. 12.4).



**Рис. 12.4.** Создание туннеля разработки в Visual Studio

10. В открывшемся диалоговом окне выберите учетную запись Microsoft или GitHub, введите имя туннеля, например Northwind Web API, выберите тип туннеля Temporary (Временный) и укажите уровень доступа Private (Приватный). После этого нажмите кнопку OK (рис. 12.5).



**Рис. 12.5.** Создание туннеля разработки в Visual Studio

11. Запустите проект без отладки.
12. Если используется учетная запись GitHub, для работы с туннелями разработки может потребоваться пройти авторизацию, чтобы подтвердить личность.
13. Проверьте работу сервиса погоды и посмотрите, как он снова возвращает случайный прогноз (рис. 12.6).

The screenshot shows a browser window with the title 'Swagger UI'. The address bar contains '8jq6g75k-5121.eks1.devtunnels.ms/swagger/index.html'. Below the address bar, there's a 'Request URL' field with the value 'https://8jq6g75k-5121.eks1.devtunnels.ms/weatherforecast'. Under the 'Server response' section, there are tabs for 'Code' and 'Details', with 'Code' selected. The status code '200' is shown, followed by the 'Response body'. The response body is a JSON array containing two objects, each representing a weather forecast for a specific date. The first object has a date of '2024-03-06', a temperature of 2, a summary of 'Cool', and a temperatureF of 35. The second object has a date of '2024-03-07', a temperature of 21, a summary of 'Warm', and a temperatureF of 68.

```

[{"date": "2024-03-06", "temperatureC": 2, "summary": "Cool", "temperatureF": 35}, {"date": "2024-03-07", "temperatureC": 21, "summary": "Warm", "temperatureF": 68}]
  
```

**Рис. 12.6.** Случайный прогноз погоды, полученный через веб-сервис в интернете

14. В терминале видно, что веб-сервис продолжает работать на `localhost` и прослушивает порты 5121 и 5122. Туннель разработки перенаправляет внешние HTTP-запросы через публичный URL на локальный веб-сервис, благодаря чему переменная среды с адресом туннеля становится доступной, как показано в следующем выводе:

```
Tunnel URL: https://8jq6g75k-5121.eks1.devtunnels.ms/
```

15. В Visual Studio выполните команду меню `View > Output` (Вид > Вывод), затем в списке `Show output from` (Показать вывод из) выберите пункт `Dev Tunnels` (Туннели разработки). В выводе будет указана информация о туннеле и доступных адресах:

```

Getting dev tunnels for account 'Mark Price (markjprice@msn.com)': Succeeded
Getting dev tunnels for account 'markjprice (GitHub)': Succeeded
Dev tunnel 'Northwind Web API' was created successfully
Successfully configured the following urls on dev tunnel 'Northwind Web API':
  https://localhost:5121 -> https://8jq6g75k-5121.eks1.devtunnels.ms/
  
```



**В копилку знаний:** подробнее о тестировании веб-сервисов с помощью туннелей разработки можно узнать на сайте [learn.microsoft.com/ru-ru/aspnet/core/test/dev-tunnels](https://learn.microsoft.com/ru-ru/aspnet/core/test/dev-tunnels).

Туннели разработки помогают быстро опубликовать локальный проект для внешнего доступа — как для одиночной работы, так и в составе команды. Чтобы использовать эту возможность безопасно, стоит внимательно относиться к раскрытию локальной среды.

## Основы тестирования безопасности

Проверка безопасности — это разновидность тестирования ПО, направленная на выявление уязвимостей, угроз и рисков в приложении. Благодаря этому удается защитить данные и ресурсы от потенциальных злоумышленников. В ходе проверки анализируются возможные недостатки системы безопасности, которые могут быть использованы для атаки, — например, слабости в проверке подлинности, авторизации, шифровании данных или обработке ошибок. Главная задача — обнаружить и устранить уязвимости, чтобы приложение могло эффективно противостоять вредоносным воздействиям и надежно защищать конфиденциальную информацию, обеспечивая целостность, доступность и конфиденциальность данных.

Для разработчика .NET проверка безопасности не должна ограничиваться заполнением контрольного списка. Важно с самого начала закладывать мышление, ориентированное на безопасность, в основу процесса разработки. Поскольку на платформе .NET можно создавать самые разные типы приложений — веб-приложения, мобильные решения, классические приложения, — охватить все аспекты тестирования безопасности в рамках одного раздела невозможно. В этом материале я сосредоточусь на основных принципах.

Профессиональный разработчик .NET должен обладать хотя бы базовыми знаниями о проверке безопасности. В частности, важно обратить внимание на следующие ключевые моменты.

- **OWASP Top 10.** Рекомендуется начать с отчета OWASP Top 10, в котором описаны наиболее значимые риски безопасности веб-приложений. Для разработчика .NET знание этих рисков особенно важно, поскольку они охватывают самые распространенные уязвимости, которые могут угрожать приложениям. Понимание сути этих угроз помогает заранее учитывать их при проектировании и разработке, снижая вероятность возникновения известных уязвимостей.
- **Практика безопасного программирования.** Проверка безопасности связана не только с поиском ошибок — она начинается с написания безопасного кода. Важно следовать проверенным практикам безопасного программирования, которые помогают предотвращать появление уязвимостей. Для разработчиков .NET это означает, в частности, необходимость безопасно обрабатывать пользовательский ввод, надежно реализовывать проверку подлинности и авторизацию, грамотно управлять сессиями и правильно шифровать конфиденциальные данные.
- **Средства безопасности .NET.** Платформа .NET предлагает широкий набор возможностей и библиотек, созданных для поддержки разработки безопасных приложений. В их числе — механизмы проверки подлинности и авторизации, средства безопасной связи (SSL/TLS), API для защиты данных и криптографические сервисы. Глубокое понимание встроенных функций помогает разработчикам использовать весь потенциал платформы для повышения безопасности своих решений. Многие из этих базовых возможностей уже рассматривались в главе 8.

- **Автоматизированные средства тестирования безопасности.** Полезно ознакомиться с инструментами автоматизированного тестирования безопасности, поддерживающими работу с приложениями .NET. Например, инструмент OWASP ZAP предлагает возможности автоматического сканирования, помогающие выявлять уязвимости в коде. Помимо этого, в экосистему .NET интегрированы средства статического анализа кода, позволяющие находить потенциальные слабые места еще на этапе разработки. Эти технологии известны как статическая проверка безопасности приложений (Static Application Security Testing, SAST) и динамическая проверка безопасности приложений (Dynamic Application Security Testing, DAST). Их можно интегрировать в конвейер CI/CD, чтобы обеспечить непрерывное сканирование и быстро получать обратную связь о состоянии безопасности кода.
- **Моделирование угроз.** Моделирование угроз представляет собой проактивный метод выявления возможных угроз безопасности и оценки связанных с ними рисков. Для разработчиков .NET моделирование угроз помогает глубже понять, какие сценарии атак могут угрожать приложению и какие меры контроля или изменения архитектуры могут снизить риски. Компания Microsoft предлагает руководство и инструменты для проведения моделирования угроз, которые особенно полезны при разработке приложений на платформе .NET.
- **Требования к соответствию.** В зависимости от области применения приложения могут действовать определенные требования к обеспечению безопасности. Например, необходимо учитывать правила GDPR для защиты персональных данных, HIPAA для медицинской информации или PCI DSS для обработки платежных данных. Понимание этих требований и их влияния на процесс разработки особенно важно для разработчиков .NET, поскольку это помогает обеспечить соблюдение нормативных стандартов и защиту конфиденциальной информации.

Теперь подробнее остановимся на отчете OWASP.

## Открытый проект обеспечения безопасности приложений

Открытый проект обеспечения безопасности приложений (Open Web Application Security Project, OWASP) – это некоммерческая организация, работающая над повышением безопасности ПО через проекты с открытым исходным кодом, координируемые сообществом. У OWASP сотни отделений по всему миру и десятки тысяч участников. Фонд также организует образовательные мероприятия и обучающие семинары.

OWASP объединяет коллективную мудрость лучших специалистов по безопасности приложений со всего мира. Благодаря этому фонд предлагает проверенные идеи, методы и решения, основанные не на теоретических моделях, а на реальных инцидентах и практическом опыте. Рекомендации OWASP отличаются высокой прикладной ценностью и применимы в реальных проектах.

Руководства и инструменты, разработанные OWASP, часто используются в качестве эталона безопасности в информационной отрасли. Для организаций следование рекомендациям OWASP помогает укрепить доверие клиентов и партнеров к уровню безопасности предоставляемых услуг и продуктов.

Рекомендации OWASP все чаще включаются в отраслевые стандарты, нормативные документы и даже в законодательство, связанное с кибербезопасностью. Поэтому их изучение и соблюдение важно не только для внедрения лучших практик, но и для выполнения требований нормативного регулирования.

## Отчет OWASP Top 10

OWASP Top 10 — это ключевой документ в области безопасности веб-приложений, в котором представлены наиболее распространенные и серьезные риски, затрагивающие веб-приложения сегодня. На момент написания книги в июне 2024 года последней актуальной версией остается список 2021 года, однако в течение 2025 года ожидается публикация новой версии. OWASP Top 10 отражает изменяющийся ландшафт угроз, выделяя как новые риски, так и традиционные уязвимости. Хотя список регулярно обновляется, например в 2017 и 2021 годах, ключевые темы остаются стабильными — это проверка ввода, проверка подлинности, контроль доступа и защита данных.



**Внимание!** Будьте внимательны при поиске информации на эту тему. Рекомендуется искать материалы по запросу «OWASP Top Ten 2021» или «OWASP Top Ten 2024», а не просто «OWASP Top 10», чтобы избежать устаревших результатов. Стоит также следить за выходом новой версии: ее публикация запланирована на сентябрь 2024 года в рамках конференции OWASP Global AppSec Conference. Дополнительную информацию можно найти на сайте [www.owasptop10.org](http://www.owasptop10.org).

Для разработчиков .NET отслеживание обновлений OWASP Top 10 имеет важное значение. Постоянное знакомство с актуальными рисками помогает защищать приложения от наиболее распространенных и опасных угроз. В следующих подразделах будет представлено краткое описание десяти наиболее значимых рисков для веб-приложений, а также даны примеры и рекомендации по их устранению в экосистеме .NET.

### Нарушение контроля доступа (A1:2021)

Нарушение контроля доступа происходит, когда пользователю предоставляется доступ к данным или функциям других пользователей без надлежащей проверки авторизации.

Для защиты приложений рекомендуется использовать встроенные механизмы авторизации, предлагаемые ASP.NET Core. Перед выполнением критически

важных операций или предоставлением доступа к конфиденциальным данным всегда следует проверять, обладает ли вошедший в систему пользователь соответствующими разрешениями. Для реализации надежной проверки подлинности можно использовать ASP.NET Core Identity, который поддерживает *многофакторную проверку подлинности* (*Multi-Factor Authentication, MFA*), а также интеграцию с внешними поставщиками, такими как OAuth.

Пароли необходимо хранить с применением безопасных алгоритмов хеширования, например bcrypt, scrypt или Argon2. ASP.NET Core Identity по умолчанию использует безопасное хеширование паролей. Этот процесс подробно рассматривался в главе 8.

Важно предоставлять пользователям только минимальный объем привилегий, необходимый для выполнения их задач. Это помогает снизить потенциальный ущерб в случае компрометации учетной записи. Следует четко определить роли и разрешения пользователей, а для защиты ресурсов и сервисов активно использовать встроенный атрибут [Authorize] в ASP.NET Core.

## Криптографические сбои (A2:2021)

Эта уязвимость, ранее известная как «Раскрытие конфиденциальных данных», возникает из-за неспособности должным образом защитить данные как в процессе передачи, так и в состоянии покоя. Типичные примеры включают хранение паролей в незашифрованном виде или использование слабых методов шифрования для защиты конфиденциальной информации.

Для защиты данных рекомендуется использовать API Data Protection, доступный в ASP.NET Core. Этот API предоставляет удобные средства для безопасного шифрования и дешифрования данных, хранящихся в приложении или базе данных.

Все передаваемые данные должны быть защищены с помощью протокола HTTPS. В ASP.NET Core можно настроить промежуточное ПО, которое автоматически перенаправляет все HTTP-запросы на HTTPS, обеспечивая безопасность данных при передаче. Для защиты данных в состоянии покоя важно применять надежные криптографические алгоритмы.

Рекомендуется также включить механизм *HTTP Strict Transport Security (HSTS)*, который принудительно заставляет браузеры использовать только защищенные соединения. Это помогает предотвратить атаки типа «человек посередине» (Man-in-the-Middle, MitM).

```
var builder = WebApplication.CreateBuilder(args);
...
var app = builder.Build();
...
app.UseHsts();
app.UseHttpsRedirection();
```

## Внедрения (A3:2021)

К категории внедрений, или инъекций, относятся уязвимости, возникающие при некорректной обработке вводимых данных, — например, внедрение SQL-кода, внедрение команд и другие похожие атаки. Один из типичных примеров — уязвимость, связанная с конкатенацией пользовательского ввода при формировании SQL-запросов.

Для защиты приложений следует использовать параметризованные запросы в Entity Framework или хранимые процедуры. При работе с ADO.NET или LINQ to Entities также важно применять параметризованные запросы, чтобы избежать риска внедрения SQL-кода. Никогда не стоит строить SQL-запросы через простую конкатенацию строк. Инструменты объектно-реляционного отображения (ORM), такие как EF Core, по умолчанию защищают от внедрения SQL-кода за счет использования параметризованных запросов при доступе к данным.

Все вводимые пользователем данные необходимо проверять и очищать. Это касается не только пользовательского ввода, но и данных, поступающих из файлов, баз данных или внешних сервисов. Следует проверять тип, длину, формат и диапазон значений. Кроме того, для защиты от атак *межсайтового скрипtingа* (*Cross-Site Scripting, XSS*) нужно использовать встроенные в .NET методы кодирования HTML и URL при обработке данных, отображаемых на страницах или включаемых в URL-адреса.

## Небезопасный дизайн (A4:2021)

Эта новая категория отражает риски, возникающие из-за недостатков проектирования приложения. Например, разработка без учета принципов безопасности с самого начала может привести к появлению фундаментальных уязвимостей.

Рекомендуется придерживаться подхода *Secure by Design*, предусматривающего внедрение безопасности на этапе проектирования. Полезно использовать моделирование угроз, а также применять проверенные принципы и шаблоны проектирования, ориентированные на безопасность, начиная с самых ранних этапов проекта.

## Неправильная конфигурация системы безопасности (A5:2021)

Эта категория охватывает риски, связанные с ошибками в настройке, — например, неправильно заданные разрешения, использование настроек по умолчанию, чрезмерно подробные сообщения об ошибках и другие уязвимости. Одним из примеров может служить раскрытие конфиденциальной информации из-за неправильной обработки ошибок в приложениях .NET.

Для защиты рекомендуется обеспечивать безопасность всех конфигураций на всех уровнях приложения. Регулярный аудит и обновление конфигураций помогают

своевременно устранять потенциальные слабости. В приложениях ASP.NET Core полезно использовать собственные страницы ошибок, чтобы предотвратить не-преднамеренное раскрытие конфиденциальной информации.

### **Уязвимые и устаревшие компоненты (A6:2021)**

Эта категория подчеркивает риски, связанные с использованием ПО, содержащего известные уязвимости. Например, к таким рискам приводит использование устаревших или неподдерживаемых версий .NET или сторонних библиотек, особенно тех, для которых уже выявлены критические проблемы безопасности.

Для снижения рисков рекомендуется регулярно обновлять версию .NET, сторонние библиотеки и все зависимости проекта. Удобно использовать такие средства, как NuGet Package Manager, для управления обновлениями. Также стоит рассмотреть использование инструментов анализа состава ПО (Software Composition Analysis, SCA) для обнаружения известных уязвимостей в зависимостях. Например, инструмент OWASP Dependency Check можно интегрировать в процесс сборки, чтобы проводить регулярное сканирование проектов.

### **Сбои идентификации и проверки подлинности (A7:2021)**

Эта категория, ранее известная как «Нарушение проверки подлинности», охватывает проблемы, связанные с реализацией проверки подлинности и управлением сессиями. Среди распространенных примеров — слабые требования к паролям, которые допускают атаки методом перебора, или недостаточная защита токенов сессии, позволяющая злоумышленнику перехватить управление учетной записью.

Для защиты приложений рекомендуется реализовать надежные механизмы проверки подлинности с помощью ASP.NET Core Identity. Следует применять строгие политики паролей, а также обеспечивать безопасное управление сессиями.

### **Сбои в работе ПО и целостности данных (A8:2021)**

Эта новая категория посвящена рискам, связанным с внесением несанкционированных изменений в ПО или данные, а также с отсутствием механизмов проверки целостности. Примером может служить использование стороннего кода или данных без предварительной проверки их целостности перед выполнением или интеграцией в приложение.

Для минимизации этих рисков рекомендуется использовать цифровые подписи или аналогичные механизмы для проверки целостности ПО, сторонних компонентов и данных.

### **Сбои в журнализации и мониторинге безопасности (A9:2021)**

Эта категория подчеркивает риски, связанные с отсутствием или недостаточностью журнализации и мониторинга событий безопасности. Например, если события безопасности не фиксируются в журналах или журналы не анализируются

на наличие подозрительной активности, возрастает риск пропустить атаки или нарушения.

Для защиты приложений рекомендуется реализовать надежное журналирование с использованием встроенных функций фреймворка ASP.NET Core. Журналы должны содержать важную для безопасности информацию и регулярно проверяться на наличие аномалий.

Следует также грамотно настраивать обработку ошибок, чтобы избежать раскрытия конфиденциальных сведений о внутреннем устройстве приложения, которые могут быть использованы злоумышленниками. При ведении журналов событий безопасности важно не записывать в них чувствительные данные. Необходимо обеспечить безопасное хранение файлов журнала и постоянно отслеживать подозрительную активность.

## Подделка запросов на стороне сервера (A10:2021)

Эта новая категория описывает риски, связанные с тем, что сервер делает запросы к непредусмотренным ресурсам или сервисам. Например, уязвимость возникает, если веб-приложение принимает URL-адрес от пользователя без должной проверки, что может привести к доступу к внутренним системам.

Для защиты рекомендуется тщательно проверять и очищать все вводимые пользователем данные, особенно URL-адреса. Следует внедрять строгие списки разрешенных направлений для исходящих запросов, чтобы предотвратить атаки подделки запросов на стороне сервера (Server-Side Request Forgery, SSRF).

Полезно также использовать представления Razor в ASP.NET Core, которые автоматически кодируют вывод данных, снижая риск XSS-атак. В дополнение стоит настроить *политику безопасности содержимого* (*Content Security Policy, CSP*), определяя, какие внешние ресурсы разрешено загружать, чтобы предотвратить внедрение вредоносного кода.

Для защиты форм от атак межсайтовой подделки запросов (Cross-Site Request Forgery, CSRF) рекомендуется использовать встроенные средства ASP.NET Core. Фреймворк автоматически проверяет анти-CSRF-токены при обработке POST-запросов.

## Краткий обзор OWASP Top 10

Для разработчиков .NET понимание и устранение десяти наиболее значимых рисков безопасности играет ключевую роль в создании надежных веб-приложений. Использование встроенных средств безопасности платформы .NET и следование лучшим практикам помогает значительно снизить уровень угроз и защитить приложения от потенциальных атак.

Применение подходов безопасного программирования позволяет разработчикам .NET создавать более защищенные приложения, надежно обрабатывать

пользовательские данные и соблюдать требования нормативных актов. Кроме того, важно постоянно следить за новыми тенденциями и угрозами в области безопасности, поскольку кибербезопасность остается быстро развивающейся сферой.



**В копилку знаний:** с последней версией OWASP Top 10 можно ознакомиться по адресу [owasp.org/Top10/](http://owasp.org/Top10/).

## Моделирование угроз

*Моделирование угроз* — важный процесс в обеспечении безопасности приложений, направленный на выявление потенциальных угроз и уязвимостей еще на ранних этапах разработки. Раннее обнаружение проблем позволяет устраниить их до развертывания приложения. Для разработчиков .NET участие в моделировании угроз предполагает систематический анализ архитектуры приложений с точки зрения безопасности, чтобы заранее определить возможные риски. Компания Microsoft предлагает специальные руководства и инструменты для моделирования угроз, которые особенно полезны в контексте работы с платформой .NET.

Ниже приведен обзор основных подходов и возможностей.

### Инструмент моделирования угроз Microsoft

Microsoft Threat Modeling Tool — один из наиболее актуальных и мощных ресурсов для разработчиков .NET. Этот бесплатный инструмент помогает выявлять и устранять угрозы безопасности еще на ранних этапах жизненного цикла разработки ПО. Он включает шаблоны для разных типов приложений, таких как веб-приложения, облачные решения и традиционные приложения. Благодаря использованию шаблонов можно выявлять распространенные угрозы, сосредоточив внимание на наиболее важных аспектах безопасности в зависимости от типа создаваемого приложения.

Инструмент позволяет строить *диаграммы потоков данных* (*Data Flow Diagram, DFD*), отражающие компоненты приложения и пути прохождения данных между ними. Такие диаграммы помогают наглядно представить, как именно данные перемещаются внутри системы, и выявить потенциальные уязвимости. На основе построенных DFD инструмент автоматически применяет набор предопределенных правил для выявления угроз безопасности, благодаря чему разработчики могут сосредоточиться на их устранении.

Хотя Microsoft Threat Modeling Tool не является специализированным решением именно для приложений .NET, его возможности интеграции с другими продуктами и сервисами Microsoft делают его особенно полезным в экосистемах, построенных на технологиях Microsoft.

## Жизненный цикл безопасной разработки (SDL)

*Жизненный цикл безопасной разработки Microsoft (Security Development Lifecycle, SDL)* представляет собой процесс, направленный на создание более безопасного ПО и выполнение требований по соблюдению стандартов безопасности при одновременном снижении затрат на разработку.

SDL предлагает четкое руководство, практические методы и набор инструментов, которые охватывают все этапы жизненного цикла разработки ПО, включая моделирование угроз.

## Ресурсы OWASP

Хотя проекты OWASP ориентированы на широкий спектр технологий и не предназначены специально для .NET, существует несколько ресурсов, которые особенно полезны для разработчиков .NET, занимающихся моделированием угроз.

Первый ресурс — *OWASP Threat Dragon*. Это инструмент моделирования угроз с открытым исходным кодом, предназначенный для работы с веб- и мобильными приложениями. Он предоставляет удобную интерактивную среду, где можно строить диаграммы потоков данных и автоматически выявлять потенциальные угрозы.

Второй важный ресурс — *OWASP Cheat Sheets*. Это набор кратких и практических рекомендаций по различным аспектам безопасности, включая моделирование угроз. Cheat Sheets помогают быстро получить четкие советы по защите приложений и лучше понять, как устранять конкретные типы угроз.

## Azure Security and Compliance Blueprints

Для приложений .NET, размещенных в Azure, компания Microsoft предлагает решение *Azure Security and Compliance Blueprints*. Этот инструмент предоставляет набор рекомендаций и средств автоматического развертывания для настройки ресурсов Azure согласно конкретным стандартам соответствия и передовым практикам безопасности. Использование таких шаблонов позволяет эффективно интегрировать моделирование угроз в процесс разработки облачных приложений.

## Передовые методы обеспечения безопасности .NET

Важным ресурсом для разработчиков .NET служит официальная документация Microsoft по передовым методам обеспечения безопасности. Она охватывает широкий спектр тем — от защиты приложений ASP.NET до работы с криптографическими API в .NET. Хотя это руководство не является отдельным инструментом, оно играет ключевую роль в понимании встроенных функций безопасности и способов их эффективного применения в реальных проектах.

Используя доступные инструменты и ресурсы, разработчики .NET могут систематически выявлять потенциальные угрозы безопасности и минимизировать риски на ранних этапах. Моделирование угроз следует включать в жизненный цикл разработки ПО как неотъемлемую часть процесса, регулярно пересматривая и актуализируя модели угроз по мере развития приложения и появления новых типов угроз.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 12.1. Дополнительные онлайн-материалы

OWASP Top 10 — это в первую очередь информационный документ. Если вы хотите использовать его в качестве стандарта программирования или тестирования, важно понимать, что он служит лишь отправной точкой.

Одной из особенностей OWASP Top 10 является то, что его цель — документировать наиболее критичные риски безопасности, а не формулировать конкретные требования, которые легко проверить в ходе тестирования. Из-за этого не все риски можно полностью обнаружить, протестировать или устранить с помощью автоматических средств. Например, риск «A04:2021 — Небезопасный дизайн» невозможно проверить инstrumentальными методами.

Если требуется полноценный стандарт для тестирования и верификации безопасности приложений, OWASP рекомендует использовать OWASP Application Security Verification Standard (ASVS). Этот стандарт разработан специально для применения на всех этапах жизненного цикла безопасной разработки. С текстом ASVS можно ознакомиться на сайте [github.com/OWASP/ASVS/blob/master/README.md](https://github.com/OWASP/ASVS/blob/master/README.md).

### Упражнение 12.2. Практические задания

Дополнительно рекомендуется изучить справочник по терминалу туннелей разработки на сайте [learn.microsoft.com/ru-ru/azure/developer/dev-tunnels/cli-commands](https://learn.microsoft.com/ru-ru/azure/developer/dev-tunnels/cli-commands), чтобы понять, как тестировать сервисы в защищенных условиях.

### Упражнение 12.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Как соотносятся между собой тестовые дублеры: имитации, шпионы, заглушки, фальшивки и пустышки?

2. Какие внешние зависимости в интеграционных тестах стоит тестировать напрямую, а какие — заменять имитациями?
3. Что такое туннель разработки и в чем его польза для .NET-разработчика?
4. С чего начать проверку безопасности .NET-приложения?
5. Что такое жизненный цикл разработки безопасного ПО Microsoft и как он применяется на практике?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 12.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-12---integration-and-security-testing](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-12---integration-and-security-testing).

## Резюме

В этой главе мы рассмотрели:

- основы интеграционного тестирования;
- методы проведения интеграционного тестирования с использованием хранилищ данных;
- тестирование сервисов с помощью туннелей разработки;
- основные концепции тестирования безопасности, включая OWASP Top 10.

В следующей главе мы поговорим о методах оценки производительности приложений. Вы узнаете, как использовать инструмент BenchmarkDotNet для оценки производительности, а также освоите базовые приемы нагружочного и стрессового тестирования с помощью таких инструментов, как Bombardier и NBomber.

# 13 Оценка производительности, нагрузочное и стресс-тестирование

Эта глава посвящена оценке производительности с помощью таких инструментов, как BenchmarkDotNet, а также нагрузочному и стресс-тестированию. Оценка производительности играет важную роль для разработчиков .NET, поскольку помогает убедиться, что создаваемый код эффективен и надежно функционирует в различных условиях.

## Оценка производительности

*Оценка производительности (бенчмаркинг)* — это процесс выявления узких мест и неэффективных участков кода, которые могут замедлять выполнение приложений. Сравнивая различные алгоритмы или стратегии в одинаковых условиях, можно определить, какой из вариантов обеспечивает лучшую производительность — например, более быстрое выполнение, меньшее потребление памяти или более эффективное использование ресурсов процессора.

Регулярная оценка производительности помогает выявить проблемы, которые могут не проявляться при обычном тестировании. В частности, это могут быть утечки памяти, ошибки конкурентного доступа при многопоточности или трудности с масштабированием при обработке больших объемов данных и высоких нагрузок.

Конкретные данные о сравнении различных реализаций позволяют принимать обоснованные решения. Независимо от того, стоит ли выбирать между разными библиотеками, фреймворками или архитектурными паттернами, оценка производительности дает надежную фактическую основу для принятия решений. Это помогает избежать преждевременной оптимизации и излишнего увлечения новыми технологиями или инструментами, которые в действительности могут не подходить для конкретного проекта. Например, важно оценить, действительно ли требуется высокоэффективный микросервис на gRPC или достаточно более простого веб-сервиса.

Для приложений, в которых скорость и эффективность имеют критическое значение — например, торговых платформ с высокой частотой операций, систем

аналитики в реальном времени или игровых проектов, — возможность быстрее обрабатывать данные и поддерживать высокую параллельность пользователей напрямую влияет на успех решений, создаваемых на платформе .NET.

Удобство работы пользователей в значимой мере зависит от производительности ПО. Пользователям не нравятся медленно загружающиеся сайты, приложения с задержками или сбои, вызванные исчерпанием ресурсов. Регулярная оценка производительности и оптимизация на основе ее результатов помогают повысить удовлетворенность пользователей.

В средах облачных вычислений, где стоимость услуг напрямую зависит от потребления ресурсов, эффективный код позволяет значительно сократить операционные расходы. Оценка производительности помогает выявить области нерационального использования ресурсов и найти способы их более экономичного применения без ущерба для работы приложений.

Регулярная оценка производительности приносит пользу не только проектам, но и вашей профессиональной квалификации. Вы не только становитесь более ценным специалистом для работодателей, но и вносите заметный вклад в успех проектов и повышение удовлетворенности пользователей. Это одна из тех областей, где вложение времени и усилий окупается многократно.

Одна из ключевых концепций в оценке производительности — запись базовых измерений. Давайте подробнее рассмотрим, почему это так важно.

## Важность базовых показателей

Запись базовых показателей играет ключевую роль в оценке производительности, поскольку задает контекст для всех последующих измерений. Без исходных данных невозможно определить, привели изменения к улучшению или ухудшению производительности.

*Базовые показатели* — это конкретный, зафиксированный набор данных, с которым затем сравниваются результаты изменений, оптимизаций и модификаций. Без такой точки отсчета невозможно объективно оценить влияние внесенных изменений: улучшилась производительность, ухудшилась или осталась прежней.

Фиксация базовых показателей позволяет создать подробный моментальный снимок работы системы или приложения в определенных условиях. Такой снимок обычно включает метрики:

- время отклика;
- пропускную способность;
- использование ресурсов, таких как процессор, память и дисковый ввод-вывод.

Выбор метрик зависит от того, какие аспекты производительности наиболее критичны для достижения ваших целей. Понимая текущий уровень производительности, можно установить реалистичные и достижимые цели для ее

улучшения. Без базовых измерений легко ошибиться в ожиданиях и поставить недостижимые задачи.

Технологические среды динамичны — код, инфраструктура и модели использования постоянно меняются. Базовые показатели, зафиксированные в начале проекта или в стабильный период, становятся точкой отсчета для мониторинга производительности со временем. Постоянное сравнение с базовыми значениями помогает выявлять регресс производительности, проблемы масштабируемости и необходимость дополнительной оптимизации.

Во многих случаях оптимизация требует инвестиций: времени разработчиков, закупки нового оборудования или использования специализированных программных инструментов. Базовые показатели предоставляют доказательства, обосновывающие такие вложения, демонстрируя конкретные улучшения. Привлечь ресурсы значительно проще, когда можно представить четкие данные о текущей производительности и обосновать ожидаемые выгоды от изменений.

Некоторые проекты подпадают под действие нормативных требований или должны соответствовать *соглашениям об уровне обслуживания* (*Service Level Agreements, SLA*). Наличие базовых показателей позволяет не только подтвердить текущее соответствие требованиям, но и контролировать влияние будущих изменений, чтобы избежать их непреднамеренного нарушения.

Запись базовых показателей можно сравнить с отслеживанием прогресса в достижении личной цели. Представьте, что вы хотите похудеть: вы занимаетесь спортом, правильно питаетесь, а через полгода становитесь на весы. Чтобы понять результат, нужно было зафиксировать исходные данные — вес и окружность талии — еще до начала усилий. Только тогда сравнение будет объективным и покажет реальный прогресс.

Понимая исходное состояние, можно принимать обоснованные решения, отслеживать изменения со временем и добиваться реальных улучшений. Базовые показатели становятся якорем, который позволяет уверенно двигаться к значимым результатам.

Когда речь заходит о производительности и сравнении различных вариантов ее повышения, полезно использовать стандартные способы описания временной и пространственной сложности задачи — то есть оценивать, сколько времени и ресурсов требуется для ее выполнения. Давайте рассмотрим несколько распространенных способов оценки сложности.

## Нотация «O большое»

*Нотация Big O*, или «*O большое*», вместе со связанными концепциями производительности лежит в основе понимания эффективности алгоритмов, особенно с точки зрения временной и пространственной сложности. Эти понятия помогают программистам и исследователям в области компьютерных наук предсказывать поведение алгоритмов как в наихудших сценариях, так и при увеличении объема входных данных.

Нотация « $O$  большое» — это математический способ описания верхней границы времени выполнения алгоритма или объема используемой им памяти в наихудшем случае. Она позволяет на высоком уровне оценить, как производительность алгоритма зависит от размера входных данных, обозначаемого как  $n$ .

Основная задача нотации « $O$  большое» — обеспечить возможность сравнения эффективности алгоритмов, абстрагируясь от постоянных множителей и факторов низшего порядка. Нотация помогает сосредоточиться на ключевых аспектах, влияющих на рост времени выполнения или использования памяти, когда масштаб данных увеличивается.

В табл. 13.1 представлены основные обозначения временной сложности с примерами алгоритмов.

**Таблица 13.1.** Описание временной сложности в нотации « $O$  большое» и примеры алгоритмов

Условные обозначения	Время выполнения	Описание	Пример
$O(1)$	Постоянное	Время выполнения не зависит от размера входных данных	Доступ к элементу массива по индексу
$O(n)$	Линейное	Время выполнения растет пропорционально размеру входных данных	Линейный поиск элемента в несортированном массиве
$O(\log n)$	Логарифмическое	Время выполнения увеличивается логарифмически при росте входных данных	Двоичный поиск в отсортированном массиве
$O(n \log n)$	Линейно-логарифмическое	Время выполнения растет быстрее, чем линейно, но медленнее, чем квадратично	Сортировка слиянием, пирамидальная сортировка
$O(n^2)$	Квадратичное	Время выполнения растет пропорционально квадрату размера входных данных	Сортировка пузырьком
$O(2^n)$	Экспоненциальное	Время выполнения удваивается с каждым новым элементом входных данных	Рекурсивный расчет чисел Фибоначчи методом полного перебора

Чтобы лучше представить, как разные типы сложности влияют на производительность при увеличении размера входных данных, в табл. 13.2 я привел примерные значения для разных функций.

**Таблица 13.2.** Примеры значений временной сложности в нотации «O большое»

<b>n</b>	<b>log n</b>	<b>n log n</b>	<b>n<sup>2</sup></b>	<b>2<sup>n</sup></b>
1	—	—	1	2
2	0,3	0,6	4	4
5	0,7	3,5	25	32
20	1,3	26,0	400	1 048 576
50	1,7	85,0	2 500	1 125 899 906 842 624

*Пространственная сложность* описывает объем памяти, необходимый алгоритму для завершения работы, выраженный как функция от размера входных данных. Подобно временной сложности, пространственную сложность также можно выразить с помощью нотации «O большое».

Учитывая пространственную сложность, разработчики могут создавать алгоритмы, которые укладываются в ограничения по объему памяти системы, что особенно важно при работе с большими наборами данных или в ограниченных средах, таких как встроенные устройства.

Кроме нотации «O большое», существуют и другие способы описания поведения алгоритмов.

- **Нотация тета ( $\Theta$ )** — описывает одновременно верхнюю и нижнюю границы времени выполнения алгоритма, давая более точную оценку его производительности. Она задает точный порядок роста времени выполнения в зависимости от размера входных данных.
- **Нотация омега ( $\Omega$ )** — указывает нижнюю границу времени выполнения, то есть минимальное время, которое потребуется алгоритму для обработки любых входных данных.

Использование этих нотаций позволяет разработчикам прогнозировать поведение алгоритмов и осознанно выбирать подходящие решения в зависимости от требований проекта и специфики задач.

## Статистические метрики

Метрики P95 и P99 широко используются для описания характеристик производительности веб-сервисов, приложений и систем, особенно в части задержки и времени отклика. Эти метрики основаны на понятии процентиляй и помогают лучше понять распределение времени обработки запросов.

- **P95 (95-й процентиль)** показывает, что 95 % всех запросов обрабатываются за время, меньшее или равное этому значению, и только 5 % — медленнее. Например, если значение P95 составляет 200 миллисекунд, это означает, что 95 % запросов завершаются за 200 миллисекунд или быстрее, а лишь 5 % запросов

требуют больше времени. Эта метрика хорошо отражает общее качество пользовательского опыта при нормальных условиях работы, поскольку исключает влияние самых редких медленных запросов.

- **P99 (99-й процентиль)** показывает, что 99 % всех запросов обрабатываются за время, меньшее или равное этому значению, а только 1 % запросов выполняется медленнее. Например, если значение P99 составляет 500 миллисекунд, это значит, что 99 % запросов укладываются в 500 миллисекунд, а 1 % выходит за эти рамки. Эта метрика особенно важна для сервисов с высокими требованиями к надежности и производительности, так как помогает выявить даже редкие случаи ухудшения пользовательского опыта.

Метрики P95 и P99 позволяют лучше оценить реальный пользовательский опыт, включая тех пользователей, которые сталкиваются с задержками выше среднего уровня. Эти процентили помогают выявлять и устранять узкие места в производительности систем. Ориентируясь на более высокие процентили, разработчики могут обнаруживать проблемы, проявляющиеся только при высокой нагрузке или в нестандартных условиях. Понимание распределения времени отклика, включая крайние значения, позволяет более точно планировать пропускную способность системы и обеспечивать ее устойчивость не только в нормальных, но и в пиковых условиях работы.

Многие соглашения об уровне обслуживания (Service Level Agreements, SLA) используют процентные показатели для установления допустимых порогов производительности. Поэтому метрики P95 и P99 играют ключевую роль в соблюдении таких требований.

Перцентили P95 и P99 дают более полное представление о поведении системы по сравнению со средним временем отклика, поскольку акцентируют внимание на тех пользователях, которые сталкиваются с наибольшими задержками. Оптимизация системы с учетом этих метрик может значительно повысить удовлетворенность пользователей и обеспечить устойчивую производительность в критически важных приложениях.

Теперь познакомимся с одним из самых популярных практических инструментов для оценки производительности.

## Оценка производительности с помощью BenchmarkDotNet

*BenchmarkDotNet* — популярный пакет NuGet для оценки производительности в .NET, активно используемый в блогах компании Microsoft, а также в публикациях в LinkedIn и тематических рассылках. Этот инструмент позволяет наглядно подтверждать эффективность различных решений и подходов к разработке на .NET.

Понимание принципов работы BenchmarkDotNet и умение использовать его для собственных тестов производительности важно для каждого разработчика .NET.

Написание тестов производительности с BenchmarkDotNet напоминает создание модульных тестов: необходимо создать класс с методами, каждый из которых представляет отдельный тест. В BenchmarkDotNet методы помечаются атрибутом `[Benchmark]`. Один из методов можно обозначить как базовый, добавив параметр `Baseline = true`, чтобы использовать его в качестве отправной точки для сравнения производительности других методов.

Рассмотрим на практике, как использовать BenchmarkDotNet для сравнения производительности между простым сложением строк (которое мы примем за базовый вариант) и использованием `StringBuilder` (который, как ожидается, покажет улучшение).

1. В редакторе кода создайте консольное приложение `BenchmarkApp` и добавьте его в решение `Chapter13`.
2. В файле проекта `BenchmarkApp` включите обработку предупреждений как ошибок, статически и глобально импортируйте класс `Console` и добавьте ссылку на пакет `BenchmarkDotNet`:

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

<ItemGroup>
  <PackageReference Include="BenchmarkDotNet" Version="0.13.12" />
</ItemGroup>
```



Вы можете проверить последнюю версию пакета BenchmarkDotNet по адресу [www.nuget.org/packages/BenchmarkDotNet](http://www.nuget.org/packages/BenchmarkDotNet).

3. Выполните сборку проекта `BenchmarkApp` для восстановления зависимостей.
4. Создайте файл класса `StringBenchmarks.cs`.
5. В этом файле добавьте код для определения класса с методами, каждый из которых будет отдельным тестом производительности. В данном примере будут реализованы два метода, объединяющие 100 чисел, разделенных запятыми, с использованием конкатенации строк и класса `StringBuilder`:

```
using BenchmarkDotNet.Attributes; // Подключение атрибута [Benchmark]

// То же пространство имен, что и в файле Program.cs

public class StringBenchmarks
{
    int[] numbers;

    public StringBenchmarks()
    {
        numbers = Enumerable.Range(
```

```
        start: 1, count: 100).ToArray();
    }

[Benchmark(Baseline = true)]
public string StringConcatenationTest()
{
    string s = string.Empty; // т.п. ""
    for (int i = 0; i < numbers.Length; i++)
    {
        s += numbers[i] + ", ";
    }

    return s;
}

[Benchmark]
public string StringBuilderTest()
{
    System.Text.StringBuilder builder = new();
    for (int i = 0; i < numbers.Length; i++)
    {
        builder.Append(numbers[i]);
        builder.Append(", ");
    }

    return builder.ToString();
}
}
```

6. В файле `Program.cs` удалите существующее содержимое. Затем импортируйте пространство имен `BenchmarkDotNet` и добавьте код для оценки производительности:

```
using BenchmarkDotNet.Running; // Подключение класса BenchmarkRunner  
BenchmarkRunner.Run<StringBenchmarks>();
```

7. Выполните запуск консольного приложения в конфигурации `Release` (Релиз):

- в Visual Studio на панели инструментов присвойте параметру `Solution Configurations` (Конфигурации решения) значение `Release` (Релиз), а затем выберите команду меню `Debug ▶ Start Without Debugging` (Отладка ▶ Запуск без отладки);
- в Visual Studio Code в терминале выполните команду `dotnet run --configuration Release`.



**Дельный совет:** при тестировании производительности всегда запускайте приложение в конфигурации релиза. Это необходимо для получения точных результатов измерений.

8. Изучите результаты выполнения программы, включая экспортированные файлы отчетов и, самое главное, сводную таблицу производительности. В примере конкатенация строк заняла в среднем 2484,6 нс, а использование

`StringBuilder` – 780,6 нс, что составляет менее одной трети времени (ниже показан фрагмент вывода):

```
// Validating benchmarks:  
// ***** BenchmarkRunner: Start *****  
// ***** Found 2 benchmark(s) in total *****  
// ***** Building 1 exe(s) in Parallel: Start *****  
// start dotnet restore /p:UseSharedCompilation=false  
/p:BuildInParallel=false /m:1 /p:Deterministic=true /p:Optimize=true in  
C:\tools-skills-net8\Chapter13\BenchmarkApp\bin\Release\net8.0\36a5f5cd053-  
416c-92e9-a04028e30b91  
...  
// ***** Done, took 00:00:18 (18.09 sec) *****  
// Found 2 benchmarks:  
//   StringBenchmarks.StringConcatenationTest: DefaultJob  
//   StringBenchmarks.StringBuilderTest: DefaultJob  
...  
// ***** BenchmarkRunner: Finish *****  
  
// * Export *  
BenchmarkDotNet.Artifacts\results\StringBenchmarks-report.csv  
BenchmarkDotNet.Artifacts\results\StringBenchmarks-report-github.md  
BenchmarkDotNet.Artifacts\results\StringBenchmarks-report.html  
  
// * Detailed results *  
StringBenchmarks.StringConcatenationTest: DefaultJob  
Runtime = .NET 8.0.2 (8.0.224.6711), X64 RyuJIT AVX-  
512F+CD+BW+DQ+VL+VBM; GC = Concurrent Workstation  
Mean = 2.485 us, StdErr = 0.012 us (0.50%), N = 18, StdDev = 0.053 us  
Min = 2.418 us, Q1 = 2.438 us, Median = 2.481 us, Q3 = 2.520 us, Max =  
2.588 us  
IQR = 0.082 us, LowerFence = 2.315 us, UpperFence = 2.644 us  
ConfidenceInterval = [2.435 us; 2.534 us] (CI 99.9%), Margin = 0.049 us  
(1.98% of Mean)  
Skewness = 0.29, Kurtosis = 1.7, MValue = 2  
----- Histogram -----  
[2.416 us ; 2.469 us) | @@@@@@@@  
[2.469 us ; 2.544 us) | @@@@  
[2.544 us ; 2.601 us) | @@  
-----  
StringBenchmarks.StringBuilderTest: DefaultJob  
Runtime = .NET 8.0.2 (8.0.224.6711), X64 RyuJIT AVX-  
512F+CD+BW+DQ+VL+VBM; GC = Concurrent Workstation  
Mean = 780.606 ns, StdErr = 3.884 ns (0.50%), N = 63, StdDev = 30.832 ns  
Min = 734.198 ns, Q1 = 768.066 ns, Median = 774.160 ns, Q3 = 783.365 ns,  
Max = 944.730 ns  
IQR = 15.299 ns, LowerFence = 745.118 ns, UpperFence = 806.313 ns  
ConfidenceInterval = [767.187 ns; 794.025 ns] (CI 99.9%), Margin = 13.419 ns  
(1.72% of Mean)  
Skewness = 2.96, Kurtosis = 14.79, MValue = 2  
----- Histogram -----  
[724.028 ns ; 744.242 ns) | @  
[744.242 ns ; 764.820 ns) | @@@@@@@@  
[764.820 ns ; 785.160 ns) | @@@@@@@@  
[785.160 ns ; 817.544 ns) | @@@@  
[817.544 ns ; 837.885 ns)
```

```
[837.885 ns ; 874.367 ns) | @@
[874.367 ns ; 894.707 ns) |
[894.707 ns ; 915.047 ns) |
[915.047 ns ; 934.560 ns) |
[934.560 ns ; 954.900 ns) | @

// * Summary *

BenchmarkDotNet v0.13.12, Windows 11 (10.0.22631.3235/23H2/2023Update/
SunValley3)
11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and 4 physical cores
.NET SDK 9.0.100-preview.1.24101.2
[Host] : .NET 8.0.2 (8.0.224.6711), X64 RyuJIT AVX-512F+CD+BW+DQ+VL+VBMI
DefaultJob : .NET 8.0.2 (8.0.224.6711), X64 RyuJIT AVX-512F+CD+BW+DQ+VL+VBMI

| Method | Mean | Error | StdDev | Ratio | RatioSD |
|-----:|-----:|-----:|-----:|-----:|-----:|
| StringConcatenationTest | 2,484.6 ns | 49.14 ns | 52.58 ns | 1.00 | 0.00 |
| StringBuilderTest | 780.6 ns | 13.42 ns | 30.83 ns | 0.32 | 0.02 |
// * Hints *
Outliers
StringBenchmarks.StringBuilderTest: Default -> 18 outliers were removed
(1.00 us..1.47 us)
// * Legends *
Mean : Arithmetic mean of all measurements
Error : Half of 99.9% confidence interval
StdDev : Standard deviation of all measurements
Ratio : Mean of the ratio distribution ([Current]/[Baseline])
RatioSD : Standard deviation of the ratio distribution ([Current]/
[Baseline])
1 ns : 1 Nanosecond (0.000000001 sec)

// ***** BenchmarkRunner: End *****
Run time: 00:01:44 (104.27 sec), executed benchmarks: 2

Global total time: 00:02:02 (122.6 sec), executed benchmarks: 2
// * Artifacts cleanup *
Artifacts cleanup is finished
```



Ваши результаты могут отличаться в зависимости от обеспечения компьютера и окружения. Разделы Hints и Outliers могут отсутствовать, если во время выполнения не будет обнаружено выбросов!

Теперь сравним производительность разных способов расчета чисел Фибоначчи.

1. В проекте `BenchmarkApp` создайте файл класса `FibonacciBenchmarks.cs`.
2. В этом файле добавьте код для определения класса с методами, каждый из которых будет отдельным тестом производительности:

```
using BenchmarkDotNet.Attributes; // Подключение атрибута [Benchmark]

public class FibonacciBenchmarks
{
    private const long n = 10;
```

```

public long RecursiveFibonacci(long n)
{
    if (n <= 0) return 0;
    if (n == 1) return 1;

    return RecursiveFibonacci(n - 1) + RecursiveFibonacci(n - 2);
}

[Benchmark(Baseline = true)]
public long RecursiveTest()
{
    return RecursiveFibonacci(n);
}

public static long BinetsFibonacci(long n)
{
    double sqrt5 = Math.Sqrt(5);
    double phi = (1 + sqrt5) / 2;
    double psi = (1 - sqrt5) / 2;

    return (long)((Math.Pow(phi, n) - Math.Pow(psi, n)) / sqrt5);
}

[Benchmark]
public long BinetsTest()
{
    return BinetsFibonacci(n);
}
}

```



**В копилку знаний:** формула Бине позволяет напрямую вычислять  $n$ -е число Фибоначчи без использования рекурсии, что значительно повышает производительность. В этой формуле используются золотое сечение (обозначаемое буквой  $\phi$ , фи) и его сопряженная величина  $\psi$  (пси). Подробнее об этом можно прочитать на сайте [artofproblemsolving.com/wiki/index.php/Binet%27s\\_Formula](http://artofproblemsolving.com/wiki/index.php/Binet%27s_Formula).

3. В файле `Program.cs` закомментируйте код для запуска тестов производительности для строк и добавьте код для запуска тестов производительности для алгоритмов Фибоначчи:

```
// BenchmarkRunner.Run<StringBenchmarks>();
BenchmarkRunner.Run<FibonacciBenchmarks>();
```

4. Выполните запуск консольного приложения в конфигурации **Release** (Релиз).
  - В Visual Studio: на панели инструментов присвойте параметру **Solution Configurations** (Конфигурации решения) значение **Release** (Релиз), а затем выберите команду меню **Debug ▶ Start Without Debugging** (Отладка ▶ Запуск без отладки).
  - В Visual Studio Code: в терминале выполните команду `dotnet run --configuration Release`.

5. Изучите результаты выполнения тестов. Так, рекурсивный метод в среднем занимает 155,22 нс, а метод Бине — 40,34 нс, что составляет примерно четверть времени (ниже показан фрагмент вывода):

Method	Mean	Error	StdDev	Ratio
RecursiveTest	155.22 ns	2.940 ns	2.750 ns	1.00
BinetsTest	40.34 ns	0.385 ns	0.360 ns	0.26

## Предотвращение ошибок при оценке производительности

Как и в случае с любым другим инструментом, эффективность инструмента BenchmarkDotNet зависит от того, насколько корректно он используется.

### Изоляция кода теста производительности от кода настройки/завершения выполнения

Одна из распространенных ошибок — не изолировать код, который тестируется, от логики подготовки и завершения выполнения. Пакет BenchmarkDotNet предоставляет атрибуты `[GlobalSetup]` и `[GlobalCleanup]`, позволяющие отделить код настройки и очистки от собственно тестируемого кода. Неправильное использование этих атрибутов может привести к тому, что время настройки будет включено в результаты теста производительности, что исказит итоговые показатели.

Рассмотрим пример правильного использования атрибутов `[GlobalSetup]` и `[GlobalCleanup]`.

1. В проекте `BenchmarkApp` добавьте файл класса `SortingBenchmarks.cs`.
2. В этом файле определите класс, который будет измерять производительность методов сортировки массива. Мы сравним два разных подхода: `Array.Sort` и пользовательскую реализацию пузырьковой сортировки:

```
using BenchmarkDotNet.Attributes; // Подключение атрибутов [GlobalSetup] и т. п.

public class SortingBenchmarks
{
    private int[] data = null!; // Отключение предупреждения о null-значениях
                                // для данных
    private int[] dataArraySort = null!;
    private int[] dataBubbleSort = null!;
    [GlobalSetup]
    public void Setup()
    {
        // Инициализация массива случайными значениями
        data = new int[1000];
        for (int i = 0; i < data.Length; i++)
        {
            data[i] = Random.Shared.Next(0, 10000);
        }
    }
}
```

```

// Клонирование массива данных перед каждой сортировкой
dataArraySort = (int[])data.Clone();
dataBubbleSort = (int[])data.Clone();
}

[Benchmark]
public void ArraySort()
{
    Array.Sort(dataArraySort);
}

[Benchmark]
public void BubbleSort()
{
    BubbleSort(dataBubbleSort);
}

private void BubbleSort(int[] array)
{
    int n = array.Length;
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (array[j] > array[j + 1])
            {
                // Смена местами элементов массива array[j] и array[j + 1]
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}

[GlobalCleanup]
public void Cleanup()
{
    // Очистка ресурсов после выполнения теста
    data = [];
    dataArraySort = [];
    dataBubbleSort = [];
}
}

```

Пояснения к коду.

- Метод `Setup`, помеченный атрибутом `[GlobalSetup]`, инициализирует массив данных случайными значениями. Это обеспечивает одинаковые начальные условия для каждого запуска теста производительности.
- Метод `ArraySort` тестирует производительность метода `Array.Sort`. Перед каждым запуском он клонирует исходный массив, чтобы гарантировать одинаковые условия сортировки.
- Метод `BubbleSort` тестирует пользовательскую реализацию пузырьковой сортировки. Он также использует клонирование массива для каждого запуска.

- Метод `Cleanup`, помеченный атрибутом `[GlobalCleanup]`, сбрасывает массив данных в `null`. В этом примере очистка минимальна, но на практике здесь можно освободить любые ресурсы, которые необходимо очистить после выполнения тестов.
3. В файле `Program.cs` настройте запуск теста производительности:

```
BenchmarkRunner.Run<SortingBenchmarks>();
```

4. Запустите проект и обратите внимание на результаты. Вы увидите, что метод `ArraySort` примерно в 122 раза быстрее, чем `BubbleSort`, как показано ниже:

Method	Mean	Error	StdDev
ArraySort	2.963 us	0.0569 us	0.0533 us
BubbleSort	363.691 us	7.1806 us	7.0523 us

Такой подход обеспечивает точность результатов тестов производительности, отражая реальную эффективность тестируемых методов без искажений, связанных с логикой подготовки или очистки данных.

## Почему важно полагаться на BenchmarkDotNet

Компилятор *Just-In-Time (JIT)* оптимизирует код во время выполнения, что может повлиять на результаты тестов производительности, если этот процесс не учитывать. Пакет BenchmarkDotNet автоматически управляет эффектами JIT-компиляции, устраняя влияние первых запусков и прогрева. Попытки вручную вмешиваться в процесс или неполное понимание работы JIT могут привести к неправильным выводам о производительности кода. Лучше полагаться на встроенные механизмы инструмента.

По умолчанию BenchmarkDotNet выполняет достаточно итераций для получения статистически значимых результатов. Изменение настроек по умолчанию без полного понимания последствий, например уменьшение количества итераций для ускорения тестирования, может привести к недостоверным данным из-за недостаточной репрезентативности выборки или нестабильности времени выполнения.

## Корректное сравнение результатов тестов производительности

Сравнение результатов тестов производительности, полученных на разных компьютерах, в различных средах или конфигурациях, может привести к ошибочным выводам. Пакет BenchmarkDotNet позволяет экспорттировать и сравнивать результаты тестов, но важно удостовериться, что сравнение проводится в одинаковых условиях и что различия в оборудовании или настройках среды учтены.

Отсутствие управления версиями результатов тестов производительности затрудняет отслеживание изменений производительности со временем. BenchmarkDotNet поддерживает экспорт результатов в различные форматы, такие как Markdown и CSV, которые рекомендуется отслеживать в системе управления версиями наряду с кодом для документирования улучшений или ухудшений.

Запуск тестов производительности на отладочных сборках может привести к недостоверным результатам из-за отсутствия оптимизаций компилятора. Чтобы получить корректные измерения, всегда используйте релиз-сборки и запускайте тесты в условиях, максимально приближенных к производственной среде.

## Учет влияния среды на результаты тестов

Результаты тестов производительности могут варьироваться из-за статистического шума: фоновых процессов, особенностей планирования потоков в операционной системе, различий в аппаратной платформе и других факторов. Важно учитывать естественную изменчивость измерений и избегать поспешных выводов на основе небольших различий, которые могут находиться в пределах допустимой погрешности.

## Выявление неточных публикаций о производительности

Сегодня в социальных сетях, таких как LinkedIn, все чаще можно встретить публикации, посвященные вопросам производительности. Однако некоторые из них могут содержать некорректные выводы из-за неправильного использования пакета BenchmarkDotNet или других инструментов оценки производительности. Давайте разберемся, на что стоит обращать внимание, чтобы выявить такие ошибки.

При чтении публикаций в LinkedIn, технических блогах или на форумах обратите внимание на следующие признаки.

- **Отсутствие контекста:** если в сообщении не указаны условия тестирования — спецификации оборудования, версия среды выполнения или операционной системы, — доверять результатам нельзя. Без этих данных невозможно оценить применимость результатов к другим средам.
- **Неуказанная версия BenchmarkDotNet:** BenchmarkDotNet активно развивается, и производительность самих тестов может изменяться между версиями. Для прозрачности и воспроизводимости авторы должны указывать, какую версию инструмента они использовали.
- **Игнорирование статистической значимости:** BenchmarkDotNet предоставляет статистический анализ прогонов, включая доверительные интервалы. Если в публикации заявляется о превосходстве одного метода над другим без обсуждения статистической значимости, такие выводы могут быть недостоверными.
- **Сравнение различных рабочих нагрузок:** убедитесь, что методы действительно выполняют сопоставимую работу. Распространенная ошибка — сравнивать методы, которые выглядят похожими, но обрабатывают разные объемы данных или по-разному справляются с крайними случаями.
- **Игнорирование влияния холодного старта:** первое выполнение метода может быть медленнее из-за JIT-компиляции и других накладных расходов на инициализацию. Авторы публикаций, не учитывающие этот эффект и не предусматривающие период прогрева, могут неточно отражать реальную производительность метода.

- **Сосредоточение исключительно на скорости:** оптимизация часто связана с компромиссами в использовании памяти, удобстве сопровождения или читабельности кода. Авторы публикаций, акцентирующие внимание только на скорости выполнения без обсуждения этих аспектов, дают неполную картину.
- **Небольшие размеры выборки:** BenchmarkDotNet позволяет настраивать количество итераций тестирования и прогрева. Результаты, полученные на основе слишком малого числа итераций, могут быть нестабильными и менее показательными.
- **Выборочное представление результатов:** остерегайтесь публикаций, в которых демонстрируются только отдельные случаи, выгодные для одного метода, без полного контекста тестирования, включая ситуации, где другой метод может показывать лучшие результаты.
- **Редко выполняемые рабочие нагрузки:** даже если все остальное сделано правильно, оптимизация кода, который практически никогда не выполняется в вашем проекте, теряет смысл. Один из распространенных примеров — сравнение производительности преобразования значения `enum` в строку с помощью методов `ToString()` и `nameof()`. Метод `nameof()` значительно быстрее, поскольку преобразование происходит на этапе компиляции, а не во время выполнения.

При оценке публикаций о производительности ищите сообщения, где приведено полное описание тестов: настройка окружения, параметры конфигурации и глубокий анализ результатов. Критическое мышление и здоровый скептицизм помогут определить, действительно ли сравнение производительности полезно или же оно вводит в заблуждение.

Настоящим экспертом в области производительности является Фэн Юань — бывший архитектор производительности в Microsoft, автор блога FrugalTips и признанный специалист в этой области. В последнее время он ушел на пенсию, чтобы разводить коз на своей ферме (да, действительно!), но продолжает активно делиться экспертными комментариями на LinkedIn, указывая на ошибки в постах о производительности и объясняя, где авторы допускают неточности. Если вы зарегистрированы на платформе LinkedIn, рекомендую следить за его профилем: [www.linkedin.com/in/dryuan/](http://www.linkedin.com/in/dryuan/). Вы также можете найти его статьи на сайте Frugal Cafe: [frugalcafe.beehiiv.com](http://frugalcafe.beehiiv.com).



**Дельный совет:** если вы планируете публиковать собственные статьи о тестировании производительности, ориентируйтесь на материалы Стивена Тоба. Его посты служат отличными примерами аккуратного, взвешенного подхода. Последнюю публикацию можно найти на сайте [devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/](http://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/).

Теперь перейдем к нагрузочному и стресс-тестированию — двум важным разновидностям тестирования производительности.

## Нагрузочное и стресс-тестирование

*Нагрузочное тестирование* — это разновидность тестирования производительности, предназначенная для оценки поведения системы при нормальной и ожидаемой пиковой нагрузке. Оно помогает выявить узкие места и понять, как система справляется с максимальным объемом запросов. Проведение нагрузочного тестирования позволяет убедиться, что приложение сможет выдержать повышенную нагрузку, сохранив положительный пользовательский опыт.

Одной из ключевых целей нагрузочного тестирования является проверка способности приложения масштабироваться для поддержки ожидаемого максимального числа пользователей. Это требует эффективного распределения ресурсов и обеспечения стабильной производительности даже при увеличении нагрузки.

Обычно производительность системы измеряется при различных уровнях нагрузки, как показано в табл. 13.3.

**Таблица 13.3.** Различные типы нагрузки

Нагрузка	Описание
Типичная нагрузка	Средняя ожидаемая нагрузка в обычный день
Пиковая нагрузка	Максимальная ожидаемая пользовательская нагрузка. Например, 1000 запросов в секунду к главной странице или 100 оформлений заказов в секунду
Пиковая нагрузка +20 %	Максимальная ожидаемая нагрузка с добавлением запаса для непредвиденных ситуаций. Этот тест позволяет понять, как система ведет себя при превышении ожидаемой нагрузки: насколько снижается пропускная способность и как это влияет на пользователей. Например, если только некоторые пользователи сталкиваются с тайм-аутами, а основная масса запросов обрабатывается успешно, это допустимая ситуация. Если же каждый пользователь испытывает проблемы, может потребоваться пересмотр архитектуры приложения
Перегрузка	Нагрузка, при которой система уже не способна поддерживать требуемую пропускную способность. Нагрузка постепенно увеличивается сверх пикового уровня до достижения предела производительности, после чего фиксируется максимальное значение. Эта процедура также известна как стресс-тестирование

Нагрузочное тестирование, имитирующее реальное использование, помогает убедиться, что ваше приложение остается стабильным и надежным в различных условиях. Оно позволяет выявить проблемы, которые могут не проявляться на этапе разработки или при стандартном тестировании, — например, состояния гонки,

утечки памяти или блокировки в базах данных, способные привести к простоям или ухудшению пользовательского опыта.

Еще один важный аспект нагрузочного тестирования — обнаружение узких мест в системе. Проблемы могут возникать в коде приложения, базе данных, сетевой инфраструктуре или аппаратных ресурсах. Понимание того, где именно находятся эти узкие места, позволяет разработчикам вносить целенаправленные улучшения для повышения общей производительности и надежности системы.

Для предприятий, которые сталкиваются с сезонными всплесками трафика или планируют маркетинговые кампании с ожидаемым ростом посещаемости, нагрузочное тестирование помогает подготовиться заранее. Проведенные тесты позволяют минимизировать риск сбоев в критические периоды и обеспечивают устойчивую работу приложения при повышенной нагрузке.

*Стресс-тестирование*, в отличие от нагрузочного тестирования, направлено на проверку устойчивости приложения и его способности обрабатывать ошибки в экстремальных условиях. Оно предназначено для доведения системы до предела ее возможностей — и даже за пределы, — чтобы увидеть, как она ведет себя при перегрузке и как восстанавливается после возвращения к нормальным условиям.

Основное внимание при стресс-тестировании уделяется стабильности и процессу восстановления. Тестирование включает подачу нагрузки и данных, выходящих за рамки нормальной эксплуатации, часто вплоть до полного сбоя системы. Это позволяет определить, как именно происходит отказ и насколько корректно система реагирует на такие ситуации.

Данные, собранные в ходе стресс-тестирования, помогают установить реальные пределы возможностей приложения. Оно показывает, насколько плавно система деградирует, как она справляется с ошибками и насколько быстро восстанавливает нормальную работу после снятия перегрузки. Это особенно важно для приложений, где сбои могут иметь серьезные последствия.

- **Нагрузочное тестирование** помогает подтвердить, что система способна справляться с ожидаемыми объемами трафика, поддерживая стабильную производительность и масштабируемость.
- **Стресс-тестирование** позволяет определить условия выхода системы из строя, сценарии деградации и признаки надвигающихся сбоев, которые можно использовать для раннего предупреждения и предотвращения критических ситуаций.

Нагрузочное и стресс-тестирование являются важной частью комплексной стратегии тестирования производительности. Эти подходы дают разные, но взаимодополняющие представления о поведении приложения: его производительности,

стабильности и устойчивости. Нагрузочное тестирование позволяет убедиться, что приложение способно справляться с ожидаемыми объемами запросов и поддерживать надежную работу в обычных условиях. Стress-тестирование помогает подготовить систему к непредвиденным ситуациям, выявляя пределы возможностей и обеспечивая минимизацию последствий при возможных сбоях. Вместе эти методы позволяют создавать надежные, эффективные и удобные приложения, которые уверенно работают как при нормальной, так и при экстремальной нагрузке.

Теперь рассмотрим несколько инструментов, которые помогают автоматизировать нагрузочное и стресс-тестирование.



**Внимание!** Ранее среди разработчиков .NET для проведения тестирования производительности и нагрузки веб-приложений непосредственно в Visual Studio пользовался популярностью инструмент Visual Studio Load Test. С его помощью можно было тестировать веб-приложения, сервисы и ресурсы Azure. Несмотря на то что данный инструмент устарел, вы все еще можете встретить ссылки на него в старых материалах. Для новых проектов сотрудники корпорации Microsoft рекомендуют использовать альтернативные решения, такие как Apache JMeter.

## Apache JMeter

Инструмент тестирования Apache JMeter работает на платформе Java, поэтому он кросс-платформенный, так же как и .NET. Тесты, созданные с помощью JMeter, можно интегрировать в конвейеры непрерывной интеграции и развертывания (CI/CD). Для проектов .NET это означает возможность автоматизировать тестирование производительности на этапе сборки и развертывания, используя такие инструменты, как Jenkins, TeamCity или Azure DevOps.

Интеграция тестов производительности в конвейер разработки помогает выявлять проблемы на ранних этапах, упрощая их устранение до того, как они смогут повлиять на рабочие среды.

Разработчики .NET, создающие веб-приложения на основе фреймворка ASP.NET Core, могут использовать JMeter для имитации высокой нагрузки на свои приложения. Инструмент позволяет моделировать одновременный доступ множества пользователей, отправляя различные HTTP-запросы и анализируя ответы серверов. Это помогает оценить, как приложение справляется с высокой нагрузкой и соответствует ли оно целевым показателям производительности.

Кроме того, JMeter можно применять для тестирования производительности SQL-запросов и хранимых процедур. Моделируя параллельные обращения к базе данных, разработчики могут выявлять медленно выполняющиеся запросы и транзакции, которые в будущем могут стать узкими местами в производственной среде.

Apache JMeter — это проект с открытым исходным кодом, доступный бесплатно. Он поддерживается активным сообществом и имеет обширную документацию, что облегчает начало работы.

Большинство моих читателей не хотят устанавливать Java на свой компьютер, поэтому в книге мы не будем подробно рассматривать примеры использования Apache JMeter. Вместо этого мы познакомимся с альтернативами, включая легкий инструмент командной строки Bombardier.



**В копилку знаний:** подробнее об инструменте Apache JMeter можно узнать по адресу [jmeter.apache.org](http://jmeter.apache.org).

## Bombardier — быстрый кросс-платформенный инструмент для оценки производительности HTTP

*Bombardier* — это высокопроизводительный, кросс-платформенный инструмент для оценки производительности HTTP, написанный на языке Go. Он предназначен для генерации значительной нагрузки на веб-серверы и сервисы при минимальном влиянии на саму систему. Bombardier способен генерировать большое количество запросов в секунду, идеально подходя для стресс-тестирования и нагрузочного тестирования веб-приложений и позволяя понять, как они ведут себя в условиях интенсивного трафика.

Ключевые особенности Bombardier.

- **Поддержка конкурентности.** Инструмент позволяет моделировать множество одновременных подключений к веб-сервису, что помогает оценить, как приложение справляется с параллельными запросами.
- **Быстрый HTTP-движок.** Bombardier способен быстро отправлять большое количество запросов, что делает его отличным выбором для создания высокой нагрузки.
- **Кросс-платформенность.** Благодаря тому что Bombardier написан на языке Go, он может быть скомпилирован и запущен на различных операционных системах — Windows, Linux и macOS. Поэтому он так удобен для разработчиков .NET независимо от используемой платформы.
- **Настраиваемые запросы.** Поддерживается настройка заголовков, полезной нагрузки и HTTP-методов, что позволяет создавать более точные и разнообразные сценарии тестирования.
- **Подробная статистика.** После выполнения тестов Bombardier предоставляет подробные данные о количестве запросов, частоте успешных ответов, задержках и пропускной способности, помогая получить ценные сведения о характеристиках производительности веб-сервиса.

Теперь рассмотрим основные шаги по установке и использованию Bombardier, а затем перейдем к практическому примеру.

## Применение инструмента Bombardier

Bombardier можно загрузить в виде предварительно скомпилированных двоичных файлов или собрать из исходного кода, доступного в репозитории GitHub. На некоторых платформах доступна установка через менеджеры пакетов.

Перед началом тестирования убедитесь, что ваш веб-сервис развернут в среде, максимально приближенной к производственной. Если точная копия недоступна, можно использовать локальный компьютер, предварительную среду или уменьшенную версию основной инфраструктуры.

Тесты с помощью Bombardier запускаются через терминал, указывая целевой URL-адрес и настраивая параметры командной строки для задания характера нагрузки.

Например, чтобы отправить 10 000 запросов при 100 одновременных соединениях, выполните команду:

```
bombardier -c 100 -n 10000 http://yourwebservice.com/api/resource
```

После завершения теста проанализируйте результаты, предоставленные Bombardier. Основные метрики, на которые стоит обратить внимание:

- количество запросов в секунду (RPS);
- средняя задержка;
- процент успешных запросов.

Эти показатели помогут выявить узкие места, оценить пределы пропускной способности и принять решения о масштабировании системы.

Bombardier удобно использовать для итеративного тестирования различных аспектов веб-сервиса — например, изменяя полезную нагрузку запроса, заголовки или HTTP-методы. Такой подход позволяет обнаруживать конкретные области, требующие оптимизации.

Для более сложных сценариев стоит рассмотреть возможность интеграции Bombardier в конвейеры CI/CD. Автоматический запуск тестов производительности после каждого развертывания или обновления помогает оперативно выявлять регрессии и поддерживать высокий уровень качества на всех этапах разработки.

## Загрузка и установка инструмента Bombardier

Установим инструмент Bombardier.

1. Перейдите на страницу [github.com/codesenberg/bombardier/releases](https://github.com/codesenberg/bombardier/releases). Я использовал версию 1.2.6, выпущенную в апреле 2023 года.
2. Загрузите подходящий исполняемый файл для вашей платформы. Например, для Windows x64 скачайте файл `bombardier-windows-amd64.exe`.

3. Переименуйте загруженный файл в `bombardier.exe`.
4. При желании добавьте каталог с файлом `bombardier.exe` в системный путь, чтобы запускать Bombardier из любой локации.
5. В терминале проверьте, что Bombardier установлен корректно, выполнив команду для запроса справки:

```
bombardier --help
```



Дополнительную справочную информацию можно получить с помощью команд `--help-long` и `--help-man`.

Рассмотрим наиболее важные переключатели (флаги) командной строки Bombardier, приведенные в табл. 13.4.

**Таблица 13.4.** Основные переключатели командной строки Bombardier

Переключатель	Описание
<code>-c, --connections &lt;целое_число&gt;</code>	Максимальное количество одновременных соединений
<code>-d, --duration &lt;продолжительность&gt;</code>	Продолжительность теста. Например, <code>10s</code> означает десять секунд
<code>-n, --requests &lt;целое_число&gt;</code>	Общее количество запросов
<code>-r, --rate &lt;целое_число&gt;</code>	Ограничение количества запросов в секунду
<code>-t, --timeout &lt;продолжительность&gt;</code>	Тайм-аут для сокета или запроса. Например, <code>2s</code>
<code>--http1</code>	Использование клиента net/http с принудительным режимом HTTP/1.x
<code>--http2</code>	Использование клиента net/http с включенной поддержкой HTTP/2.0
<code>-m, --method &lt;строка&gt;</code>	HTTP-метод запроса, например <code>GET</code> или <code>POST</code>
<code>-H, --headers &lt;строка&gt;</code>	Указание HTTP-заголовков для использования (можно указывать несколько раз)
<code>-b, --body &lt;строка&gt;</code>	Тело запроса
<code>-f, --body-file &lt;путь&gt;</code>	Файл, содержащий тело запроса
<code>-l, --latencies</code>	Вывод подробной статистики задержек
<code>-p, --print &lt;строка&gt;</code>	Тип выводимых данных: <code>intro</code> (сокращенно <code>i</code> ), <code>progress</code> (сокращенно <code>p</code> ) или <code>result</code> (сокращенно <code>r</code> ), через запятую
<code>-o, --format &lt;строка&gt;</code>	Формат вывода результатов: <code>plain-text</code> (сокращенно <code>pt</code> ) или <code>json</code> (сокращенно <code>j</code> ). Можно также указать путь к пользовательскому шаблону на основе синтаксиса Go

Для тестирования и сравнения с помощью Bombardier нам понадобятся проекты. Недавно команда .NET добавила отличную возможность для разработки веб-сервисов — встроенную поддержку компиляции Native AOT (Ahead-Of-Time). Давайте с помощью Bombardier изучим, насколько веб-сервис, собранный с использованием АОТ, эффективнее традиционного.

## Сравнение веб-сервисов с компиляцией АОТ и без нее

Технология Native AOT позволяет создавать приложения и сервисы, заранее скомпилированные в нативный код. Так сокращается время запуска и потенциально снижается объем памяти, что особенно полезно в сценариях с большим числом экземпляров — например, при масштабировании микросервисов, которые часто останавливаются и перезапускаются.

В отличие от стандартного подхода, где промежуточный код (IL) компилируется в нативный код в момент выполнения с помощью JIT-компилятора, Native AOT выполняет эту компиляцию во время публикации. Но насколько такой подход действительно улучшает производительность?

Чтобы выяснить это, создадим два простых погодных веб-сервиса: один будет собран обычным способом, а второй — с использованием компиляции АОТ. Оба проекта будут настроены только для работы с HTTP без поддержки Swagger/OpenAPI, чтобы минимизировать сложность.

1. В редакторе кода создайте проект веб-API, указав следующие настройки:
  - выберите шаблон проекта ASP.NET Core Web API (Веб-API ASP.NET Core) / webapi --no-https --no-openapi;
  - в поле Solution name (Имя решения) укажите значение Chapter13;
  - в поле Project name (Имя проекта) укажите значение Northwind.WebApi;
  - в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
  - снимите флажок Configure for HTTPS (Настроить для HTTPS);
  - снимите флажок Enable Docker (Включить поддержку контейнера);
  - снимите флажок Enable OpenAPI support (Включить поддержку OpenAPI);
  - снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
  - снимите флажок Use controllers (Использовать контроллеры);
  - если вы используете терминал, перейдите в каталог Chapter13 и выполните следующую команду:

```
dotnet new webapi --no-https --no-openapi -o Northwind.WebApi
```



**Внимание!** Программа Rider на момент чтения книги может не поддерживать создание проектов веб-API на базе минимальных API. В этом случае рекомендуется создать проект с помощью команды `dotnet new`, а затем вручную добавить его в решение Rider.

2. Создайте второй проект веб-сервиса с аналогичными параметрами, но с именем `Northwind.WebApiAot`. Этот проект будет вручную преобразован для применения компиляции Native AOT.

Если вы используете терминал, перейдите в каталог `Chapter13` и выполните следующую команду:

```
dotnet new webapi --no-https --no-openapi -o Northwind.WebApiAot
```

3. В файле `Northwind.WebApiAot.csproj` проекта включите публикацию с использованием АОТ, добавив следующую строку (выделена):

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

<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <Nullable>enable</Nullable>
  <ImplicitUsings>enable</ImplicitUsings>
  <PublishAot>true</PublishAot>
</PropertyGroup>

</Project>
```

4. В файле `Program.cs` проекта `Northwind.WebApiAot` внесите следующие изменения.

- Импортируйте пространство имён для работы с сериализацией JSON.
- Замените вызов метода `CreateBuilder` на `CreateSlimBuilder`.
- Зарегистрируйте контекст сериализатора JSON.
- Закомментируйте строку, перенаправляющую трафик на HTTPS (`app.UseHttpsRedirection()`).
- В нижней части файла определите контекстный класс для сериализатора JSON.

```
using System.Text.Json.Serialization; // Подключение класса
JsonSerializerContext.
```

```
var builder = WebApplication.CreateSlimBuilder(args);
// Добавление сервисов в контейнер.
builder.Services.ConfigureHttpJsonOptions(options =>
{
  options.SerializerOptions.TypeInfoResolverChain.Insert(0,
```

```

        AppJsonSerializerContext.Default);
});

var app = builder.Build();

// Настройка конвейера HTTP-запросов

// app.UseHttpsRedirection();

var summaries = new[]
{
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm",
    "Balmy", "Hot", "Sweltering", "Scorching"
};

app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)])
        ))
        .ToArray();
    return forecast;
});

app.Run();

internal record WeatherForecast(DateOnly Date, int TemperatureC, string?Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}

[JsonSerializable(typeof(WeatherForecast[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}

```



Метод CreateSlimBuilder не поддерживает спецификации HTTPS и HTTP/3, хотя их можно добавить при необходимости. Он также поддерживает работу с конфигурацией через файлы appsettings.json и встроенное журналирование.

5. В папке проекта Northwind.WebApi выполните публикацию веб-сервиса (через терминал):

```
dotnet publish
```

6. Откройте папку `bin\Release\net8.0\publish` на панели File Explorer (Обозреватель файлов). Вы увидите, что итоговый EXE-файл имеет размер около 140 Кбайт. Для развертывания веб-сервиса достаточно этого EXE-файла и соответствующей DLL. Файл `appsettings.json` требуется только для переопределения настроек, а файл PDB — исключительно для отладки.



Скомпилированная сборка не зависит от операционной системы.

7. В папке проекта `Northwind.WebApiAot` опубликуйте веб-сервис с включенной компиляцией Native AOT (через терминал):

```
dotnet publish
```



Обратите внимание, что процесс компиляции проекта с АОТ займет значительно больше времени.



**Внимание!** Если вы используете Rider, для включения поддержки АОТ необходимо установить пакет Build Tools for Visual Studio. Иначе при публикации вы получите ошибку `error: Platform linker not found`. Установить необходимые инструменты можно по адресу [visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2022](https://visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2022).

## Тестирование созданных веб-сервисов с помощью Bombardier

Теперь протестируем оба созданных веб-сервиса.

1. Откройте папку `bin\Release\net8.0\win-x64\publish` на панели File Explorer (Обозреватель файлов). Вы увидите EXE-файл размером около 9272 Кбайт. Это единственный файл, который требуется для развертывания веб-сервиса на другом компьютере под управлением операционной системы Windows. Файлы `appsettings.json` нужны только для переопределения конфигурации, а файл PDB используется для отладки.



Эта сборка зависит от операционной системы — она скомпилирована нативно для Windows x64.

2. В терминале перейдите в папку `bin\Release\net8.0\publish` и запустите веб-сервис на порту 5131:

```
.\Northwind.WebApi --urls="http://localhost:5131"
```



В PowerShell перед именем исполняемого файла нужно добавить префикс `.`. В других оболочках это может быть необязательно, но добавление префикса не помешает.

3. В другом терминале перейдите в папку `bin\Release\net8.0\win-x64\publish` и запустите веб-сервис АОТ на порту 5132:

```
.\Northwind.WebApiAot --urls="http://localhost:5132"
```

4. Откройте браузер и убедитесь, что оба веб-сервиса работают, посетив следующие ссылки:

```
http://localhost:5131/weatherforecast
http://localhost:5132/weatherforecast
```

5. В терминале запустите нагрузочное тестирование обычного веб-сервиса с помощью Bombardier, сделав один миллион запросов через 125 соединений (имитирующих пользователей) к веб-сервису на порту 5131:

```
bombardier -c 125 -n 1000000 http://localhost:5131/weatherforecast
```

6. Обратите внимание на результаты:

```
Bombarding http://localhost:5131/weatherforecast with 1000000 request(s)
using 125 connection(s)
1000000 / 1000000 [=====] 100.0% 85310/s 11s
=====
Done!
Statistics      Avg       Stdev      Max
  Req/sec     86113.31   14277.37   142372.22
  Latency      1.44ms    171.32us   27.01ms
HTTP codes:
  1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
  others - 0
Throughput:    51.39MB/s
```



В приведенном примере веб-сервис обработал в среднем 86 113 запросов в секунду и максимум 142 372. Средняя задержка составила 1,44 мс, максимальная — 27,01 мс. Обратите внимание, что тест выполнялся локально на одном компьютере.

7. Повторите нагрузочное тестирование для веб-сервиса АОТ: выполните один миллион запросов через 125 соединений к веб-сервису на порту 5132:

```
bombardier -c 125 -n 1000000 http://localhost:5132/weatherforecast
```

## 8. Обратите внимание на результаты:

```
Bombarding http://localhost:5132/weatherforecast with 1000000 request(s)
using 125 connection(s)
1000000 / 1000000 [=====] 100.00% 94956/s 10s
Done!
Statistics      Avg      Stdev      Max
  Reqs/sec     95165.74   11185.27  142047.69
  Latency       1.30ms    600.72us   50.20ms
HTTP codes:
  1xx - 0, 2xx - 1000000, 3xx - 0, 4xx - 0, 5xx - 0
  others - 0
Throughput:    56.78MB/s
```



В данном случае веб-сервис АОТ обработал в среднем 95 165 запросов в секунду и максимум 142 047. Средняя задержка составила 1,30 мс, максимальная — 50,20 мс. Количество запросов в секунду и средняя задержка примерно на 10 % лучше по сравнению с обычной версией сервиса.

## 9. Проведите еще один тест на нагрузку, отправляя как можно больше запросов за 10 секунд через 200 соединений для обычного веб-сервиса:

```
bombardier -c 200 -d 10s -l http://localhost:5131/weatherforecast
```

## 10. Обратите внимание на распределение задержек:

```
Bombarding http://localhost:5131/weatherforecast for 10s using 200
connection(s)
[=====] 10s
Done!
Statistics      Avg      Stdev      Max
  Reqs/sec     87073.67   9378.44  103406.56
  Latency       2.28ms    356.54us   68.68ms
Latency Distribution
  50%      2.07ms
  75%      2.81ms
  90%      3.21ms
  95%      3.56ms
  99%      4.31ms
HTTP codes:
  1xx - 0, 2xx - 877089, 3xx - 0, 4xx - 0, 5xx - 0
  others - 0
Throughput:    52.16MB/s
```



Согласно результатам, 95 % запросов возвращаются за 3,56 мс или быстрее, а 99 % — за 4,31 мс. В течение 10 секунд успешно возвращено 877 089 запросов.

11. Повторите аналогичный тест для веб-сервиса АОТ:

```
Bombarding http://localhost:5132/weatherforecast for 10s using 200
connection(s)
[=====] 10s
Done!
Statistics      Avg       Stdev       Max
  Req/sec     96390.06   25083.96   242524.25
  Latency      2.06ms    807.55us   65.86ms
  Latency Distribution
    50%        2.00ms
    75%        2.47ms
    90%        3.16ms
    95%        4.00ms
    99%        6.51ms
  HTTP codes:
    1xx - 0, 2xx - 968744, 3xx - 0, 4xx - 0, 5xx - 0
    others - 0
  Throughput:   57.61MB/s
```



В этом случае 95 % запросов возвращаются за 4,00 мс или быстрее, а 99 % — за 6,51 мс. В течение 10 с успешно возвращено 968 744 запроса. При этом задержка для худших 5 % и 1 % запросов у АОТ-сервиса оказалась выше, но общее количество обработанных запросов выросло примерно на 10 % у веб-сервиса с АОТ.

## Интерпретация результатов Bombardier

Важно правильно интерпретировать результаты тестирования, чтобы сделать обоснованные выводы о производительности. В табл. 13.5 представлены основные метрики и их толкование.

**Таблица 13.5.** Пояснения к метрикам Bombardier

Метрика	Значение и способы толкования
<b>Запросы в секунду (RPS)</b>	Измеряет, сколько запросов в секунду обрабатывает веб-сервис. Более высокий показатель свидетельствует о лучшей способности сервиса справляться с нагрузкой. Сравните RPS с ожидаемым объемом трафика, чтобы оценить, справится ли приложение с пиковыми нагрузками
<b>Задержка</b>	Время, необходимое для прохождения запроса от клиента к серверу и обратно. Включает обработку на сервере. Более низкая задержка означает лучшее время отклика для пользователей. Следует учитывать как среднее значение задержки, так и процентили (например, P95 и P99), чтобы оценить опыт пользователей при пиковых нагрузках

Метрика	Значение и способы толкования
<b>Пропускная способность</b>	Объем данных, передаваемых за единицу времени, обычно измеряется в килобайтах в секунду (KBps). Высокая пропускная способность может указывать на передачу больших объемов данных за один запрос, что влияет на стоимость и масштабируемость сервиса
<b>Количество успешных запросов</b>	Процент запросов, завершенных без ошибок. Показатель ниже 100 % может свидетельствовать о тайм-аутах, ошибках сервера (5xx) или других сбоях под нагрузкой. Ошибки требуют дальнейшего анализа и устранения
<b>Ошибки</b>	Включают тайм-ауты, коды ошибок HTTP и другие сбои. Анализ ошибок помогает определить узкие места в производительности или проблемы в логике приложения. Частые ошибки — повод обратить внимание на оптимизацию
<b>Время подключения</b>	Показатели самого быстрого, самого медленного и среднего времени соединения позволяют судить о стабильности работы сервиса. Значительные колебания могут указывать на сетевые проблемы или необходимость оптимизации серверной части

Хотя библиотека Bombardier сама по себе не отслеживает использование процессора и памяти, эти показатели важно контролировать отдельно во время тестирования. Высокая загрузка процессора или памяти может свидетельствовать о неэффективности кода или необходимости усилить оборудование. Мониторинг таких ресурсов помогает выявить узкие места и спланировать оптимизацию.

Следует учитывать, что результаты тестирования могут варьироваться в зависимости от среды — будь то среда разработки, промежуточная (staging) или эксплуатационная. Также важны внешние условия: скорость сети, нагрузка на сервер и другие факторы. Оценка производительности — это не разовая задача. Регулярное тестирование веб-сервиса после внесения изменений позволяет своевременно выявлять их влияние на производительность.

Интерпретируя результаты Bombardier, сосредоточьтесь на понимании поведения вашего веб-сервиса при разных уровнях нагрузки, выявлении узких мест и возможных ошибок в условиях стресса. Эти данные помогут в целенаправленной оптимизации.

Для более полной картины полезно сочетать результаты Bombardier с мониторингом ресурсов с помощью таких инструментов, как Prometheus и Grafana или стек ELK (Elasticsearch, Logstash и Kibana). Это позволит отслеживать использование ресурсов, производительность и состояние приложений, обеспечивая всесторонний контроль над стабильностью веб-сервиса.



**В копилку знаний:** дополнительную информацию о Bombardier можно найти в GitHub-репозитории по адресу [github.com/codesenberg/bombardier](https://github.com/codesenberg/bombardier).

## NBomber — фреймворк для нагрузочного тестирования

NBomber — это фреймворк нагрузочного тестирования, ориентированный на разработчиков .NET и предназначенный для создания сценариев тестирования производительности. Он позволяет писать сценарии на языке C#, благодаря чему легко интегрируется в привычный процесс разработки. NBomber подходит для тестирования веб-сервисов, баз данных SQL и пользовательских протоколов. Фреймворк предоставляет подробную отчетность по тестам и поддерживает масштабирование сценариев на несколько узлов, что позволяет моделировать высоконагруженные среды и получать реалистичные результаты.

### Сценарии NBomber

Сценарии в NBomber представляют собой планы тестирования, в которых определяется, какие действия будут выполняться во время теста, сколько пользователей или одновременных задач будет имитироваться и как долго будет длиться нагрузка.

Каждый сценарий в NBomber включает несколько основных компонентов.

- **Этапы** — отдельные операции или действия, которые выполняются в ходе теста. Например, для веб-приложения таким шагом может быть выполнение HTTP-запроса к определенной конечной точке.
- **Моделирование нагрузки** — определяет способ генерации нагрузки: указываются количество одновременных пользователей или запросов и продолжительность теста. NBomber поддерживает различные стратегии моделирования, включая постоянную нагрузку или постепенное увеличение и уменьшение нагрузки.
- **Настройки сценария** — дополнительные параметры, такие как глобальные заголовки для HTTP-запросов или действия по настройке и очистке, которые выполняются до и после тестирования.



**В копилку знаний:** подробнее о создании сценариев можно узнать на сайте [nbomber.com/docs/nbomber/scenario](http://nbomber.com/docs/nbomber/scenario).

### Моделирование нагрузки

Если ожидается, что ваш сервис будет обрабатывать до 120 запросов в секунду (RPS), вы можете смоделировать такой сценарий. Сначала плавно увеличьте количество запросов от 0 до 120 RPS, затем поддерживайте этот уровень в течение заданного времени и постепенно снизьте нагрузку.

Рассмотрим несколько примеров моделирования нагрузки.

- **Плавное увеличение скорости** — увеличение количества запросов до 120 в секунду за одну минуту. Так как интервал составляет одну секунду, скорость будет расти примерно на два запроса в секунду:

```
Simulation.RampingInject(rate: 120,
    interval: TimeSpan.FromSeconds(1),
    during: TimeSpan.FromMinutes(1))
```

- **Поддержание постоянной нагрузки** — отправка 120 запросов каждую секунду в течение двух минут:

```
Simulation.Inject(rate: 120,
    interval: TimeSpan.FromSeconds(1),
    during: TimeSpan.FromMinutes(2))
```

- **Плавное снижение скорости** — уменьшение количества запросов до 0 за 30 секунд. При интервале в одну секунду скорость будет снижаться примерно на четыре запроса в секунду:

```
Simulation.RampingInject(rate: 0,
    interval: TimeSpan.FromSeconds(1),
    during: TimeSpan.FromSeconds(30))
```

## Типы NBomber

Некоторые ключевые типы NBomber приведены в табл. 13.6.

**Таблица 13.6.** Типы NBomber

Тип	Описание
Scenario	Определяет сценарий с именем и делегатом для выполнения
LoadSimulation	Определяет симуляцию нагрузки с указанием целевой скорости, интервала между вводами и продолжительности. Благодаря режиму нарастающего ввода можно увеличивать или снижать количество запросов во время теста
Simulation	Фабрика классов для создания экземпляров LoadSimulation с помощью методов <code>Inject</code> и <code>RampingInject</code>
Response	Класс и его фабрика для создания ответов с помощью методов <code>Ok()</code> и <code>Fail()</code>
NBomberRunner	Определяет один или несколько сценариев с помощью метода <code>RegisterScenarios</code> , а затем запускает их с помощью метода <code>Run()</code>
IScenarioContext	Представляет контекст выполнения сценария. Предоставляет возможности для регистрации событий, получения информации о teste и остановки теста или конкретного сценария через методы <code>StopCurrentTest</code> и <code>StopScenario</code>

Основные методы NBomber перечислены в табл. 13.7.

**Таблица 13.7.** Методы NBomber

Метод	Описание
WithInit	Используется для подготовки сценария и его зависимостей. Подходит для настройки целевой системы, заполнения базы данных или применения конфигураций
WithClean	Очищает ресурсы после завершения сценария
WithWarmUpDuration	Устанавливает продолжительность прогрева перед началом основного тестирования (по умолчанию 30 секунд)
WithoutWarmUp	Отключает этап прогрева
WithLoadSimulations	Настраивает параметры моделирования нагрузки для текущего сценария. Подробнее можно узнать на сайте <a href="http://nbomber.com/docs/nbomber/load-simulation/">nbomber.com/docs/nbomber/load-simulation/</a>
WithMaxFailCount	Определяет максимальное количество допустимых сбоев ( <code>FailCount</code> ). По достижении этого значения нагрузочное тестирование будет остановлено. По умолчанию <code>MaxFailCount</code> равно 5000

## Пример проекта NBomber

Рассмотрим пример создания простого теста с NBomber.

1. В редакторе кода создайте консольное приложение `NBomberApp` и добавьте его в решение `Chapter13`.
2. В файле созданного проекта включите обработку предупреждений как ошибок, статически и глобально импортируйте класс `Console` и добавьте ссылки на пакеты `NBomber` и `NBomber.Http` для работы с HTTP-сценариями:

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

<ItemGroup>
  <PackageReference Include="NBomber" Version="5.6.0" />
  <PackageReference Include="NBomber.Http" Version="5.1.0" />
</ItemGroup>
```



Актуальные версии пакетов можно найти на сайтах [www.nuget.org/packages/NBomber](http://www.nuget.org/packages/NBomber) и [www.nuget.org/packages/NBomber.Http](http://www.nuget.org/packages/NBomber.Http).

3. Выполните сборку проекта NBomberApp для восстановления зависимостей.
4. В файле Program.cs удалите все содержимое и добавьте код для определения симуляции нагрузки, создания сценария NBomber и его запуска:

```
using NBomber.Contracts; // Подключение набора методов ScenarioProps
using NBomber.CSharp; // Подключение утилит Scenario, Simulation, NBomberRunner
using NBomber.Http; // Подключение плагина HttpMetricsPlugin
using NBomber.Http.CSharp; // Подключение плагина Http
using NBomber.Plugins.Network.Ping; // Подключение плагина PingPlugin

// Использование класса System.Net.Http.HttpClient для выполнения HTTP-запросов
using HttpClient client = new();

LoadSimulation[] loads = [
    // Постепенное увеличение нагрузки до 50 запросов в секунду за одну минуту
    Simulation.RampingInject(rate: 50,
        interval: TimeSpan.FromSeconds(1),
        during: TimeSpan.FromMinutes(1)),
    // Поддержание уровня 50 запросов в секунду в течение следующей минуты
    Simulation.Inject(rate: 50,
        interval: TimeSpan.FromSeconds(1),
        during: TimeSpan.FromMinutes(1)),
    // Плавное снижение нагрузки до 0 запросов в секунду за одну минуту
    Simulation.RampingInject(rate: 0,
        interval: TimeSpan.FromSeconds(1),
        during: TimeSpan.FromMinutes(1))
];

ScenarioProps scenario = Scenario.Create(
    name: "http_scenario",
    run: async context =>
{
    HttpRequestMessage request = Http.CreateRequest(
        "GET", "http://localhost:5131/weatherforecast")
        .WithHeader("Accept", "application/json");

    // Пример использования методов WithHeader и WithBody для отправки
    // JSON-полезной нагрузки:
    // .WithHeader("Content-Type", "application/json")
    // .WithBody(new StringContent("{ some JSON }",
    // Encoding.UTF8, "application/json"));

    Response<HttpResponseMessage> response = await Http.Send(client, request);

    return response;
})

.WithoutWarmUp()
.WithLoadSimulations(loads);
```

```
NBomberRunner
    .RegisterScenarios(scenario)
    .WithWorkerPlugins(
        new PingPlugin(PingPluginConfig.CreateDefault("nbomber.com")),
        new HttpMetricsPlugin([HttpVersion.Version1])
    )
    .Run();
```

5. Запустите проект NBomberApp без отладки.
6. Подождите три минуты, пока завершится выполнение теста, а затем изучите результаты, представленные в итоговом отчете (фрагмент):

```
13:20:39 [INF] NBomber "5.5.0" started a new session: "2024-03-12_13.20.19_session_ef615dae"
...
13:20:39 [INF] Starting bombing...
...
scenario: http_scenario
- ok count: 6030
- fail count: 0
- all data: 2.5 MB
- duration: 00:03:00

load simulations:
- ramping_inject, rate: 50, interval: 00:00:01, during: 00:01:00
- inject, rate: 50, interval: 00:00:01, during: 00:01:00
- ramping_inject, rate: 0, interval: 00:00:01, during: 00:01:00



| step               | ok stats  |
|--------------------|---|
| name               | global information  |
| request count      | all = 6030, ok = 6030, RPS = 33.5                             |
| latency            | min = 0.16 ms, mean = 0.87 ms, max = 55.25 ms, StdDev = 0.82  |
| latency percentile | p50 = 0.9 ms, p75 = 0.97 ms, p95 = 1.08 ms, p99 = 1.28 ms     |
| data transfer      | min = 0.406 KB, mean = 0.419 KB, max = 0.438 KB, all = 2.5 MB |



status codes for scenario: http_scenario


| status code | count | message |
|-------------|-------|---------|
| OK          | 6030  |         |


...
13:23:44 [INF] Reports saved in folder: "C:\tools-skills-net8\Chapter13\NBomberApp\bin\Release\net8.0\reports\2024-03-12_13.20.19_session_ef615dae"
13:23:44 [WRN] THIS VERSION IS FREE ONLY FOR PERSONAL USE. You can't use it for an organization.
```

7. После завершения теста остановите веб-сервисы и консольные приложения.



**В копилку знаний:** больше информации о NBomber можно получить на сайте nbomber.com.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 13.1. Дополнительные онлайн-материалы

- Официальную документацию по BenchmarkDotNet можно найти по адресу [benchmarkdotnet.org/articles/overview.html](http://benchmarkdotnet.org/articles/overview.html).
- В частности, полезно ознакомиться с разделом о передовых практиках, доступным по адресу [benchmarkdotnet.org/articles/guides/good-practices.html](http://benchmarkdotnet.org/articles/guides/good-practices.html).
- Существует интересный сайт, на котором разработчики могут публиковать свои результаты производительности и оставлять комментарии к работам других участников, — [dotnetbenchmarks.com](http://dotnetbenchmarks.com).

### Ошибки при оценке производительности

Один из моих любимых авторитетов в области .NET — Ник Косентино, также известный как Dev Leader. У Ника есть еженедельная рассылка (подписка доступна на сайте [subscribe.devleader.ca](http://subscribe.devleader.ca)), курсы ([devleader.ca/courses](http://devleader.ca/courses)) и репозитории на GitHub ([github.com/ncosentino/](http://github.com/ncosentino/)).

Недавно он обнаружил, что допустил ошибку в одном из своих видеороликов о производительности .NET. Вместо того чтобы замалчивать свою ошибку, он откровенно рассказал о ней.

«Совершать ошибки — это нормально. Мы все люди».

«Ошибки — отличный способ взять на себя ответственность и показать другим, что мы можем расти благодаря им».

«Неудачи — один из лучших способов учиться. Я, например, даже не знал, что MatchCollection — это отложенная коллекция, хотя пользовался Regex в C# больше десяти лет».

Открытость Ника и его стремление помогать другим учиться — одна из причин, почему я доверяю его советам.

Вы можете прочитать о его тестах производительности регулярных выражений и о том, как отложенная оценка повлияла на результаты, по адресу [dev.to/devleader/c-regular-expression-benchmarks-how-to-avoid-my-mistakes-2cef](https://dev.to/devleader/c-regular-expression-benchmarks-how-to-avoid-my-mistakes-2cef).

## Упражнение 13.2. Практические задания

- Больше о нотации «О большое» с использованием расширения Polyglot Notebooks можно узнать, прочитав статью Мэтта Эланда, автора книги *Refactoring with C#* издательства Packt. Статья опубликована по адресу [newdevsguide.com/2023/05/16/polyglot-notebooks-big-o/](https://newdevsguide.com/2023/05/16/polyglot-notebooks-big-o/).
- Вы также можете опробовать готовые шаблоны BenchmarkDotNet для создания проектов по адресу [benchmarkdotnet.org/articles/guides/dotnet-new-templates.html](https://benchmarkdotnet.org/articles/guides/dotnet-new-templates.html).

## Упражнение 13.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Что такое нотация «О большое»?
2. Какая типичная ошибка встречается в процессе оценки производительности и что предлагает BenchmarkDotNet, чтобы получить надежные результаты?
3. В чем разница между нагрузочным тестированием и стресс-тестированием и в каких случаях применяется каждое из них?
4. Что представляет собой инструмент Bombardier и чем он полезен при тестировании производительности?
5. Из чего состоит сценарий нагрузки в NBomber?

## Упражнение 13.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [git-hub.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-13---benchmarking-performance-load-and-stress-testing](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-13---benchmarking-performance-load-and-stress-testing).

## Резюме

В этой главе мы разобрали:

- ключевые концепции оценки производительности, такие как базовые показатели и нотация «О большое»;

- как использовать BenchmarkDotNet для проверки производительности кода;
- как применять Bombardier для оценки эффективности сервисов;
- как использовать NBomber для нагрузочного тестирования приложений.

В следующей главе мы поговорим о функциональном и сквозном тестировании. Вы узнаете, как убедиться, что приложения .NET работают так, как задумано, обеспечивая ожидаемый пользовательский опыт и соответствие требованиям.

# 14

## Функциональное и комплексное тестирование сайтов и сервисов

Эта глава посвящена функциональному и сквозному (End-to-End, E2E) тестированию, которые помогают убедиться, что решения .NET работают так, как ожидается с точки зрения пользователя, и соответствуют заданным требованиям. Вы познакомитесь с конкретными примерами применения этих подходов к тестированию в средах .NET, а также с основными инструментами, которые будут использоваться. Затем подробнее разберем применение фреймворка Microsoft Playwright для тестирования пользовательских веб-интерфейсов.

### Основы функционального и сквозного тестирования

Функциональное тестирование помогает проверить, работают ли отдельные компоненты или функции приложения так, как задумано, сравнивая фактические результаты с заданными требованиями. Оно детализировано, часто автоматизировано и сосредоточено на проверке входных и ожидаемых выходных данных, без углубления в устройство системы.

Сквозное тестирование имитирует реальные пользовательские сценарии и проверяет, как приложение работает от начала до конца. Оно затрагивает сразу несколько интегрированных компонентов и помогает убедиться, что они правильно взаимодействуют друг с другом. Такой подход позволяет выявить ошибки на стыке частей системы и дает более полное представление о функциональности и надежности приложения.

Мы начнем с практических примеров, которые охватывают разные аспекты современных приложений .NET – от веб-сервисов до пользовательских интерфейсов. На их основе будет видно: E2E-тесты затрагивают сразу несколько функций и выглядят сложнее, а функциональные тесты сосредоточены на проверке каждой функции отдельно.

## Пример 1. Тестирование сервиса веб-API

Допустим, что вы пользуетесь веб-API на базе ASP.NET Core, который применяется в качестве бэкенда для приложения управления задачами. Этот API включает конечные точки для создания, получения, обновления и удаления задач.

**Сценарий сквозного тестирования.** Пользователь создает задачу, вносит в нее изменения, отмечает как выполненную и извлекает обратно, чтобы проверить, что все сработало правильно.

1. Вызовите конечную точку `POST` и создайте новую задачу с нужными параметрами.
2. Вызовите конечную точку `PUT`, чтобы обновить название и описание задачи.
3. Вызовите еще одну конечную точку `PUT`, чтобы отметить задачу как выполненную.
4. Вызовите конечную точку `GET` и проверьте, что все изменения сохранены.

Для тестирования можно воспользоваться программой Postman вручную, создать тесты с помощью ее ИИ-помощника Postbot или написать автоматические тесты с использованием библиотеки RestSharp либо собственной клиентской библиотеки вместе с пакетом xUnit.

**Сценарий функционального тестирования.** Нужно проверить, правильно ли обрабатывается ввод при создании новой задачи.

1. Вызовите конечную точку `POST` с некорректными данными, например без обязательных полей.
2. Убедитесь, что API возвращает код состояния `400 Bad Request` с понятным сообщением об ошибке. Обычно такие сообщения формируются с помощью класса `ProblemDetails` в сервисах .NET.

Для написания тестов удобно использовать пакет xUnit, библиотеку FluentAssertions для выразительных проверок и фреймворк NSubstitute для создания имитаций.

## Пример 2. Тестирование сайта ASP.NET Core

Рассмотрим проект на базе MVC ASP.NET Core или Blazor для сайта электронной коммерции. Такой сайт позволяет просматривать товары, добавлять их в корзину и оформлять заказы.

**Сценарий сквозного тестирования.** Посетитель просматривает каталог товаров, находит нужный товар, добавляет его в корзину и завершает оформление заказа.

1. Перейдите на страницу списка товаров и выберите нужный товар.
2. Добавьте выбранный товар в корзину.
3. Перейдите на страницу корзины, убедитесь, что товар добавлен, и перейдите к оформлению заказа.

4. Заполните форму оформления заказа и подтвердите покупку.
5. Убедитесь, что открылась страница подтверждения с деталями заказа.

Для автоматизации таких сценариев удобно использовать фреймворк Playwright, интегрированный с xUnit.

**Сценарий функционального тестирования.** Проверьте, что добавление товара в корзину правильно обновляет количество товаров и общую стоимость.

1. Добавьте определенный товар в корзину.
2. Получите данные корзины и убедитесь, что количество товаров и сумма заказа обновились корректно.

Для написания таких тестов можно использовать пакет xUnit, а для имитации работы с базой данных или другими внешними системами — фреймворк NSubstitute.

### Пример 3. Тестирование чат-приложения на базе SignalR

Если вы разрабатываете приложение для обмена сообщениями в реальном времени с использованием библиотеки SignalR, особенно важно тщательно протестировать работу функций общения.

**Сценарий сквозного тестирования.** Два пользователя обмениваются сообщениями в чате и проверяют, что сообщения приходят в реальном времени.

1. Пользователь А отправляет сообщение в чате.
2. Убедитесь, что пользователь Б получает сообщение без задержки.
3. Пользователь Б отвечает на сообщение.
4. Убедитесь, что пользователь А получает ответ также в реальном времени.

Для автоматизации такого сценария можно использовать фреймворк Playwright с пользовательской логикой обработки событий чата. При необходимости можно эмулировать сервер или клиента SignalR для имитации реального взаимодействия.

**Сценарий функционального тестирования.** Проверьте, что отправка сообщения одним пользователем приводит к доставке сообщения всем подключенными клиентам.

1. Сымитируйте подключение нескольких клиентов к чату.
2. Отправьте сообщение от одного из клиентов.
3. Убедитесь, что все подключенные клиенты получили сообщение.

Для написания тестов удобно использовать пакет xUnit, а для имитации поведения клиентов — фреймворк NSubstitute совместно с эмуляцией среды SignalR.

Теперь, когда вы увидели на примерах, как сквозные и функциональные тесты помогают проверять разные стороны .NET-приложений и обеспечивают уверенность

в их стабильности, давайте перейдем к знакомству с отличным инструментом для тестирования пользовательских веб-интерфейсов.

## Тестирование пользовательских веб-интерфейсов с помощью Playwright

Playwright — это фреймворк с открытым исходным кодом для автоматизированного тестирования сайтов и веб-приложений в разных браузерах. Разработанный компанией Microsoft, он позволяет разработчикам и тестировщикам писать сценарии, имитирующие поведение пользователей на веб-страницах. С помощью Playwright можно автоматизировать навигацию по страницам, заполнение форм, нажатие кнопок, а также более сложные действия — например, взаимодействие с одностраничными приложениями, веб-компонентами и работу с загрузкой или выгрузкой файлов.

Чаще всего для тестирования используются браузеры WebKit, Firefox, Google Chrome, Microsoft Edge и другие на базе Chromium. В Playwright применяются сборки Chromium с открытым исходным кодом. Так как проект Chromium развивается быстрее фирменных браузеров, Playwright поддерживает более свежие версии: когда фирменный релиз Google Chrome будет иметь версию N, Playwright уже сможет работать с версией Chromium N+1, которая появится в браузерах спустя несколько недель.

Версия Firefox для Playwright основана на самой актуальной сборке Firefox Stable. Версия для WebKit использует последнюю версию ветки WebKit trunk до того, как она станет доступна в Apple Safari и других браузерах на базе WebKit. При этом Playwright не поддерживает фирменные версии Firefox или Safari, поскольку они используют специфические патчи.

Playwright также может работать с фирменными браузерами, установленными на вашем компьютере. В частности, поддерживается работа с каналами Stable и Beta браузеров, например, для настройки Microsoft Edge.

```
<?xml version="1.0" encoding="utf-8"?>
<RunSettings>
  <Playwright>
    <BrowserName>chromium</BrowserName>
    <LaunchOptions>
      <Channel>msedge</Channel>
    </LaunchOptions>
  </Playwright>
</RunSettings>
```

При запуске тестового проекта можно указать браузер и канал:

```
dotnet test -- Playwright.BrowserName=chromium Playwright.LaunchOptions.
  Channel=msedge
```

Различные браузеры могут по-разному отображать веб-страницы из-за различий в их движках. Указывая браузер, разработчики могут убедиться, что их приложения корректно работают в разных средах. Это помогает заранее выявлять проблемы, специфичные для конкретных браузеров, еще на ранних этапах разработки.

Выбор канала браузера особенно полезен, если нужно обеспечить совместимость с новыми функциями или с определенными версиями, которые пользователи все еще могут использовать. Иногда ошибки проявляются только в конкретных браузерах или на отдельных каналах. Явно указывая их в тестах, можно быстрее находить и устранять такие ошибки, повышая надежность и удобство работы с веб-приложением.

Тесты Playwright можно запускать сразу в нескольких браузерах. Для этого можно передать несколько аргументов командной строки, хотя чаще используется файл конфигурации. По умолчанию Playwright запускает тесты параллельно. Если настроить выполнение в нескольких браузерах, тесты будут распараллеливаться еще и по браузерам, что заметно ускоряет процесс и позволяет быстрее получить полное покрытие.

В Playwright предусмотрены специальные пакеты для интеграции со средами тестирования NUnit и MSTest. Если вы работаете с другими фреймворками, например с xUnit, или просто хотите писать тесты в консольном приложении, можно использовать основной пакет Playwright — именно так мы и поступим в этой главе.

## Преимущества Playwright для разработчиков .NET

С Playwright разработчикам .NET доступен набор мощных возможностей, которые помогают упростить и ускорить процессы разработки и тестирования. Среди ключевых преимуществ следующие.

- **Поддержка кросбраузерности.** Playwright позволяет тестировать приложения во всех современных браузерах, включая Chrome, Firefox, Safari и Edge. Благодаря этому можно убедиться в совместимости и надежности работы приложений в разных пользовательских средах без необходимости вручную проверять работу проекта в каждом браузере.
- **Богатый набор API.** В Playwright доступен широкий спектр API для автоматизации взаимодействия с веб-страницами, включая поддержку современных технологий — таких как веб-компоненты, теневой DOM и асинхронные операции. Они помогают создавать более сложные и точные тесты, которые реалистично имитируют поведение пользователей.
- **Скорость и надежность.** Тесты на базе Playwright работают быстро и стабильно. Архитектура фреймворка сводит к минимуму случайные отклонения в поведении тестов и повышает их надежность — это особенно важно для Agile-разработки и коротких циклов итераций.
- **Headless-режим.** Playwright поддерживает запуск браузеров в Headless-режиме — без отображения пользовательского интерфейса. Такой режим рабо-

тает быстрее и потребляет меньше памяти, что делает его отличным выбором для автоматизированных конвейеров тестирования и систем непрерывной интеграции (CI). Хотя Headless-режим поддерживается и в Selenium, реализация Playwright более оптимизирована, благодаря чему тесты выполняются быстрее.

- **Параллельное выполнение тестов.** Playwright позволяет запускать тесты параллельно, что существенно сокращает общее время тестирования. Эта возможность особенно цenna в средах CI/CD, где скорость и эффективность имеют большое значение.

## Альтернативы Playwright

Существует несколько альтернатив Playwright, которые могут быть интересны разработчикам .NET. Среди них особенно выделяются два основных конкурента.

- **Selenium.** Один из самых известных и широко используемых инструментов для тестирования веб-приложений. Selenium предлагает развитую экосистему и широкую поддержку браузеров. Однако в ряде сценариев он может уступать Playwright по скорости и эффективности, особенно при работе с современными веб-приложениями. Например, Playwright автоматически дожидается готовности элементов перед выполнением действий, благодаря чему снижается нестабильность тестов и отпадает необходимость в явных командах ожидания или принудительных паузах, которые часто встречаются в сценариях Selenium и могут замедлять выполнение тестов. Параллельное выполнение тестов в Playwright реализовано лучше: Selenium в этом плане ограничен из-за своей архитектуры и накладных расходов, связанных с использованием протокола WebDriver. При тестировании одностраничных приложений (SPA), которые активно используют JavaScript для динамической загрузки контента, Playwright взаимодействует с веб-страницами более плавно, так как изначально разрабатывался с учетом современных JavaScript-фреймворков и сложной навигации на стороне клиента. Кроме того, Selenium требует отдельные реализации WebDriver для каждого браузера, что добавляет сложности и может привести к несогласованности в тестах.
- **Puppeteer.** Разработанный командой Chrome DevTools Puppeteer — еще один популярный инструмент для автоматизации работы браузеров. В первую очередь он ориентирован на браузеры, основанные на Chrome и Chromium. Несмотря на схожесть по функциям с Playwright, Puppeteer не поддерживает работу с другими браузерами, такими как Firefox и Safari, поэтому его возможности при кросбраузерном тестировании ограничены.

Оба инструмента — и Selenium, и Playwright — имеют свои сильные и слабые стороны. Выбор между ними зависит от конкретных потребностей проекта: какие браузеры нужно поддерживать, насколько сложное у вас веб-приложение и в какой среде ведется разработка.

Главное преимущество Selenium перед Playwright — его богатая экосистема и поддержка огромного сообщества. Selenium используется с 2004 года и за это время стал одним из самых известных и широко используемых инструментов для автоматизации тестирования веб-приложений. Благодаря этому у него есть обширная документация, множество обучающих материалов, сторонних библиотек и интеграций — все, что может пригодиться в реальной разработке.

Playwright, в свою очередь, благодаря глубокой поддержке всех современных браузеров и удобной интеграции с .NET выглядит особенно привлекательным для .NET-разработчиков, которые ищут современный и эффективный способ автоматизировать тестирование своих веб-приложений.

## Распространенные типы объектов в Playwright

В Playwright предусмотрены типы для тестирования пользовательского интерфейса сайтов. Краткое их описание представлено в табл. 14.1.

**Таблица 14.1.** Распространенные типы объектов в Playwright

Тип	Описание
IPlaywright	Представляет систему Playwright. Содержит свойства Chromium, Firefox и Webkit для работы с разными браузерами, а также Selectors для выбора элементов на веб-странице в сценариях автоматизации
IBrowser	Представляет экземпляр браузера. Предоставляет свойства, такие как Contexts, IsConnected, BrowserType и Version, и методы вроде NewContextAsync и NewPageAsync для создания новых контекстов и страниц
IBrowserContext	Представляет отдельную сессию браузера. Хорошей практикой считается создание контекста, а затем страниц внутри него. Cookie-файлы и кэш не передаются между разными контекстами. Среди свойств — APIRequest и Pages, а среди методов — NewPageAsync, AddCookiesAsync, CookiesAsync и StorageStateAsync. Для эмуляции возможностей браузера доступны методы SetGeolocationAsync и SetOfflineAsync
IResponse	Представляет HTTP-ответ от веб-сервера. Предоставляет свойства Headers, Ok, Status и StatusText, а также методы BodyAsync, JsonAsync и TextAsync для работы с содержимым ответа
IPage	Представляет отдельную веб-страницу. Каждая страница принадлежит своему браузерному контексту и имеет собственные cookie-файлы и кэш
ILocator	Представляет один или несколько HTML-элементов на веб-странице, с которыми можно взаимодействовать в тестах

## Распространенные методы работы со страницами в Playwright

В Playwright используются различные методы для автоматизации действий на веб-странице, представленной экземпляром интерфейса `IPage`. Краткое описание некоторых из них приведено в табл. 14.2.

**Таблица 14.2.** Распространенные методы работы со страницами в Playwright

Метод <code>IPage</code>	Описание
<code>GotoAsync</code> , <code>GoBackAsync</code> , <code>GoForwardAsync</code>	Выполняют переход к указанному ресурсу, а также навигацию назад или вперед. Каждый из этих методов возвращает экземпляр интерфейса <code>IResponse</code>
<code>ContentAsync</code>	Позволяет получить полное HTML-содержимое текущей страницы
<code>TitleAsync</code>	Возвращает заголовок страницы

## Распространенные методы поиска элементов в Playwright

В Playwright применяются разные методы для поиска одного или нескольких элементов на веб-странице. Все эти методы возвращают экземпляр `ILocator`. Краткое описание основных методов приведено в табл. 14.3.

**Таблица 14.3.** Распространенные методы поиска элементов в Playwright

Метод	Описание
<code>GetByRole</code>	Ищет элементы на основе роли доступности. Например, <code>GetByRole(AriaRole.Heading, new() { Name = "Sign up" })</code> выберет элемент <code>&lt;h3&gt;Sign up&lt;/h3&gt;</code> , а <code>GetByRole(AriaRole.Button, new() { Name = "Sign in" })</code> — элемент <code>&lt;button&gt;Sign in&lt;/button&gt;</code>
<code>GetByLabel</code>	Помогает находить поля ввода по тексту метки. Например, <code>GetByLabel("Password").FillAsync("secret")</code> заполнит элемент <code>&lt;label&gt;Password&lt;input type="password" /&gt;&lt;/label&gt;</code>
<code>GetByPlaceholder</code>	Находит поля ввода по тексту-заполнителю (placeholder). Например, <code>GetByPlaceholder("name@example.com").FillAsync("playwright@microsoft.com")</code> заполнит элемент <code>&lt;input type="email" placeholder="name@example.com" /&gt;</code>
<code>GetByTestId</code>	Ищет элементы по значению атрибута <code>data-testid</code> , установленного в коде страницы
<code>GetByText</code>	Находит элементы по видимому тексту — например, в элементах <code>div</code> , <code>span</code> , <code>p</code> и других

*Продолжение ➔*

**Таблица 14.3** (окончание)

Метод	Описание
GetByTitle	Ищет элемент по значению атрибута <code>title</code> . Например, <code>GetByTitle("Unread messages")</code> найдет элемент <code>&lt;span title="Unread messages"&gt;3&lt;/span&gt;</code>
GetByAltText	Ищет изображения по тексту альтернативного описания (alt-текста). Например, <code>GetByAltText("playwright logo")</code> найдет элемент <code>&lt;img alt="playwright logo" src="/img/playwright-logo.svg" width="100" /&gt;</code>
Locator	Позволяет искать элементы по CSS- или XPath-селекторам. Однако стоит отдавать предпочтение методам <code>GetBy...</code> , так как CSS- и XPath-селекторы обычно менее устойчивы к изменениям в коде страницы
And, Or	Позволяют объединять несколько локаторов в логические выражения для более точного поиска элементов
First, Last, Nth	Возвращают первый, последний или указанный по порядку элемент из группы найденных. Эти методы следует использовать осторожно: при изменениях на странице могут возникнуть ошибки из-за выбора неправильного элемента



**Дельный совет:** старайтесь как можно чаще использовать локаторы поиска по роли (`GetByRole`). Это наиболее надежный способ находить элементы — он ближе к тому, как воспринимают страницу пользователи и вспомогательные технологии.

## Распространенные методы работы с локаторами в Playwright

В Playwright применяются разные методы для взаимодействия с элементами страницы через экземпляры интерфейса `ILocator`. Большинство этих методов рассчитано на работу с одним элементом. Если локатор сопоставит сразу несколько элементов, будет выброшено исключение. Краткое описание основных методов представлено в табл. 14.4.

**Таблица 14.4.** Распространенные методы работы с локаторами в Playwright

Метод <code>ILocator</code>	Описание
<code>CheckAsync, UncheckAsync</code>	Устанавливают/сбрасывают флагок или переключатель
<code>SelectOptionAsync</code>	Выбирает пункт в раскрывающемся списке
<code>ClickAsync</code>	Выполняет щелчок по кнопке или другому элементу

<b>Метод ILocator</b>	<b>Описание</b>
DblClickAsync	Выполняет двойной щелчок кнопкой мыши на элементе
FillAsync	Заполняет поле ввода текстом
FocusAsync	Устанавливает фокус на элемент
HoverAsync	Устанавливает указатель мыши на элемент
PressAsync	Нажимает клавишу внутри элемента
PressSequentiallyAsync	Нажимает последовательность клавиш внутри элемента
ScreenshotAsync	Делает снимок экрана элемента
ScrollIntoViewIfNeededAsync	Прокручивает страницу, чтобы элемент оказался в зоне видимости
SelectTextAsync	Выделяет текст внутри элемента
TapAsync	«Касается» элемента, что особенно полезно для тестирования на мобильных устройствах



**Внимание!** Все операции над локаторами, рассчитанные на один элемент DOM, вызовут исключение, если локатор найдет несколько совпадений. Например, если локатор соответствует сразу всем кнопкам на странице и при этом вызывается ClickAsync, будет выброшено исключение.



**В копилку знаний:** больше информации о работе с локаторами можно найти на сайте [playwright.dev/dotnet/docs/locators](https://playwright.dev/dotnet/docs/locators).

## Тестирование распространенных сценариев с помощью приложения eShopOnWeb

Инструмент Playwright подходит для автоматизации самых разных сценариев тестирования — от простой навигации по сайту до сложных взаимодействий с элементами страницы и проверок данных.

Чтобы приступить к практике, понадобится сайт для тестирования. Для этого мы воспользуемся эталонным приложением eShopOnWeb на базе ASP.NET Core компании Microsoft. Оно демонстрирует однопроцессную (монолитную) архитектуру и базовую модель развертывания приложений.



Репозиторий eShopOnWeb был заархивирован 23 мая 2024 года и теперь доступен только для чтения. Однако для целей обучения и тестирования он по-прежнему подходит.

В этом пошаговом упражнении вы загрузите репозиторий eShopOnWeb, развернете сайт и познакомитесь с ключевыми страницами и функциональностью, которые позже будут использоваться в тестах.

1. Перейдите в репозиторий eShopOnWeb по адресу [github.com/dotnet-architecture/eShopOnWeb](https://github.com/dotnet-architecture/eShopOnWeb).
2. Загрузите или клонируйте репозиторий на свой компьютер.
3. Откройте файл решения eShopOnWeb.sln в редакторе кода.
4. Выполните сборку решения.
5. Запустите проект Web без отладки. (Нет необходимости запускать проект веб-сервиса или административные проекты на Blazor — проект Web работает независимо.)
6. Проанализируйте вывод в терминале: при первом запуске создается база данных и автоматически заполняется данными.

```
info: Web[0]
      App created...
info: Web[0]
      Seeding Database...
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (194ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
      CREATE DATABASE [Microsoft.eShopOnWeb.CatalogDb];
...
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (1ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      CREATE TABLE [Baskets] (
          [Id] int NOT NULL IDENTITY,
          [BuyerId] nvarchar(40) NOT NULL,
          CONSTRAINT [PK_Baskets] PRIMARY KEY ([Id])
      );
...
info: Web[0]
      Adding Development middleware...
info: Web[0]
      LAUNCHING
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\GitHub\eShopOnWeb\src\Web
```

7. Если браузер не откроется автоматически, запустите его вручную и перейдите по адресу `localhost:5001/`. Должна отобразиться главная страница сайта (рис. 14.1).

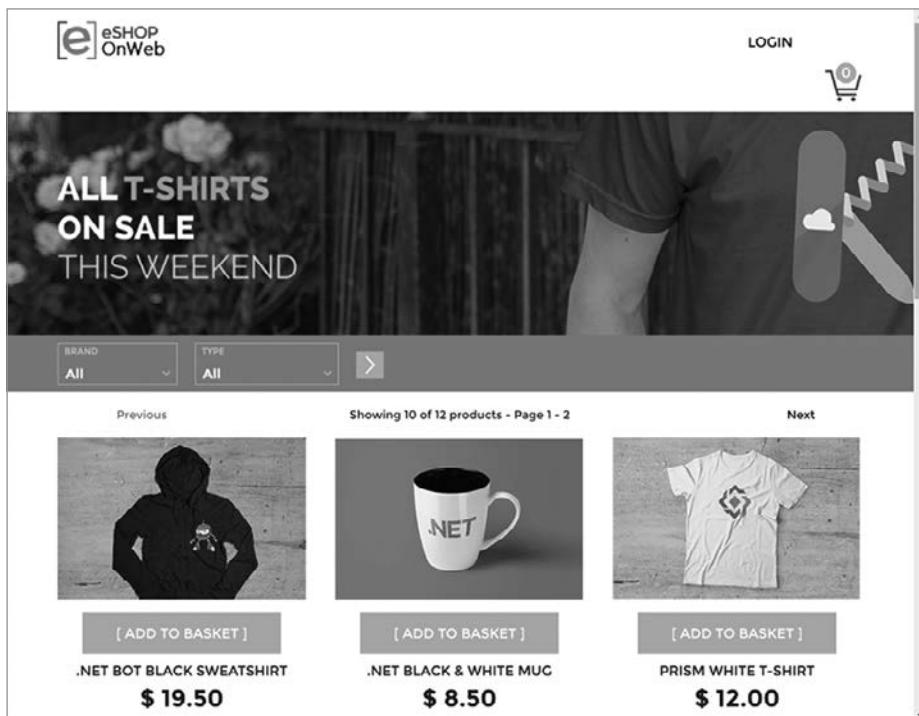


Рис. 14.1. Домашняя страница eShopOnWeb

8. Щелкните правой кнопкой мыши в центре белой шапки страницы, выберите в контекстном меню команду `View page source` (Просмотр исходного кода страницы) и обратите внимание на следующие элементы: элемент `title` внутри элемента `head`, а также элемент `div`, который используется для отображения количества товаров в корзине.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Catalog - Microsoft.eShopOnWeb</title>
    ...
    <a class="esh-basketstatus" href="/Basket">
        <div class="esh-basketstatus-image">
            
        </div>
```

```
<div class="esh-basketstatus-badge">
  0
</div>

</a>
...
```

9. Щелкните правой кнопкой мыши по раскрывающемуся списку BRAND, выберите в контекстном меню команду **Inspect** (Просмотреть код) и обратите внимание на два элемента **select** внутри элементов **label**, а также на элемент **input** внутри формы **form**.

```
<form method="GET">
  <label class="esh-catalog-label" data-title="brand">
    <select class="esh-catalog-filter"
      id="CatalogModel_BrandFilterApplied"
      name="CatalogModel.BrandFilterApplied">
      <option>All</option>
      <option value="2">.NET</option>
      <option value="1">Azure</option>
      <option value="5">Other</option>
      <option value="4">SQL Server</option>
      <option value="3">Visual Studio</option>
    </select>
  </label>
  <label class="esh-catalog-label" data-title="type">
    <select class="esh-catalog-filter"
      id="CatalogModel_TypesFilterApplied"
      name="CatalogModel.TypesFilterApplied"><option>All</option>
      <option value="1">Mug</option>
      <option value="3">Sheet</option>
      <option value="2">T-Shirt</option>
      <option value="4">USB Memory Stick</option>
    </select>
  </label>
  <input class="esh-catalog-send" type="image" src="/images/arrow-right.svg">
</form>
```



**Дельный совет:** HTML-разметка, с которой мы будем работать, взята из готового решения eShopOnWeb. Если бы мы создавали такую страницу самостоятельно, стоило бы внести два важных улучшения.

- Во-первых, метки (**label**) должны содержать текст, отображаемый в пользовательском интерфейсе, например BRAND или TYPE. Однако в проекте вместо текста используются атрибуты **data-title**, что затрудняет поиск элементов.
- Во-вторых, элементы управления, которые предполагается автоматизировать, такие как раскрывающиеся списки **select**, желательно снабжать атрибутом **data-testid** с уникальным значением. Это упростило бы автоматизацию поиска элементов.

Поскольку в текущем коде эти рекомендации не соблюдены, нам придется использовать менее удобные методы для поиска нужных элементов на странице.

10. Закройте панель инспектора кода в браузере.
11. На главной странице выберите в списке BRAND пункт .NET.
12. В списке TYPE выберите пункт Mug.
13. Рядом со списком TYPE щелкните кнопкой мыши на изображении стрелки, направленной вправо. Обратите внимание, что товары на странице отфильтрованы — теперь отображаются только кружки с надписью .NET, например товар с названием .NET BLACK & WHITE MUG.
14. В верхней части страницы нажмите кнопку LOGIN (Войти).
15. Щелкните правой кнопкой мыши на белом фоне страницы, выберите в контекстном меню команду View page source (Просмотр исходного кода страницы) и найдите элементы разметки: две метки (label), два поля ввода (input) и кнопку для входа в систему.

```
<label for="Input_Email">Email</label>
<input class="form-control" type="email" data-val="true"
    data-val-email="The Email field is not a valid e-mail address."
    data-val-required="The Email field is required." id="Input_Email"
    name="Input.Email" value="" />
...
<label for="Input_Password">Password</label>
<input class="form-control" type="password" data-val="true"
    data-val-required="The Password field is required."
    id="Input_Password" name="Input.Password" />
...
<button type="submit" class="btn btn-default">Log in</button>
```

16. Закройте вкладку с просмотром исходного кода страницы.
17. На странице входа заполните форму, указав адрес электронной почты `admin@microsoft.com` и пароль `Pass@word1`, затем нажмите кнопку Log in (Войти). После успешного входа ваша электронная почта должна отобразиться в верхней части главной страницы.
18. Завершите работу, закрыв браузер и остановив веб-сервер. Для этого в терминале нажмите сочетание клавиш `Ctrl+C`.

В следующих шагах мы рассмотрим несколько распространенных примеров тестов, которые часто автоматизируют с помощью Playwright, — начиная с базовой навигации по сайту.

## Навигация по страницам и проверка заголовков

Один из базовых тестов — переход на веб-страницу и проверка ее названия (элемента `title`). Такой тест помогает убедиться, что загружается правильная страница, и может использоваться как *дымовой тест* для проверки доступности сайта и корректной маршрутизации.

Приступим!

1. Оставьте решение eShopOnWeb открытым в редакторе кода и запустите новый экземпляр редактора.
2. В редакторе кода создайте новый C#-проект xUnit Test Project (Тестовый проект xUnit) / xunit, присвойте ему имя WebUITests и добавьте в решение Chapter14. В терминале для этого откройте папку Chapter14 и выполните следующие команды:

```
dotnet new sln
dotnet new xunit -o WebUITests
dotnet sln add WebUITests
```

3. В файле WebUITests.csproj проекта добавьте ссылку на пакет Playwright:

```
<PackageReference Include="Microsoft.Playwright" Version="1.44.0" />
```



Актуальную версию Playwright для .NET можно найти по адресу [www.nuget.org/packages/Microsoft.Playwright](http://www.nuget.org/packages/Microsoft.Playwright).

4. Выполните сборку проекта WebUITests, чтобы восстановить все зависимости.
5. Перейдите в каталог WebUITests\bin\Debug\net8.0 и с помощью терминала установите браузеры для работы Playwright:

```
pwsh playwright.ps1 install
```



Перед началом работы Playwright нужно загрузить специальные версии браузеров, предназначенные для тестирования. Их установка выполняется с помощью сценария PowerShell playwright.ps1, который входит в проект. Если при запуске тестов возникают ошибки, дополнительную информацию можно найти на сайте [playwright.dev/dotnet/docs/browsers](https://playwright.dev/dotnet/docs/browsers).

6. Обратите внимание: Playwright загрузит собственные копии браузеров Chrome, Firefox и WebKit. Процесс будет отображен в выводе терминала:

```
Downloading Chromium 123.0.6312.4 (playwright build v1105) from https://playwright.azureedge.net/builds/chromium/1105/chromium-win64.zip
122.2 MiB [=====] 100% 0.0s
Chromium 123.0.6312.4 (playwright build v1105) downloaded to C:\Users\markj\AppData\Local\ms-playwright\chromium-1105
Downloading Firefox 123.0 (playwright build v1440) from https://playwright.azureedge.net/builds/firefox/1440/firefox-win64.zip
83.4 MiB [=====] 100% 0.0s
Firefox 123.0 (playwright build v1440) downloaded to C:\Users\markj\AppData\Local\ms-playwright\firefox-1440
```

```
Downloading Webkit 17.4 (playwright build v1983) from https://playwright.azureedge.net/builds/webkit/1983/webkit-win64.zip
47.2 MiB [=====] 100% 0.0s
Webkit 17.4 (playwright build v1983) downloaded to C:\Users\markj\AppData\Local\ms-playwright\webkit-1983
```



Если браузеры не будут установлены заранее, при попытке запуска тестов появится сообщение:

```
Looks like Playwright was just installed or updated. Please run the
following command to download new browsers: pwsh bin/Debug/netX/
playwright.ps1 install
```

(Похоже, что Playwright был только что установлен или обновлен. Пожалуйста, выполните следующую команду для загрузки новых браузеров: pwsh bin/Debug/netX/playwright.ps1 install.)

7. В проекте `WebUITests` переименуйте файл `UnitTest1.cs` в `eShopWebUITests.cs`.
8. В этом файле создайте класс с фабричным методом для настройки Playwright на работу с браузером Chromium. Добавьте тестовый метод, который будет выполнять следующие действия.
  - Открывать браузер и переходить на домашнюю страницу.
  - Проверять, что домашняя страница загружается непустой и возвращает код состояния 200 OK.
  - Проверять, что название страницы (`title`) соответствует ожидаемому тексту.
  - Делать снимок экрана домашней страницы и сохранять его на рабочем столе, чтобы можно было увидеть, как страница выглядела в момент тестирования.

```
using Microsoft.Playwright; // Подключение пространств имен Playwright, IBrowser
                           // и других необходимых компонентов
namespace WebUITests;
```

```
public class eShopWebUITests
{
    private IBrowser? _browser;
    private IBrowserContext? _session;
    private IPage? _page;
    private IResponse? _response;

    private async Task GotoHomePage(IPPlaywright playwright)
    {
        _browser = await playwright.Chromium.LaunchAsync(
            new BrowserTypeLaunchOptions { Headless = false });

        _session = await _browser.NewContextAsync();
        _page = await _session.NewPageAsync();
        _response = await _page.GotoAsync("https://localhost:5001/");
    }
}
```

```
[Fact]
public async void HomePage_Title()
{
    // Подготовка: запуск браузера Chrome и переход на домашнюю страницу.
    // Использование конструкции Dispose гарантирует правильное завершение
    // работы браузера после выполнения теста.
    using IPlaywright? playwright = await Playwright.CreateAsync();
    await GotoHomePage(playwright);

    string actualTitle = await _page.TitleAsync();

    // Проверка: переход на главную страницу выполнен успешно, название
    // страницы соответствует ожидаемому.
    string expectedTitle = "Catalog - Microsoft.eShopOnWeb";
    Assert.NotNull(_response);
    Assert.True(_response.Ok);
    Assert.Equal(expectedTitle, actualTitle);

    // При сохранении снимка экрана используется универсальный сортируемый
    // формат даты и времени ("u"): 2009-06-15 13:45:30Z.
    // Символы двоеточия и пробелы заменяются тире, чтобы избежать
    // ошибок в имени файла.
    string timestamp = DateTime.Now.ToString("u")
        .Replace(":", "-").Replace(" ", "-");

    await _page.ScreenshotAsync(new PageScreenshotOptions
    {
        Path = Path.Combine(Environment.GetFolderPath(
            Environment.SpecialFolder.Desktop),
            $"homepage-{timestamp}.png")
    });
}
```

9. В решении eShopOnWeb запустите проект Web без отладки.
10. Запустите тест и посмотрите результат: тест должен пройти успешно. На короткое время появится окно браузера Chrome.
11. На рабочем столе откройте сохраненное изображение и убедитесь, что это снимок главной страницы.
12. В методе GotoHomePage файла eShopWebUITests.cs измените настройки запуска браузера, установив режим Headless (без пользовательского интерфейса):

```
browser = await playwright.Chromium.LaunchAsync(
    new BrowserTypeLaunchOptions { Headless = true });
```



При запуске браузера с помощью метода LaunchAsync объект BrowserTypeLaunchOptions можно не передавать, так как по умолчанию свойство Headless установлено в значение true.

13. Запустите тест снова. Обратите внимание: теперь браузер не появляется на экране, а тест выполняется быстрее.

Теперь мы готовы перейти к более сложным примерам тестирования.

## Взаимодействие с элементами веб-интерфейса

Одни из самых распространенных действий в веб-интерфейсе — это выбор значений из раскрывающихся списков, щелчки на кнопках и значках, заполнение и отправка форм, а также проверка видимости элементов или содержания их текста.

### Выбор элементов списка и элементов управления

На главной странице сайта eShopOnWeb отображается корзина с значком, отражающим количество товаров. Посетитель может отфильтровать товары, выбрав значения в двух раскрывающихся списках и нажав на изображение стрелки. Давайте автоматизируем эти действия.

1. В файле `eShopWebUITests.cs` статически импортируйте класс `Assertions`, чтобы можно было использовать свойство `Expect` без явного указания полного пути:

```
using static Microsoft.Playwright.Assertions; // Подключение свойства Expect
```

2. В файл `eShopWebUITests.cs` добавьте тест, который проверяет, что корзина пуста и отображается на странице:

```
[Fact]
public async void HomePage_CartEmptyAndVisible()
{
    // Подготовка: запуск браузера Chrome и переход на домашнюю страницу
    using IPlaywright? playwright = await Playwright.CreateAsync();
    await GotoHomePage(playwright);

    // Для выбора значка корзины потребуется использовать CSS-селектор
    ILocator element = _page.Locator("css=div.esh-basketstatus-badge");

    // Текст внутри значка может содержать пробельные символы, такие как `\\n`,
    // поэтому их нужно предварительно удалить перед проверкой
    string? actualCount = (await element.TextContentAsync())?.Trim();

    // Проверка: значок корзины содержит ожидаемое значение
    string expectedCount = "0";
    Assert.Equal(expectedCount, actualCount);
    await Expect(element).ToBeVisibleAsync();
}
```

3. В файле `eShopWebUITests.cs` добавьте тест, который проверяет работу фильтрации товаров:

```
[Fact]
public async void HomePage_FilterCategories()
{
    // Подготовка: запуск браузера Chrome и переход на домашнюю страницу
    using IPlaywright? playwright = await Playwright.CreateAsync();
    await GotoHomePage(playwright);

    // Метод GetByTestId по умолчанию ищет элементы по атрибуту `data-testid`,
    // однако в проекте eShopOnWeb он не используется, поэтому потребуется
    // явно указать фреймворку использовать атрибут `id`.
    playwright.Selectors.SetTestIdAttribute("id");

    // Установка в поле списка BRAND значения ".NET"
    ILocator brand = _page.GetByTestId("CatalogModel_BrandFilterApplied");
    await brand.SelectOptionAsync(".NET");

    // Установка в поле списка TYPE значения "Mug"
    ILocator type = _page.GetByTestId("CatalogModel_TypesFilterApplied");
    await type.SelectOptionAsync("Mug");

    // Щелчок кнопкой мыши на изображении стрелки для применения фильтра
    ILocator apply = _page.Locator("css=input.esh-catalog-send");
    await apply.ClickAsync();

    // Проверка: на странице отображается один товар
    ILocator topPager = _page.Locator("css=span.esh-pager-item").First;

    string? actualPager = (await topPager.TextContentAsync()).Trim();
    string expectedPager = "Showing 1 of 1 products - Page 1 - 1";
    Assert.Equal(expectedPager, actualPager);

    string timestamp = DateTime.Now.ToString("u")
        .Replace(":", "-").Replace(" ", "-");

    await _page.ScreenshotAsync(new PageScreenshotOptions
    {
        Path = Path.Combine(Environment.GetFolderPath(
            Environment.SpecialFolder.Desktop),
            $"dotnet-mug-{timestamp}.png")
    });
}
```



**Дельный совет:** на главной странице сайта предусмотрено два элемента для перехода между страницами товаров — один вверху списка, другой внизу. В строгом режиме работы Playwright тест завершится с ошибкой, если не указать явно, какой элемент использовать. Чтобы избежать ошибки, применяйте методы `First` или `Nth` для выбора нужного элемента.

- При необходимости запустите проект `Web` в решении `eShopOnWeb` без отладки.
- Запустите все тесты и проверьте результаты: все тесты должны пройти успешно. На снимке экрана будет отображен единственный отфильтрованный товар.

## Отправка форм, проверка подлинности и валидация

Автоматизация отправки форм и проверка результата — один из самых распространенных сценариев тестирования веб-приложений. С помощью Playwright можно заполнять поля форм, нажимать кнопки отправки и проверять, приводит ли это к ожидаемому результату — например, к отображению страницы благодарности, сообщению об ошибке валидации или созданию новой записи в базе данных.

Тестирование функций входа и выхода из системы также является важной частью проверки веб-приложений. Сценарий может включать в себя следующие действия:

```
// Переход на страницу входа в систему
await page.GotoAsync("https://example.com/login");
await page.FillAsync("input#username", "dummyuser");
await page.FillAsync("input#password", "123456");
await page.ClickAsync("button#login");

// Проверка успешного входа путем поиска кнопки выхода из системы
// и приветственного сообщения.
bool isLoggedIn = await page.IsVisibleAsync("button#logout");
Assert.True(isLoggedIn);
string successMessage = await page.InnerTextAsync("div.success");
Assert.Equal("Welcome, Dummy!", successMessage);

// Выполнение действий от имени аутентифицированного пользователя.
// Завершение сессии: выход из системы.
...
await page.ClickAsync("button#logout");
```

Инструмент Playwright позволяет имитировать поведение пользователя на каждом этапе, включая заполнение полей, выполнение действий после входа и корректное завершение сессии.

## Проверка адаптивности дизайна

Проверка, насколько хорошо пользовательский интерфейс реализует адаптивный дизайн, занимает важное место в современной веб-разработке. Давайте рассмотрим несколько распространенных сценариев.

### Эмуляция размеров экрана

С помощью Playwright можно протестировать, как веб-приложение работает на устройствах с разными размерами экрана. Это особенно важно для обеспечения качественного пользовательского опыта на мобильных, планшетных и десктопных устройствах. Например, можно проверить, что важные элементы интерфейса остаются доступными для мобильных пользователей:

```
await page.setViewportSizeAsync(640, 480); // Переход к мобильному представлению:
  // установка размеров экрана,
  // имитирующих мобильное устройство
await page.GotoAsync("https://example.com");
bool isLoggedIn = await page.IsVisibleAsync("div#importantSection");
Assert.True(isLoggedIn);
```

При необходимости размеры окна просмотра можно вернуть к стандартным параметрам для десктопного интерфейса:

```
await page.setViewportSizeAsync(1920, 1080); // Переход к просмотру в десктопном
  // размере: восстановление
  // размеров экрана
```

## Эмуляция устройств

Инструмент Playwright позволяет эмулировать поведение конкретных устройств уже при создании контекста браузера:

```
browser = await playwright.Chromium.LaunchAsync(
    new BrowserTypeLaunchOptions { Headless = false });
BrowserNewContextOptions iphone13 = playwright.Devices["iPhone 13"];
IBrowserContext context = await browser.NewContextAsync(iphone13);
```

## Эмуляция локали, часового пояса и геолокации

Инструмент Playwright позволяет эмулировать локаль пользователя и часовой пояс устройства:

```
BrowserNewContextOptions options = new()
{
    Locale = "de-DE", // Установка заголовка Accept-Language для имитации локали
    TimezoneId = "Europe/Berlin"
};
IBrowserContext context = await browser.NewContextAsync(options);
```



Список допустимых часовых поясов для Chromium опубликован по адресу [source.chromium.org/chromium/chromium/deps/icu.git/+/faee8bc70570192d82d2978a71e2a615788597d1:source/data/misc/metaZones.txt](https://source.chromium.org/chromium/chromium/deps/icu.git/+/faee8bc70570192d82d2978a71e2a615788597d1:source/data/misc/metaZones.txt).

Можно предоставить браузеру разрешение на доступ к геолокации и установить координаты определенного места, например Колизея в Риме:

```
BrowserNewContextOptions options = new()
{
    Permissions = [ "geolocation" ],
    Geolocation = new() { Longitude = 41.890221F, Latitude = 12.492348F }
};
IBrowserContext context = await browser.NewContextAsync(options);
```

Если нужно динамически изменить местоположение, например эмулировать перемещение к Эйфелевой башне в Париже, можно вызвать соответствующий метод:

```
await context.SetGeolocationAsync(new()
{ Longitude = 48.858455F, Latitude = 2.294474F });
```

## Эмуляция цветовых схем и темного режима

Инструмент Playwright поддерживает эмуляцию темного режима для проверки адаптивности дизайна:

```
BrowserNewContextOptions options = new()
{
    ColorScheme = ColorScheme.Dark
};
IBrowserContext context = await browser.NewContextAsync(options);
```

Если необходимо переключать цветовые схемы динамически, это можно сделать прямо на уровне страницы:

```
await page.EmulateMediaAsync(new()
{
    ColorScheme = ColorScheme.Dark
});
```

## Эмуляция агента пользователя, отключение JavaScript и офлайн-режим

Дополнительные настройки эмуляции включают изменение строки пользовательского агента (User-Agent), отключение JavaScript и переход в режим работы без доступа к интернету:

```
BrowserNewContextOptions options = new()
{
    UserAgent = "My User Agent",
    JavaScriptEnabled = false,
    Offline = true
};
IBrowserContext context = await browser.NewContextAsync(options);
```



**В копилку знаний:** больше информации об эмуляции различных аспектов среды браузера можно найти на сайте [playwright.dev/dotnet/docs/emulation](https://playwright.dev/dotnet/docs/emulation).

## Одностраничные приложения (SPA) и динамический контент

Инструмент Playwright хорошо подходит для тестирования сложных веб-приложений, которые активно используют JavaScript для динамической загрузки данных, одностраничных приложений (SPA) и AJAX-запросов. Тесты могут ожидать появления элементов на странице, полной загрузки контента или завершения сетевых запросов, прежде чем переходить к следующему действию:

```
await page.GotoAsync("https://example.com/spa");
await page.ClickAsync("button#loadData");

// Ожидание загрузки данных на странице перед выполнением действий
await page.WaitForSelectorAsync("div.dataLoaded");
string loadedData = await page.InnerTextAsync("div.dataLoaded");
```

Теперь посмотрим, как с помощью Playwright можно генерировать код тестирования автоматически.

## Генерация тестов с помощью инструмента Playwright Inspector

Генерация кода в Playwright для .NET — это очень удобная возможность, которая заметно ускоряет процесс автоматизации тестирования. Этот механизм работает как личный помощник: пока вы перемещаетесь по веб-приложению, инструмент отслеживает ваши действия и автоматически создает на их основе тестовые сценарии. Фактически это автоматизация самой автоматизации — отличный метаподход для ускорения работы.

Инструмент Playwright Inspector позволяет за короткое время создать полноценные тестовые сценарии, на написание которых вручную потребовалось бы гораздо больше усилий. Инструмент с высокой точностью фиксирует действия пользователя — например, щелчки кнопкой мыши, ввод текста и переходы между страницами, — тем самым снижая вероятность ошибок, которые могут возникнуть при ручной разработке тестов. Даже если вы только начинаете осваивать автоматизацию тестирования, с помощью инспектора можно быстро приступить к работе. Кроме того, изучение сгенерированного кода помогает лучше понять, как в Playwright строятся тесты.

По мере работы Playwright записывает каждое ваше действие и в реальном времени генерирует соответствующий код на C#. Когда вы завершите запись, инструмент остановит генерацию и выведет готовый сценарий. Этот код можно просмотреть, при необходимости доработать и интегрировать в ваш проект.

После интеграции сгенерированных тестов в проект их можно запускать с помощью стандартного механизма тестирования Playwright, чтобы убедиться, что приложение корректно работает в разных браузерах и на разных устройствах. Сгенерированные сценарии — отличная отправная точка, но не стоит бояться адаптировать их под свои конкретные требования к тестированию.

Посмотрим на Playwright Inspector в действии.

1. При необходимости запустите проект `Web` без отладки.
2. В терминале в папке проекта `WebUITests` введите команду для запуска инструмента генерации кода:

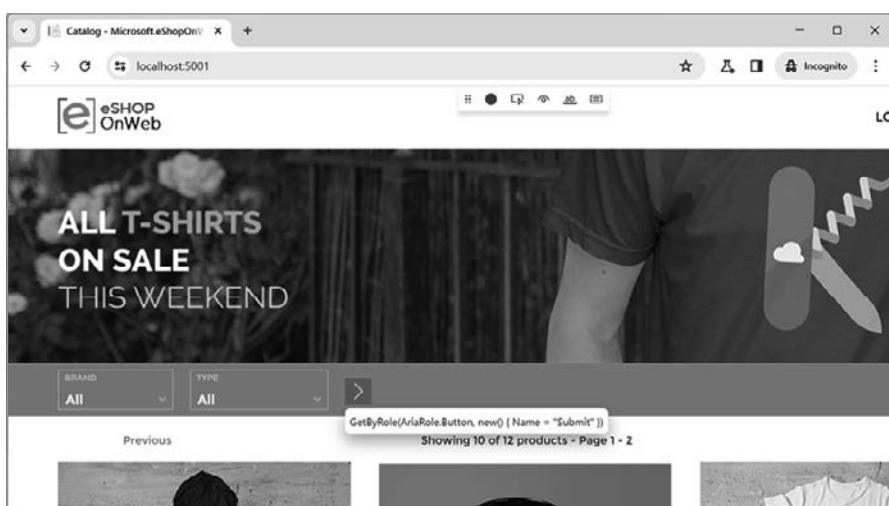
```
pwsh bin/Debug/net8.0/playwright.ps1 codegen https://localhost:5001/
```

3. Откроются два окна приложений:
  - окно браузера Chromium со значком Chrome в синем оттенке;
  - окно Playwright Inspector с изображением театральных масок (рис. 14.2).

4. В окне Chromium будет доступна плавающая панель инструментов с кнопками Record (Запись), Pick locator (Выбрать локатор), Assert visibility (Проверить видимость), Assert text (Проверить текст) и Assert value (Проверить значение). Когда вы наводите указатель мыши на элемент страницы, например на кнопку со стрелкой вправо для фильтрации товаров, инспектор автоматически предлагает локатор для выбора (рис. 14.3).

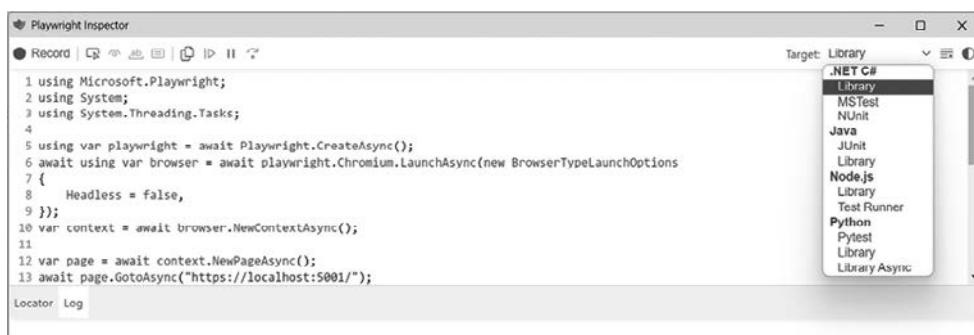


**Рис. 14.2.** Значки приложений Playwright Inspector и Chromium на панели задач Windows



**Рис. 14.3.** Окно Chromium с панелью инструментов Playwright Inspector

5. В окне Playwright Inspector отображается сгенерированный код на C# для автоматизации теста браузера. В правом верхнем углу окна можно выбрать язык и тестовый фреймворк: например, .NET C# Library, NUnit, Node.js Test Runner и др. (рис. 14.4).



**Рис. 14.4.** Playwright Inspector с C#-кодом для автоматизации теста браузера



**Внимание!** Когда кнопка Record (Запись) красного цвета — запись активна. Если кнопка черная — запись остановлена. Инструмент начинает запись автоматически при запуске.

6. Переключитесь на окно браузера Chromium.
7. Для удобства расположите оба окна рядом: так вы сможете одновременно видеть код, который генерируется при ваших действиях в браузере.
8. На главной странице сайта в верхней панели навигации нажмите кнопку LOGIN (Войти). Обратите внимание: в окне инспектора появится новое выражение для выбора локатора. Щелкните кнопкой мыши на предложенном элементе, чтобы зафиксировать действие:

```
await page.GetByRole(AriaRole.Link, new() { Name = "Login"}).ClickAsync();
```

9. В поле Email (Электронная почта) введите адрес admin@microsoft.com.
10. В поле Password (Пароль) введите значение Pass@word1.
11. Нажмите кнопку Log in (Войти).
12. На главной странице на панели инструментов нажмите кнопку Assert text (Проверить текст).
13. Щелкните кнопкой мыши на тексте admin@microsoft.com.
14. В появившемся окне Assert that element contains text (Проверка, что элемент содержит текст) подтвердите значение admin@microsoft.com и нажмите кнопку Accept (Принять).
15. В списке товаров найдите товар Cup<T> White Mug и нажмите кнопку Add To Basket (Добавить в корзину).
16. В поле Quantity (Количество) введите 3, затем нажмите кнопку Update (Обновить) и кнопку Checkout (Оформить заказ).
17. На странице Review (Просмотр) нажмите кнопку Pay Now (Оплатить сейчас).
18. На главной странице наведите указатель мыши на надпись admin@microsoft.com в верхней навигации и выберите пункт LOG OUT (Выйти).
19. На панели инструментов нажмите красную кнопку Record (Запись), чтобы остановить запись.
20. В окне Playwright Inspector появится следующий сгенерированный код:

```
using Microsoft.Playwright;
using System;
using System.Threading.Tasks;

using var playwright = await Playwright.CreateAsync();
await using var browser =
    await playwright.Chromium.LaunchAsync(new BrowserTypeLaunchOptions
```

```
{  
    Headless = false,  
});  
var context = await browser.NewContextAsync();  
var page = await context.NewPageAsync();  
await page.GotoAsync("https://localhost:5001/");  
await page.GetByRole(AriaRole.Link, new() { Name = "Login" }).ClickAsync();  
await page.GetByLabel("Email").ClickAsync();  
await page.GetByLabel("Email").FillAsync("admin@microsoft.com");  
await page.GetByLabel("Email").PressAsync("Tab");  
await page.GetByLabel("Password").FillAsync("Pass@word1");  
await page.GetByLabel("Password").PressAsync("Tab");  
await page.GetByRole(AriaRole.Button, new() { Name = "Log in" }).ClickAsync();  
await Expect(page.Locator("#logoutForm"))  
    .toContainTextAsync("admin@microsoft.com");  
await page.Locator("form").Filter(  
    new() { HasText = "[ ADD TO BASKET ] Cup<T>" })  
    .GetByRole(AriaRole.Button).ClickAsync();  
await page.GetByRole(AriaRole.Spinbutton).ClickAsync();  
await page.GetByRole(AriaRole.Spinbutton).FillAsync("3");  
await page.GetByRole(AriaRole.Button,  
    new() { Name = "[ Update ]" }).ClickAsync();  
await page.GetByRole(AriaRole.Link,  
    new() { Name = "[ Checkout ]" }).ClickAsync();  
await page.GetByRole(AriaRole.Button,  
    new() { Name = "[ Pay Now ]" }).ClickAsync();  
await page.GetText("admin@microsoft.com").ClickAsync();  
await page.GetByRole(AriaRole.Link,  
    new() { Name = "Log Out" }).ClickAsync();
```



Обратите внимание на утверждение вида:

```
await Expect(page.Locator("#logoutForm")).  
    .toContainTextAsync("admin@microsoft.com");
```

21. Теперь скопируйте этот код, вставьте его в свой тестовый проект и отредактируйте, убрав лишние действия — например, щелчки по текстовым полям или нажатия клавиши Tab:

```
// Эти действия можно удалить для упрощения теста  
await page.GetByLabel("Email").ClickAsync();  
  
// Эти действия можно удалить для упрощения теста  
await page.GetByLabel("Email").PressAsync("Tab");
```

Можно запускать Playwright Inspector с параметрами для эмуляции размеров окна просмотра:

```
pwsh bin/Debug/net8.0/playwright.ps1 codegen --viewport-size=800,600 https://  
localhost:5001/
```

Или эмулировать работу на определенном устройстве:

```
pwsh bin/Debug/net8.0/playwright.ps1 codegen --device="iPhone 13" https://localhost:5001/
```



**В копилку знаний:** больше информации о генерации тестов с помощью Playwright Inspector вы найдете на сайте [playwright.dev/dotnet/docs/codegen](https://playwright.dev/dotnet/docs/codegen).

## Тестирование веб-сервисов с помощью xUnit

Существует несколько способов тестирования веб-сервисов.

- Использование фреймворков для автоматизированного тестирования, таких как xUnit, NUnit и MSTest.
- Применение редакторов HTTP-запросов, например расширения REST Client для VS Code или встроенного редактора HTTP Editor в Visual Studio.
- Работа с инструментами графического интерфейса, такими как Postman, NSwag или Swagger UI.



**Внимание!** Пакеты интеграции Swashbuckle будут удалены в .NET 9. Подробнее об изменениях можно узнать на странице [github.com/dotnet/aspnetcore/issues/54599](https://github.com/dotnet/aspnetcore/issues/54599).

Команда ASP.NET Core планирует встроить функцию генерации документов OpenAPI непосредственно в платформу, но не собирается создавать замену существующему пользовательскому интерфейсу для тестирования.

Один из самых сложных аспектов тестирования веб-сервисов — организация их размещения во время тестирования и правильная имитация HTTP-контекста.

Для интеграционных тестов проектов на ASP.NET Core обычно требуется следующее.

- Тестовый проект, в котором будут храниться и выполняться тесты. Такой проект ссылается на сайт или веб-сервис, который тестируется (тестируемая система, SUT — System Under Test).
- Тестовый проект создает тестовый веб-хост для SUT и использует клиент тестового сервера для отправки запросов и получения ответов от SUT.

Пакет `Microsoft.AspNetCore.Mvc.Testing` упрощает эту задачу:

- автоматически устанавливает корневой каталог содержимого на корень проекта SUT, чтобы во время выполнения тестов можно было находить статические файлы, страницы и представления;
- предоставляет класс `WebApplicationFactory`, который помогает легко загружать SUT с помощью класса `TestServer`.

Мы создадим небольшой погодный сервис, применив встроенный шаблон проекта, а затем напишем для него интеграционные тесты.

Класс `WebApplicationFactory<Program>` используется для создания экземпляра тестируемого приложения, где `Program` — это класс, содержащий метод `Main` минимального API.

Метод `CreateClient` из класса `WebApplicationFactory<T>` создает объект `HttpClient`, настроенный для отправки запросов к этому экземпляру.

Первый тест отправит GET-запрос на конечную точку `/weatherforecast` и подтвердит успешность ответа, проверяя, что код состояния находится в диапазоне 200–299.

Второй тест выполнит тот же запрос и проверит, что в десериализованном JSON-ответе содержатся пять прогнозов погоды.

Благодаря использованию метода `WebApplicationFactory<T>` тесты автоматически запускают и останавливают тестовый сервер. Вы сможете выполнять тестирование в среде, очень близкой к реальной, без необходимости вручную разворачивать или размещать веб-сервис.

## Создание веб-сервиса, готового к тестированию

Подготовим веб-сервис, с которым будем работать.

1. В редакторе кода создайте проект веб-API и добавьте его в решение `Chapter14`:

- выберите шаблон проекта `ASP.NET Core Web API` (`Веб-API ASP.NET Core`) / `webapi`;
- в поле `Solution name` (`Имя решения`) укажите значение `Chapter14`;
- в поле `Project name` (`Имя проекта`) укажите значение `WeatherService`;
- в раскрывающемся списке `Authentication type` (`Тип проверки подлинности`) выберите пункт `None` (`Отсутствует`);
- установите флажок `Configure for HTTPS` (`Настроить для HTTPS`);
- снимите флажок `Enable Docker` (`Включить поддержку контейнера`);
- установите флажок `Enable OpenAPI support` (`Включить поддержку OpenAPI`);
- установите флажок `Do not use top-level statements` (`Не использовать операторы верхнего уровня`);



**Дельный совет:** если веб-сервис нужно сделать удобным для тестирования, лучше избегать использования функций программы верхнего уровня. Вместо этого стоит явно определить класс `Program` и пометить его модификатором `public`, чтобы класс был доступен за пределами проекта.

- снимите флажок **Use controllers** (Использовать контроллеры);
  - снимите флажок **Enlist in .NET Aspire orchestration** (Включить в список проект оркестрации .NET Aspire).
2. В файле `WeatherService.csproj` добавьте к классу `Program` модификатор `public`.
  3. Выполните сборку проекта `WeatherService`.
  4. При желании запустите проект `WeatherService` и убедитесь, что конечная точка `/weatherforecast` возвращает случайные прогнозы погоды. Когда проверка будет завершена, закройте браузер и остановите веб-сервер.

## Создание тестового проекта

Теперь можно создать тестовый проект, который будет ссылаться на проект веб-сервиса.

1. В редакторе кода создайте C#-проект `xUnit Test Project` (Тестовый проект xUnit / `xunit`, присвойте ему имя `WebServiceTests` и добавьте в решение `Chapter14`).
2. В файле `WebServiceTests.csproj` проекта добавьте ссылку на пакет тестирования ASP.NET Core:

```
<PackageReference Version="8.0.5" Include="Microsoft.AspNetCore.Mvc.Testing" />
```



Актуальную версию пакета `Microsoft.AspNetCore.Mvc.Testing` можно найти по адресу [www.nuget.org/packages/Microsoft.AspNetCore.Mvc.Testing/](http://www.nuget.org/packages/Microsoft.AspNetCore.Mvc.Testing/).

3. В файле `WebServiceTests.csproj` проекта добавьте ссылку на проект `WeatherService`, чтобы тесты могли взаимодействовать с сервисом:

```
<ItemGroup>
  <ProjectReference Include="..\WeatherService\WeatherService.csproj" />
</ItemGroup>
```

4. В проекте `WebServiceTests` переименуйте файл `UnitTest1.cs` в `WeatherForecastTests.cs`.
5. В файле `WeatherForecastTests.cs` определите класс с тестовыми методами:

```
using Microsoft.AspNetCore.Mvc.Testing; // Подключение класса
   // WebApplicationFactory<T>
using System.Net.Http.Json; // Подключение метода ReadFromJsonAsync
using WeatherService;      // Подключение класса Program

namespace WebServiceTests;

public class WeatherForecastTests :
    IClassFixture<WebApplicationFactory<Program>>
```

```
{  
    private readonly WebApplicationFactory<Program> _factory;  
    private const string relativePath = "/weatherforecast";  
  
    public WeatherForecastTests(WebApplicationFactory<Program> factory)  
    {  
        _factory = factory;  
    }  
  
    [Fact]  
    public async Task Get_WeatherForecasts_ReturnsSuccessStatusCode()  
    {  
        // Подготовка среды для теста  
        HttpClient client = _factory.CreateClient();  
  
        // Выполнение запроса  
        HttpResponseMessage response = await client.GetAsync(relativePath);  
  
        // Проверка результата  
        Assert.True(response.IsSuccessStatusCode); // Код состояния 200-299.  
    }  
  
    [Fact]  
    public async Task Get_WeatherForecasts_ReturnsFiveForecasts()  
    {  
        // Подготовка среды для следующего теста  
        HttpClient client = _factory.CreateClient();  
  
        // Выполнение запроса  
        HttpResponseMessage response = await client.GetAsync(relativePath);  
        WeatherForecast[]? forecasts = await response.Content  
            .ReadFromJsonAsync<WeatherForecast[]>();  
  
        // Проверка результата  
        Assert.NotNull(forecasts);  
        Assert.True(forecasts.Length == 5);  
    }  
}
```

6. Запустите тесты в проекте `WebServiceTests` и обратите внимание, что они проходят успешно без необходимости отдельного запуска проекта веб-сервиса. Тесты выполняются с помощью встроенного экземпляра `TestServer`, созданного в тестовом проекте.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

## Упражнение 14.1. Дополнительные онлайн-материалы

Официальную документацию по Playwright для .NET можно найти на сайте [playwright.dev/dotnet/docs/intro](https://playwright.dev/dotnet/docs/intro).

## Упражнение 14.2. Практические задания

Если вы завершили работу с Playwright и хотите освободить место, можно удалить установленные версии браузеров (`chromium`, `firefox`, `webkit`) из текущей установки. Для этого запустите PowerShell-сценарий с параметром удаления:

```
pwsh bin/Debug/net8.0/playwright.ps1 uninstall
```

Чтобы удалить браузеры из всех установок Playwright, добавьте ключ `--all`:

```
pwsh bin/Debug/net8.0/playwright.ps1 uninstall --all
```

## Упражнение 14.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Какие браузеры используются при выполнении тестов в Playwright?
2. Какие ключевые интерфейсы отвечают за работу с важными объектами при написании тестов в Playwright?
3. Какими способами в Playwright можно получить один или несколько элементов на веб-странице?
4. Что произойдет, если совпадет несколько элементов, а вы вызовете метод, предназначенный для одного элемента DOM, например `ClickAsync`?
5. Для чего используется Playwright Inspector?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 14.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-14---functional-testing-of-websites-and-services](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-14---functional-testing-of-websites-and-services).

## Резюме

Что мы изучили в этой главе:

- разницу между функциональным и сквозным тестированием;
- как тестировать пользовательский веб-интерфейс с помощью Playwright;
- как генерировать тесты с помощью Playwright Inspector;
- как тестировать веб-сервисы с помощью xUnit.

В следующей главе мы перейдем к изучению контейнеризации приложений с помощью Docker.

# 15 Контейнеризация с помощью Docker

В этой главе рассматривается концепция контейнеризации — технологии, которая объединяет приложения вместе с их зависимостями в изолированные блоки, обеспечивая стабильную работу в разных вычислительных средах.

В частности, вы познакомитесь с практическим применением контейнеров с помощью Docker — ведущей платформы, упрощающей создание, развертывание и управление контейнерными приложениями. Мы разберем основные компоненты Docker, его экосистему и реальные сценарии использования, чтобы вы получили все необходимые инструменты и навыки для эффективной работы с контейнерами в процессе разработки.

В отличие от традиционных *виртуальных машин (ВМ)*, эмулирующих операционные системы целиком, контейнеры используют ядро операционной системы хоста, поэтому они легче и эффективнее.

## Знакомство с контейнеризацией

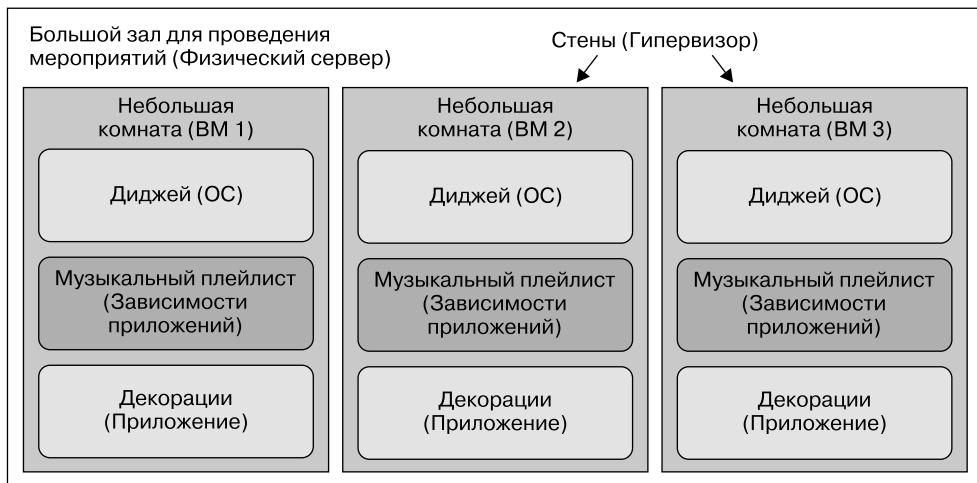
Контейнеризация — это подход, который помогает упростить разработку, развертывание и выполнение ПО, повышая их эффективность, стабильность и удобство масштабирования.

Если воспользоваться простой аналогией, представьте, что вы организуете несколько тематических вечеринок (приложения) в большом зале для мероприятий (физический сервер). Для каждой вечеринки потребуются свое пространство, декорации, диджей (операционная система) и музыкальный плейлист (зависимости приложений), чтобы создать нужную атмосферу.

Использование виртуальных машин можно сравнить с арендой нескольких отдельных комнат в зале для мероприятий. Каждая комната отделена стенами (гипервизор), запирается (изолированная операционная система) и имеет собственные декорации и музыку. Принцип показан на рис. 15.1.

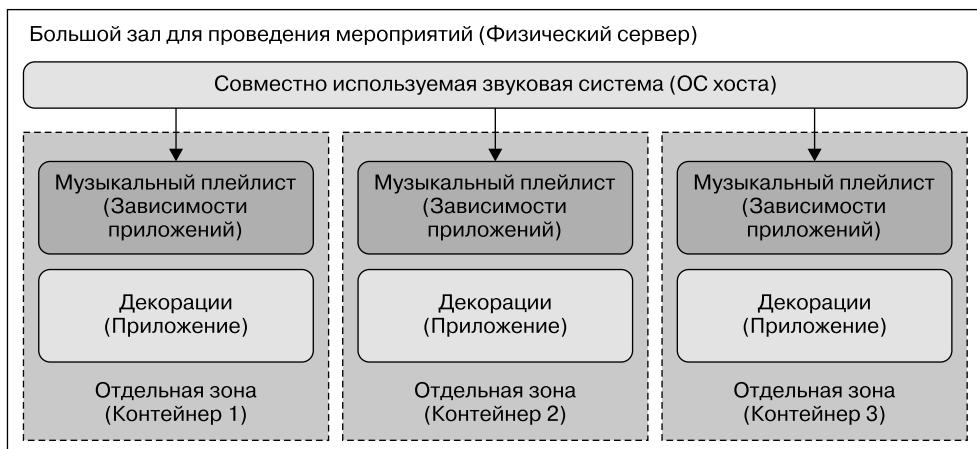
При такой организации каждая вечеринка проходит независимо от других, и у каждой есть все необходимое для полноценной работы. Однако аренда отдельных помещений и их обустройство требуют больше пространства, времени и ресурсов,

поскольку каждую комнату приходится оснащать с нуля, включая собственную звуковую систему и диджея.



**Рис. 15.1.** Виртуализация похожа на аренду небольших комнат

Контейнеризация напоминает решение провести все эти вечеринки в одном большом зале, отделяя их друг от друга с помощью подвижных перегородок (время выполнения контейнера). Вместо того чтобы на каждой вечеринке был свой диджей и своя звуковая система, используется одна большая общая звуковая система (ядро операционной системы хоста), которой пользуются все вместе. При этом каждая зона сохраняет свои декорации и музыкальный плейлист (приложение и его зависимости), а базовая инфраструктура остается общей (рис. 15.2).



**Рис. 15.2.** Большой зал с подвижными перегородками для разделения пространства

Представьте, что для полной изоляции каждой вечеринки придумали технологию «конуса молчания», которая не дает музыке мешать другим.

Такой подход помогает экономить место и средства, потому что вам не нужно дублировать дорогостоящие компоненты — например, диджеев и звуковые системы. Переставить перегородки или изменить тематику вечеринок гораздо проще и быстрее, чем сносить и перестраивать целые комнаты. К тому же в одном и том же помещении можно провести больше вечеринок, ведь каждая из них занимает меньше пространства.

Если сравнивать, виртуальные машины похожи на аренду отдельных комнат: полная изоляция, но более высокая стоимость и дополнительные накладные расходы. Контейнеризация — это разумное разделение пространства, которое помогает эффективнее и гибче размещать участников. Оба подхода имеют свои сильные стороны и выбираются в зависимости от потребностей мероприятия (или приложения). Но контейнеризация дает возможность лучше использовать ресурсы и упростить настройку во многих случаях.

Контейнеризация упаковывает ПО в полноценную файловую систему, где собрано все необходимое для работы — код, время выполнения, системные инструменты и библиотеки. Это гарантирует, что программа будет работать одинаково, где бы она ни была развернута. Можно сказать, что контейнер — это «контейнер кода», который одинаково хорошо запускается и на вашем ноутбуке, и в частном центре обработки данных, и в облаке.

Такое единство помогает решить давнюю проблему при разработке и развертывании ПО — так называемый синдром «у меня на компьютере все работает». Благодаря контейнерам разработчики могут сосредоточиться на своем коде, не задумываясь о том, в какой именно среде он будет выполняться.

## Устройство и преимущества контейнеров

Контейнеры работают на ядре операционной системы одного компьютера и используют его совместно с другими контейнерами. Они легкие, потому что им не нужна дополнительная нагрузка в виде гипервизора, который управляет виртуальными машинами. Контейнеры запускаются напрямую в ядре хост-машины (компьютера), что делает их более эффективными, быстрыми и менее ресурсоемкими по сравнению с традиционными виртуальными машинами, где для каждой требуется полноценная операционная система.

Преимущества контейнеризации.

- **Портативность.** После создания контейнер можно запустить где угодно, что позволяет с уверенностью переносить приложения между различными средами.
- **Согласованность.** Контейнеры обеспечивают единообразную среду от этапа разработки до эксплуатации, уменьшая количество ошибок и несоответствий.

- **Изоляция.** Каждый контейнер работает отдельно и не мешает другим контейнерам или хост-системе.
- **Эффективность.** Контейнеры используют системные ресурсы более экономно, чем виртуальные машины, позволяя получить больше пользы от оборудования.
- **Масштабируемость.** Контейнеры легко увеличивать или уменьшать в зависимости от нагрузки и потребностей.
- **Быстрое развертывание.** Контейнеры можно создавать и удалять за считанные секунды, что помогает быстро адаптироваться к меняющимся требованиям.

Ни одно технологическое решение не бывает идеальным, поэтому рассмотрим некоторые потенциальные недостатки контейнеризации.

- **Проблемы безопасности.** Поскольку контейнеры используют общее ядро операционной системы хоста, уязвимость в ядре может поставить под угрозу все контейнеры на компьютере. Хотя контейнеры обеспечивают изоляцию процессов, она уступает аппаратной изоляции, которую дают виртуальные машины. Это повышает риск атак на выход из контейнера, когда вредоносный контейнер получает доступ к хост-системе.
- **Сложность управления.** При эксплуатации большого количества контейнеров требуются инструменты оркестрации вроде Kubernetes. Они упрощают управление, но сами по себе добавляют сложности и требуют серьезных знаний для настройки и поддержки. На этапе разработки работу с контейнерами можно значительно упростить с помощью стека .NET Aspire, с которым вы познакомитесь в следующей главе. Сетевое взаимодействие между контейнерами тоже может вызывать сложности, особенно при работе с несколькими хостами, обнаружением сервисов и настройкой сетевых политик.
- **Управление постоянными хранилищами.** Контейнеры изначально рассчитаны на временное существование, поэтому организация постоянного хранения данных требует особого внимания. Нужно тщательно настраивать мониторинг томов или использовать сетевые хранилища, чтобы обеспечить надежное хранение и согласованность данных.
- **Проблемы совместимости.** Не все приложения легко контейнеризуются. Особенно это касается устаревших систем, которые зависят от специфического оборудования или особенностей операционной системы.
- **Проблемы с отладкой.** Многослойная структура контейнерных образов и сложность оркестрации иногда затрудняют отладку по сравнению с традиционными приложениями.

Хотя контейнеризация дает серьезные преимущества в эффективности, масштабируемости и согласованности, перед ее применением стоит учитывать и возможные ограничения.

Теперь перейдем к конкретным технологиям, которые лежат в основе контейнеризации.

## Docker, Kubernetes и .NET Aspire

Docker — это платформа, благодаря которой контейнеризация стала доступной для разработчиков и завоевала популярность. В ней реализован открытый стандарт для упаковки и распространения контейнерных приложений. Благодаря Docker создание, доставка и запуск приложений упрощаются, а рабочие процессы разработки становятся более предсказуемыми и масштабируемыми. Приложение вместе с его зависимостями упаковывается в контейнер, который можно запустить на любом сервере Linux или Windows, поддерживающем Docker. Контейнеры тратят мало ресурсов, потому на одном хосте можно размещать множество приложений.

Помимо Docker, существует целая экосистема инструментов, поддерживающих контейнеризацию. В частности, системы оркестрации, такие как Kubernetes, автоматизируют развертывание, масштабирование и управление контейнерными приложениями в рабочей среде.

А в процессе локальной разработки аналогичную роль выполняет стек .NET Aspire, помогая упростить работу с контейнерами прямо на компьютере разработчика.

Docker и Kubernetes выполняют разные, но дополняющие друг друга роли в контейнеризации. Если представить процесс как организацию доставки товаров, Docker будет системой упаковки, которая упаковывает товары (приложения и их зависимости) в компактные и удобные для транспортировки контейнеры. Kubernetes станет центром доставки, который управляет тем, куда отправляются контейнеры, где они хранятся и как масштабируются.

Docker предоставляет инструменты для управления жизненным циклом контейнеров: создание образов, запуск контейнеров, их перемещение, управление версиями и др. Он упрощает процесс создания контейнеров, делая его доступным даже для начинающих.

Контейнеры Docker можно интегрировать в конвейеры CI/CD, чтобы автоматизировать развертывание, ускорить его и снизить вероятность ошибок.



Podman — альтернатива Docker, совместимая с ним и поддерживаемая стеком .NET Aspire. Подробнее о Podman можно узнать на сайте [podman.io](https://podman.io).

## Kubernetes

Kubernetes — это система оркестрации контейнеров для Docker и других контейнерных платформ. Kubernetes помогает управлять контейнерами в масштабе: определяет, где и как их запускать, как они взаимодействуют между собой и как их масштабировать в зависимости от нагрузки.

Система поддерживает автоматическое развертывание, масштабирование и управление контейнерами приложений в кластерах компьютеров. Она берет на себя сетевое взаимодействие между контейнерами, балансировку нагрузки, обеспечение

безопасности и работу с хранилищами. Kubernetes создает инфраструктуру для построения облака на базе контейнеров и используется, когда нужно управлять большим числом контейнеров в кластере. Эта система особенно подходит для рабочих сред, где важны высокая доступность, масштабируемость и распределенная архитектура.

Хотя у Docker есть собственные инструменты оркестрации, например Docker Swarm, Kubernetes обеспечивает лучшую масштабируемость, большую гибкость и получает активную поддержку сообщества.

Docker упрощает создание и управление контейнерами, а Kubernetes берет готовые контейнеры и помогает эффективно развертывать их, оптимизируя использование ресурсов, автоматизируя масштабирование и восстановление приложений.



Эта книга посвящена инструментам и навыкам, которые доступны разработчикам .NET на локальных компьютерах. Развортывание в эксплуатацию выходит за рамки этой книги, поэтому Kubernetes здесь подробно не рассматривается. Многие организации считают Kubernetes избыточным для своих задач и предпочитают более простые способы развертывания, например cron-схемарии, которые автоматически запускаются на основе меток контейнеров Docker.

## .NET Aspire

.NET Aspire — это часть экосистемы .NET, которая помогает модернизировать платформу с использованием новейших технологий и практик. В частности, она опирается на облачные подходы, архитектуру микросервисов и технологии контейнеризации, такие как Docker. Контейнеризация приложений .NET с помощью Docker позволяет разработчикам использовать преимущества переносимости, эффективности и изоляции, которые дают контейнеры.



Стек .NET Aspire особенно полезен для локальной разработки .NET. Он подробно рассматривается в главе 16.

## Реестры контейнеров

Реестр контейнеров — это система для хранения и распространения образов контейнеров. Его можно представить как библиотеку или репозиторий, где хранятся разные версии образов, которые затем используются для развертывания и запуска контейнеров. Такой подход упрощает обмен, хранение и управление образами контейнеров в командах разработчиков и рабочих средах. Реестры контейнеров часто поддерживают функции контроля версий, управления доступом и сканирования безопасности, помогая убедиться, что образы не содержат уязвимостей.

Наиболее распространенные реестры, которые используют разработчики .NET, приведены в табл. 15.1.

**Таблица 15.1.** Распространенные реестры контейнеров

Реестр	Описание
<b>Docker Hub</b>	Основной публичный реестр образов Docker. Здесь размещаются официальные образы от производителей ПО, проектов с открытым исходным кодом и участников сообщества. Пользователи могут загружать и приватные образы. В Docker Hub вы найдете официальные образы .NET и ASP.NET Core, а также образы сообщества и сторонних разработчиков, которые можно использовать в проектах .NET. Ссылка: <a href="http://hub.docker.com">hub.docker.com</a>
<b>Microsoft Artifact Registry</b> (ранее Microsoft Container Registry, MCR)	Собственный реестр Microsoft для хранения образов контейнеров и других артефактов, включая .NET. Начиная с версии .NET 5 Microsoft рекомендует использовать его как основной источник образов .NET. Образы можно находить через Docker Hub, но фактическая загрузка происходит напрямую из Microsoft Artifact Registry. Благодаря этому Microsoft контролирует качество и актуальность образов. В реестре также размещаются проверенные и актуальные образы Windows Server, SQL Server и других продуктов. Ссылка: <a href="http://mcr.microsoft.com">mcr.microsoft.com</a>
<b>GitHub Container Registry (GHCR)</b>	Интегрирован с GitHub и позволяет публиковать и использовать образы контейнеров в экосистеме GitHub. Поддерживает хранение образов Docker и интеграцию с GitHub Actions для автоматизации CI/CD. Удобен для разработчиков .NET, которые размещают код на GitHub и используют контейнеризацию в рабочих процессах. Ссылка: <a href="http://github.com/container-registry/">github.com/container-registry/</a>
<b>Azure Container Registry (ACR)</b>	Платный частный реестр контейнеров компании Microsoft для хранения и управления образами, предназначенный для развертывания в Azure. Обеспечивает тесную интеграцию с Azure Kubernetes Service (AKS), Azure DevOps и другими службами Azure. Поддерживает георепликацию, сканирование уязвимостей и другие функции безопасности. Идеально подходит для разработчиков .NET, работающих с Azure. Ссылка: <a href="http://azure.microsoft.com/ru-ru/products/container-registry/">azure.microsoft.com/ru-ru/products/container-registry/</a>
<b>Amazon Elastic Container Registry (ECR)</b>	Платный частный реестр контейнеров, интегрированный с Amazon Web Services (AWS). Позволяет хранить, управлять и развертывать образы контейнеров Docker, оптимизирован для использования с Amazon Elastic Container Service (ECS) и Amazon Elastic Kubernetes Service (EKS). Удобен для разработчиков .NET, развертывающих приложения на AWS. Ссылка: <a href="http://aws.amazon.com/ecr/">aws.amazon.com/ecr/</a>



В табл. 15.1 перечислены самые распространенные реестры контейнеров. Существуют и другие варианты, например реестры, размещенные на сайтах GitLab и Harbor ([goharbor.io](http://goharbor.io)).

При выборе реестра контейнеров разработчикам .NET необходимо учитывать интеграцию с текущими рабочими процессами, функции безопасности, удобство использования и стоимость. Публичные реестры, такие как Docker Hub, удобны

для доступа к общедоступным образам и обмена ими. Приватные реестры, такие как ACR и ECR, обеспечивают больший контроль и безопасность для корпоративных приложений.

Интеграция с конвейерами CI/CD и облачными платформами тоже важна, поэтому GHCR, ACR и ECR часто выбирают разработчики .NET, работающие в соответствующих экосистемах.

Контейнеризация меняет подход к созданию и развертыванию приложений. Она предлагает стандартизованный способ упаковки решения и его зависимостей в модули, которые стабильно работают в любой среде. Контейнеризация помогает улучшить рабочие процессы разработки, повысить эффективность инфраструктуры и упростить масштабирование приложений.

Наиболее популярной технологией контейнеризации для локальной работы разработчиков .NET остается Docker. В следующих разделах этой главы мы рассмотрим его подробнее.

## Основные концепции Docker

При работе с Docker важно понимать его базовые концепции и терминологию. Это необходимо для эффективной разработки, развертывания и управления контейнерными приложениями. В табл. 15.2 перечислены ключевые термины и понятия Docker, которые желательно знать каждому разработчику .NET.

**Таблица 15.2.** Основные понятия и концепции Docker

Понятие	Описание
Docker Engine	Основное ПО для управления контейнерами. Позволяет создавать и запускать контейнеризированные приложения. Это движок, который запускает контейнеры на основе образов Docker
Docker Image	Легкий автономный исполняемый пакет, включающий все необходимое для запуска приложения: код, среду выполнения, библиотеки, переменные среды и конфигурационные файлы. Образы неизменяемы и служат шаблонами для контейнеров
Docker Registry	Система для хранения и распространения образов Docker. Позволяет загружать и извлекать образы с помощью команд Docker. Примеры: Docker Hub, MCR
Docker Container	Экземпляр образа Docker. Контейнер запускает приложение в изолированной среде и может работать на любом компьютере с установленным Docker. Контейнеры изолированы друг от друга и от хоста, но могут взаимодействовать по сети
Dockerfile	Текстовый файл с командами для сборки образа. Позволяет автоматизировать создание образов Docker

*Продолжение ↗*

**Таблица 15.2** (окончание)

Понятие	Описание
<b>Docker Compose</b>	Инструмент для описания и запуска многоконтейнерных приложений. Настройки сервисов, сетей и томов задаются в YAML-файле, после чего все сервисы запускаются одной командой
<b>Docker Daemon</b>	Фоновый процесс, управляющий сборкой, запуском и распространением контейнеров. Демон работает на хосте и обрабатывает команды от клиента
<b>Docker Client</b>	Интерфейс командной строки для взаимодействия с демоном Docker. Пользователь отправляет команды (например, <code>docker run</code> ), которые выполняет демон
<b>Docker Volume</b>	Механизм хранения данных, создаваемых и используемых контейнерами. Тома изолированы от основной файловой системы хоста и позволяют сохранять или совместно использовать данные
<b>Docker Network</b>	Функция сетевого взаимодействия между контейнерами и с внешним миром через хост-машину. Поддерживает разные сетевые драйверы, например мостовые, хостовые и оверлейные сети
<b>Docker Swarm</b>	Средство кластеризации, объединяющее несколько хостов Docker в общий виртуальный ресурс. Использует стандартный API Docker и позволяет масштабировать приложения на несколько компьютеров

По мере знакомства с Docker вы увидите, что эти концепции лежат в основе большинства операций и рабочих процессов в экосистеме Docker.

Теперь перейдем к важным инструментам и технологиям, связанным с Docker.

## Инструменты и технологии Docker

В табл. 15.3 представлен список инструментов и технологий для разработчиков .NET, которые работают с Docker.

**Таблица 15.3.** Инструменты и технологии Docker

Инструмент	Описание
<b>Docker Desktop</b>	Упрощенный интерфейс для управления контейнерами и образами: создание, запуск, остановка контейнеров. Доступен для Windows, macOS и Linux. Установка Docker Desktop будет частью практического задания в следующем разделе
<b>Docker Compose</b>	Инструмент для определения и запуска многоконтейнерных приложений. Одна команда запускает весь стек, включая службы, сети и тома, как указано в файле <code>docker-compose.yml</code> . Стек .NET Aspire предоставляет аналогичные возможности на C# в проекте Aspire. Docker Compose входит в комплект Docker Desktop

Инструмент	Описание
Docker Scout	Сервис для анализа образов, выявления пакетов и уязвимостей, а также предоставления рекомендаций по их устранению. Доступен через Docker Desktop и требует отдельной подписки
Расширения для редакторов кода	Добавляют поддержку Docker в проекты, позволяют создавать Dockerfile, управлять контейнерами и образами прямо из редактора. Расширения устанавливаются через магазин расширений редактора кода

## Команды интерфейса командной строки (CLI) Docker

В Docker доступен набор команд CLI для управления контейнерами, образами, томами и сетями. В табл. 15.4 приведены основные команды (а также примеры), которые формируют основу для работы с Docker и управления жизненным циклом контейнеров, образов и других ресурсов.

**Таблица 15.4.** Команды Docker

Пример команды	Описание
docker --help	Отображает справку по командам CLI
docker <команда> --help	Отображает справку по указанной команде, например docker pull --help
docker images	Выводит список всех локальных образов
docker pull <имя_образа>:<тег>	Загружает образ с указанным тегом из реестра контейнеров. Пример: docker pull mcr.microsoft.com/dotnet/sdk:8.0
docker rmi mcr.microsoft.com/dotnet/sdk:8.0	Удаляет один или несколько образов
docker build -t my-dotnet-app .	Собирает образ из файла Dockerfile
docker run -d -p 8080:80 --name webapp mcr.microsoft.com/dotnet/aspnet:8.0	Создает и запускает контейнер из образа
docker ps	Выводит список запущенных контейнеров
docker ps -a	Выводит список всех контейнеров, включая остановленные
docker stop webapp	Останавливает запущенный контейнер
docker start webapp	Запускает остановленный контейнер

*Продолжение ↗*

**Таблица 15.4** (окончание)

Пример команды	Описание
<code>docker restart webapp</code>	Перезапускает контейнер
<code>docker rm webapp</code>	Удаляет один или несколько контейнеров
<code>docker logs webapp</code>	Выводит журналы контейнера
<code>docker exec -it webapp bash</code>	Открывает интерактивную сессию внутри контейнера
<code>docker network create my-network</code>	Создает сеть
<code>docker network ls</code>	Выводит список всех сетей
<code>docker network rm my-network</code>	Удаляет одну или несколько сетей
<code>docker network connect my-network webapp</code>	Подключает контейнер к сети
<code>docker network disconnect my-network webapp</code>	Отключает контейнер от сети
<code>docker volume create my-volume</code>	Создает том для хранения данных
<code>docker volume ls</code>	Выводит список всех томов
<code>docker volume rm my-volume</code>	Удаляет один или несколько томов

Эти команды полезны для ежедневной работы — от создания приложений до их развертывания в рабочую среду. Овладение ими значительно упростит и ускорит работу с Docker.



**В копилку знаний:** официальную документацию по командам Docker CLI можно найти на сайте [docs.docker.com/reference/cli/docker/](https://docs.docker.com/reference/cli/docker/).

## Создание образов с помощью файлов Dockerfile

Dockerfile — это текстовый документ, содержащий команды для сборки образа. Формат Dockerfile разработан так, чтобы быть простым и читаемым. Он состоит из инструкций, каждая из которых выполняет определенную роль в процессе создания образа.

Основная структура выглядит так:

```
# Комментарий
INSTRUCTION arguments
```

Комментарии начинаются с символа # и продолжаются до конца строки. В табл. 15.5 приведены некоторые распространенные инструкции Dockerfile.

**Таблица 15.5.** Распространенные инструкции в Dockerfile

Инструкция	Описание
FROM	Определяет базовый образ для сборки. Это всегда первая инструкция (если не считать директивы синтаксического анализатора, комментариев и инструкций ARG, которые могут идти перед ней). Пример: FROM ubuntu:20.04
LABEL	Добавляет метаданные к образу, например автора, версию или описание. Пример: LABEL maintainer=sam.smith@example.com
RUN	Выполняет команды поверх текущего слоя образа и сохраняет результаты. Используется для установки программ и зависимостей. Пример: RUN apt-get update && apt-get install -y python3
CMD	Задает настройки по умолчанию для выполнения контейнера. В Dockerfile может быть только одна активная инструкция CMD. Если указано более одной CMD, то в силу вступит последняя. Пример: CMD ["echo", "Hello world"]
EXPOSE	Указывает, что контейнер будет прослушивать заданный порт во время работы. Фактически публикацию порта не выполняет. Пример: EXPOSE 80
ENV	Устанавливает переменные среды. Пример: ENV MY_NAME="Sam Smith"
ADD, COPY	Копируют файлы, каталоги или удаленные файлы по URL из источника ( <i>src</i> ) в файловую систему образа по указанному пути ( <i>dest</i> ). Пример: COPY ./app/app
ENTRYPOINT	Настраивает контейнер так, чтобы он запускался как исполняемый файл. Аргументы команды docker run <образ> автоматически добавляются к ENTRYPOINT. Пример: ENTRYPOINT ["python3", "-m", "http.server"]
WORKDIR	Устанавливает рабочую директорию для инструкций RUN, CMD, ENTRYPOINT, COPY и ADD, следующих за ней. Пример: WORKDIR /app
USER	Устанавливает пользователя или UID, от имени которого будут выполняться команды RUN, CMD или ENTRYPOINT. Пример: USER myuser
VOLUME	Создает точку монтирования для тома и отмечает ее как место для хранения данных, которые могут быть смонтированы с хоста или других контейнеров. Пример: VOLUME /myvol

Ниже приведен простой пример Dockerfile для базового проекта ASP.NET Core. Этот файл показывает, как собрать и запустить приложение в контейнере Docker. Предполагается, что в проекте определена точка входа Main в файле Program.cs:

```
# Использование образа .NET SDK для сборки приложения
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build-env
WORKDIR /app
```

```
# Копирование файла csproj и восстановление всех зависимостей через NuGet
COPY *.csproj ./  
RUN dotnet restore  
  
# Копирование файлов проекта и сборка релиза
COPY . ./  
RUN dotnet publish -c Release -o out  
  
# Генерация образа для выполнения
FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /app
COPY --from=build-env /app/out .
ENTRYPOINT ["dotnet", "MyApp.dll"]
```

Краткие пояснения к приведенным инструкциям.

- FROM [mcr.microsoft.com/dotnet/sdk:8.0](https://mcr.microsoft.com/dotnet/sdk:8.0) AS build-env: начинает первый этап сборки с использованием образа .NET 8.0 SDK. Часть AS build-env задает имя этапа для дальнейшего обращения к нему.
- WORKDIR /app: устанавливает рабочий каталог внутри контейнера.
- COPY \*.csproj ./: копирует файлы .csproj с хоста в рабочий каталог контейнера.
- RUN dotnet restore: восстанавливает пакеты NuGet, необходимые для проекта.
- COPY . ./: копирует остальные файлы проекта в контейнер.
- RUN dotnet publish -c Release -o out: публикует приложение в папку out в конфигурации релиза.
- FROM [mcr.microsoft.com/dotnet/aspnet:8.0](https://mcr.microsoft.com/dotnet/aspnet:8.0): начинает второй этап на основе более легкого образа среды выполнения ASP.NET Core.
- WORKDIR /app: снова устанавливает рабочий каталог.
- COPY --from=build-env /app/out .: копирует опубликованное приложение из первого этапа.
- ENTRYPOINT ["dotnet", "MyApp.dll"]: задает команду для запуска приложения. Здесь MyApp.dll нужно заменить на имя вашей DLL.

Этот Dockerfile задействует *многоэтапную сборку*: сначала приложение собирается с помощью образа SDK, затем упаковывается в более легкий образ среды выполнения. Такой подход уменьшает размер итогового контейнера, так как в нем остаются только необходимые файлы для запуска.

Сборку можно запустить с помощью команды `docker build`, которая последовательно выполнит все инструкции из Dockerfile.



Официальную документацию по Dockerfile можно найти на сайте [docs.docker.com/reference/dockerfile/](https://docs.docker.com/reference/dockerfile/). Документацию по многоэтапным сборкам — на сайте [docs.docker.com/build/building/multi-stage/](https://docs.docker.com/build/building/multi-stage/).

## Настройка портов и запуск контейнера

Контейнер Docker, который не взаимодействует с внешним миром, малопригоден для работы. Обычно требуется открыть порты контейнера, чтобы приложение стало доступным извне. Для этого команда `docker run` позволяет указать порты с помощью ключа `-p` или `--publish`, сопоставляя порт хоста с портом контейнера. Синтаксис следующий: `-p <hostPort>:<containerPort>`.

Например, для веб-приложения ASP.NET Core, которое внутри контейнера прослушивает порт 8080, можно сопоставить его с портом 8000 на хосте. Тогда приложение будет доступно по адресу `http://localhost:8000`:

```
docker run -p 8000:8080 -d --name myaspnetapp mcr.microsoft.com/dotnet/
aspnet:latest
```

Эта команда запускает контейнер с именем `myaspnetapp` из образа `mcr.microsoft.com/dotnet/aspnet:latest`, отсоединяет его (`-d`) и сопоставляет порт 8000 на хосте с портом 8080 в контейнере.



**Внимание!** В .NET 8 образы контейнеров ASP.NET Core по умолчанию настроены на прослушивание порта 8080. В версиях .NET 6 и .NET 7 по умолчанию использовался порт 80.

.NET-приложения можно также настроить для работы через HTTPS. Если контейнер прослушивает порт 443 (стандартный порт для HTTPS), а на хосте вы хотите использовать, например, порт 8443, команда будет выглядеть так:

```
docker run -p 8443:443 -d --name mysecureapp mcr.microsoft.com/dotnet/
aspnet:latest
```

Так вы получите доступ к приложению через HTTPS по адресу `http://localhost:8443`. Для корректной работы HTTPS может потребоваться дополнительная настройка и установка сертификатов.

Иногда приложениям нужно открыть несколько портов, например, когда веб-интерфейс работает через HTTP, а конечная точка управления или метрик — на другом порту. Допустим, ваше приложение .NET использует порт 80 для веб-интерфейса и порт 8081 для метрик. В этом случае оба порта можно сопоставить:

```
docker run -p 8080:80 -p 5001:8081 -d --name mydotnetapp mcr.microsoft.com/dotnet/
aspnet:latest
```

В этой команде `-p 8080:80` сопоставляет порт HTTP, а `-p 5001:8081` — порт конечной точки метрик.



**Дельный совет:** при работе с контейнерами Docker (особенно в процессе разработки или тестирования) старайтесь использовать порты, которые не заняты другими сервисами на хост-машине. Так вы избежите ошибок связывания.

В некоторых случаях задачу можно разбить на несколько этапов.

1. Загрузить базовый образ.
2. Создать собственный образ на его основе.
3. Запустить контейнер из собственного образа.

На практике чаще всего образы загружают и сразу запускают в одном действии. Команда `docker run` позволяет как загрузить образ (если его еще нет локально), так и сразу запустить контейнер:

```
docker run --rm -it -p 8000:8080 mcr.microsoft.com/dotnet/samples:aspnetapp
```



Ключ `--rm` автоматически удаляет контейнер после его остановки.

## Интерактивный режим

Ключ `-it` в команде `docker run` объединяет два отдельных флага — `-i` и `-t` — чтобы облегчить работу с интерактивными процессами, выделяя псевдо-TTY (телетайп).

- `-i` или `--interactive` удерживает стандартный ввод (STDIN) открытым для контейнера. Флаг позволяет передавать ввод из терминала в контейнер даже без активного подключения.
- `-t` или `--tty` выделяет псевдо-TTY, который эмулирует поведение обычного терминала. Флаг форматирует вывод контейнера в удобный для чтения вид и позволяет вводить команды вручную.

Ключ `-it` позволяет напрямую работать с CLI или приложением внутри контейнера. Например, чтобы запустить контейнер с оболочкой Linux в интерактивном режиме, используйте следующую команду:

```
docker run -it ubuntu /bin/bash
```

Эта команда запустит контейнер Ubuntu, откроет оболочку Bash и позволит взаимодействовать с контейнером прямо из терминала. Вы сможете выполнять команды в оболочке, просматривать вывод и работать с файловой системой или приложениями, запущенными внутри контейнера.

Ключ `-it` необходим для отладки, настройки или в случаях, когда требуется непосредственное взаимодействие с приложением в контейнере. Без `-it` контейнер

будет работать в фоновом режиме, и возможности напрямую управлять им через терминал не будет.

Другой распространенный вариант настройки — передача переменных среды при запуске контейнера.

## Настройка переменных среды

Для работы с некоторыми образами необходимо настроить переменные среды при запуске контейнера. Например, для работы образа SQL Server нужно установить две обязательные переменные и одну дополнительную переменную со значением по умолчанию. Они приведены в табл. 15.6.

**Таблица 15.6.** Переменные среды для образа SQL Server

Переменная среды	Описание
ACCEPT_EULA	Подтверждает согласие с лицензионным соглашением конечного пользователя
MSSQL_SA_PASSWORD	Устанавливает пароль системного администратора (sa) базы данных SQL Server. Пароль должен содержать не менее 8 символов как минимум из четырех категорий: прописные буквы, строчные буквы, цифры и специальные символы
MSSQL_PID	Указывает идентификатор продукта (PID) или издание. Допустимые значения: Developer, Express или идентификатор вашей лицензии. Если значение не задано, по умолчанию используется Developer



**В копилку знаний:** полный список переменных среды для SQL Server доступен на сайте [learn.microsoft.com/ru-ru/sql/linux/sql-server-linux-configure-environment-variables](https://learn.microsoft.com/ru-ru/sql/linux/sql-server-linux-configure-environment-variables).

Переменные среды задаются с помощью ключа `-e`. Например:

```
docker run
-e "ACCEPT_EULA=Y"
-e "MSSQL_SA_PASSWORD=yourStrong(!)Password"
-e "MSSQL_PID=Developer"
-p 1433:1433 --name sqlserver1 --hostname sqlserver1
-d mcr.microsoft.com/mssql/server:2022-CU10-ubuntu-22.04
```



Для удобства чтения команда отформатирована по строкам. При вводе в терминале ее нужно писать в одну строку.

## Распространенные образы контейнеров Docker

Существует множество готовых контейнерных образов, которые можно запускать и использовать в проектах. В табл. 15.7 приведены некоторые из наиболее популярных.

**Таблица 15.7.** Распространенные образы контейнеров Docker

Имя	Ссылка
SQL Server	<a href="http://hub.docker.com/_/microsoft-mssql-server">hub.docker.com/_/microsoft-mssql-server</a>
Azure SQL Edge	<a href="http://hub.docker.com/_/microsoft-azure-sql-edge">hub.docker.com/_/microsoft-azure-sql-edge</a>
PostgreSQL	<a href="http://hub.docker.com/_/postgres">hub.docker.com/_/postgres</a>
RabbitMQ	<a href="http://hub.docker.com/_/rabbitmq">hub.docker.com/_/rabbitmq</a>
Redis	<a href="http://hub.docker.com/_/redis">hub.docker.com/_/redis</a>
MongoDB	<a href="http://hub.docker.com/_/mongo">hub.docker.com/_/mongo</a>
Elasticsearch	<a href="http://hub.docker.com/_/elasticsearch">hub.docker.com/_/elasticsearch</a>
nginx (обратный прокси)	<a href="http://hub.docker.com/_/nginx">hub.docker.com/_/nginx</a>
Grafana	<a href="http://hub.docker.com/r/grafana/grafana">hub.docker.com/r/grafana/grafana</a>
.NET SDK	<a href="http://hub.docker.com/_/microsoft-dotnet-sdk/">hub.docker.com/_/microsoft-dotnet-sdk/</a>
.NET Runtime	<a href="http://hub.docker.com/_/microsoft-dotnet-runtime/">hub.docker.com/_/microsoft-dotnet-runtime/</a>
ASP.NET Core Runtime	<a href="http://hub.docker.com/_/microsoft-dotnet-aspnet/">hub.docker.com/_/microsoft-dotnet-aspnet/</a>
.NET Samples	<a href="http://hub.docker.com/_/microsoft-dotnet-samples/">hub.docker.com/_/microsoft-dotnet-samples/</a>

## Образы контейнеров .NET

Платформа .NET предоставляет разные образы контейнеров для различных сценариев применения. Для их различения используются несколько характеристик.

- Моникер целевого фреймворка (Target Framework Moniker, TFM) приложения.
- Операционная система, версия и архитектура.
- Тип образа (например, `runtime`, `aspnet`, `sdk`).
- Вариант образа (например, `*-distroless`, `*-chiseled`).
- Особенности образа (например, `*-aot`, `*-extra`).



**В копилку знаний:** подробнее о шаблонах и правилах маркировки официальных образов контейнеров .NET можно узнать на странице [github.com/dotnet/docker/blob/main/documentation/supported-tags.md](https://github.com/dotnet/docker/blob/main/documentation/supported-tags.md).

Некоторые образы .NET имеют уменьшенный размер за счет отсутствия глобализационных данных, например информации о часовых поясах. Такие образы подходят только для приложений, настроенных на инвариантный режим глобализации. Чтобы включить инвариантную глобализацию, добавьте в файл проекта следующее свойство:

```
<PropertyGroup>
  <InvariantGlobalization>true</InvariantGlobalization>
</PropertyGroup>
```



**В копилку знаний:** четыре наиболее распространенных официальных образа для .NET и ASP.NET Core приведены в конце табл. 15.7. Полный список доступен на странице [hub.docker.com/\\_/microsoft-dotnet](https://hub.docker.com/_/microsoft-dotnet).

## CVE и Chiseled Ubuntu

*Common Vulnerabilities and Exposures (CVE)* — это стандартизованный идентификатор для конкретной уязвимости или проблемы безопасности. Система CVE создает общую точку отсчета для идентификации и обсуждения уязвимостей в разных инструментах, базах данных и сервисах.

Каждая запись CVE имеет формат CVE-YYYY-NNNN, где:

- YYYY — год, в котором идентификатор был присвоен или обнародован;
- NNNN — порядковый номер начиная с 0001, который увеличивается по мере регистрации новых записей в течение года.

Например, CVE-2023-1234 — это 1234-я зарегистрированная уязвимость за 2023 год.

Каждая запись CVE содержит краткое описание проблемы: указываются характер уязвимости, затронутое программное или аппаратное обеспечение и возможные последствия.

*CVE Numbering Authority (CNA)* — это организации, уполномоченные присваивать идентификаторы CVE уязвимостям, связанным с их продуктами или сферой деятельности. CNA могут быть поставщики ПО, исследовательские группы или компании в области кибербезопасности.

Идентификаторы CVE позволяют специалистам по безопасности, исследователям и поставщикам эффективнее сообщать о конкретных уязвимостях. Это упрощает управление уязвимостями, составление отчетов и устранение последствий. Идентификаторы CVE поддерживаются многими инструментами и базами данных безопасности, включая сканеры уязвимостей, системы обнаружения вторжений и системы управления исправлениями. Такая интеграция облегчает автоматизированное управление уязвимостями и помогает повысить общую безопасность систем.

Образы контейнеров Chiseled представляют собой облегченные версии Ubuntu, включающие только необходимые для работы .NET компоненты. Эти образы примерно на 100 Мбайт меньше обычных образов Ubuntu и содержат меньше CVE, поскольку в них сокращено количество компонентов.

Особенности образов Chiseled:

- нет оболочки и менеджера пакетов, что значительно повышает уровень безопасности;
- включен не-root пользователь для выполнения приложений;
- образы настраиваются для запуска именно от имени этого пользователя.



**В копилку знаний:** более подробную информацию об образах Chiseled можно найти на странице [devblogs.microsoft.com/dotnet/announcing-dotnet-chiseled-containers/](https://devblogs.microsoft.com/dotnet/announcing-dotnet-chiseled-containers/).

## Управление контейнерами с помощью Docker

Хватит теории — давайте перейдем к практике. В этом разделе мы разберемся, как управлять контейнерами с помощью Docker.

### Установка Docker и использование готовых образов

Установим Docker и настроим его для управления контейнерами.

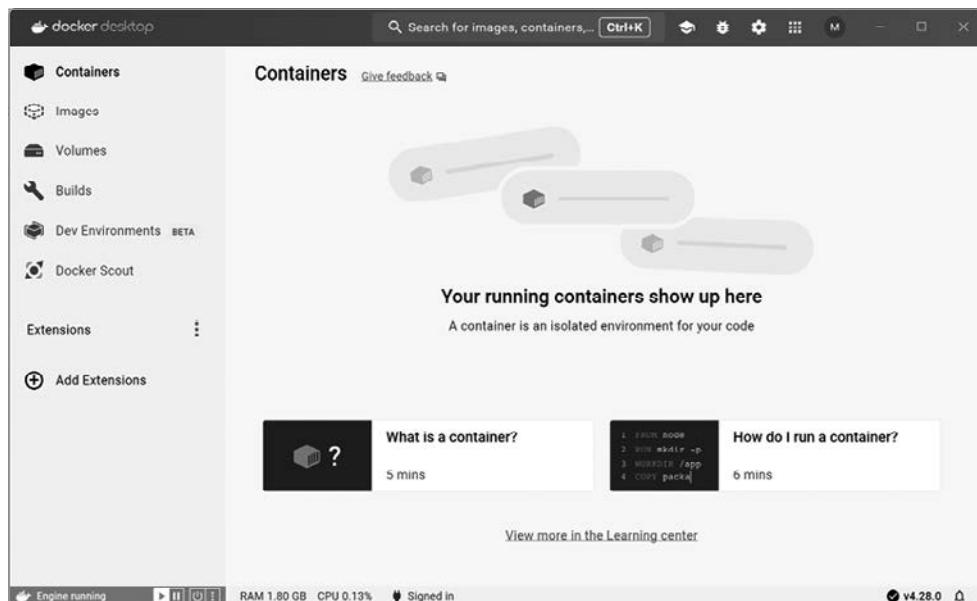
1. Если Docker Desktop еще не установлен на вашем компьютере, скачайте и установите его с сайта [www.docker.com/products/docker-desktop/](https://www.docker.com/products/docker-desktop/).



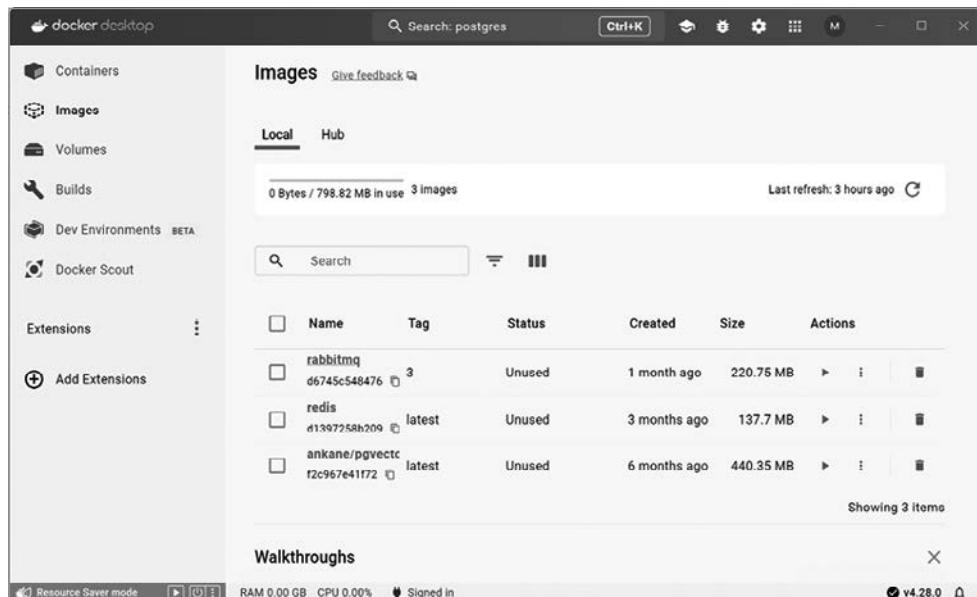
На момент подготовки материала актуальной версией Docker Desktop была 4.28.0. Вы можете увидеть установленную версию в правой части строки состояния. Если появится уведомление о новой версии, рекомендуется установить обновление.

2. Запустите Docker Desktop. Обратите внимание на интерфейс, в частности на представление **Containers** (Контейнеры), которое, скорее всего, будет пустым, если вы еще не запускали контейнеры (рис. 15.3).
3. На панели в левой части программы выберите пункт **Images** (Образы). На моем компьютере уже загружено несколько образов Docker (рис. 15.4).
4. Откройте терминал и выполните команду для отображения списка образов Docker:

```
docker images
```



**Рис. 15.3.** Представление Containers в Docker Desktop



**Рис. 15.4.** Представление Images в Docker Desktop

5. Изучите результаты вывода команды. Они соответствуют образам, показанным на рис. 15.4:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
rabbitmq	3	d6745c548476	4 weeks ago	221MB
redis	latest	d1397258b209	2 months ago	138MB
ankane/pgvector	latest	f2c967e41f72	5 months ago	440MB

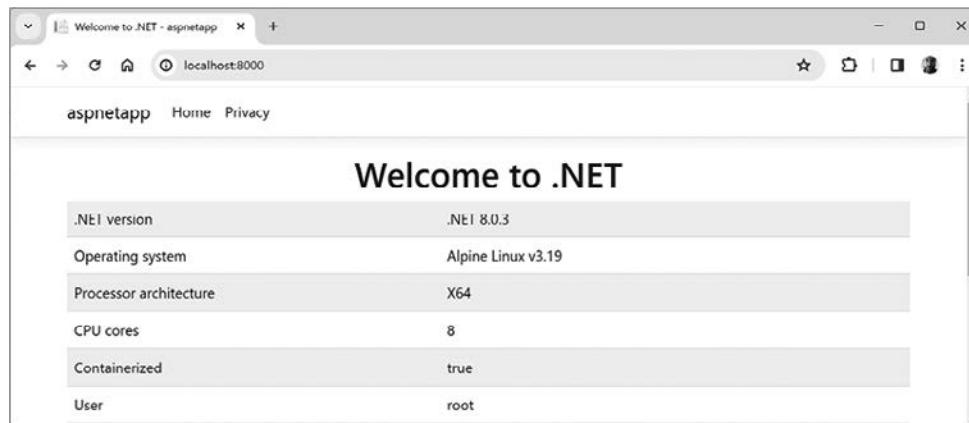
6. В терминале выполните загрузку и запуск примера проекта ASP.NET Core. Сопоставьте внешний порт 8000 с внутренним портом 8080, запустите контейнер в интерактивном режиме (-it) и автоматически удалите его после завершения работы (--rm):

```
docker run --rm -it -p 8000:8080 mcr.microsoft.com/dotnet/samples:aspnetapp
```

7. Просмотрите вывод терминала: будут показаны процессы загрузки образа, запуска контейнера и сообщения от проекта ASP.NET Core:

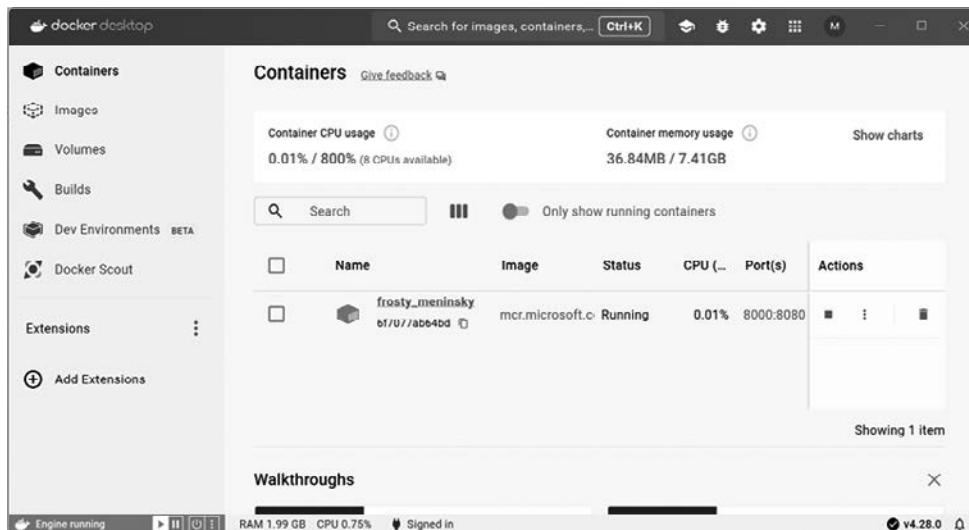
```
Unable to find image 'mcr.microsoft.com/dotnet/samples:aspnetapp' locally
aspnetapp: Pulling from dotnet/samples
4abcf2066143: Pull complete
4e1692478f05: Pull complete
73df137ef55b: Pull complete
0ab1344a44f8: Pull complete
c9a33571af57: Pull complete
458c6e372327: Pull complete
d57ff6e481d4: Pull complete
Digest:
sha256:0bca5fff4b566b29c7d323efc0142ee506681efb31a7839cec91a9acbf760dfa8
Status: Downloaded newer image for mcr.microsoft.com/dotnet/
samples:aspnetapp
warn: Microsoft.AspNetCore.DataProtection.Repositories.
FileSystemXmlRepository[60]
      Storing keys in a directory '/root/.aspnet/DataProtection-Keys' that may
      not be persisted outside of the container. Protected data will be unavailable
      when container is destroyed. For more information go to https://aka.ms/aspnet/
      dataprotectionwarning
warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
      No XML encryptor configured. Key {419c59e8-3d0b-43fa-bf2c-4574734788c4}
      may be persisted to storage in unencrypted form.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://[::]:8080
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /app
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
```

8. Откройте браузер и перейдите по адресу <http://localhost:8000>. Вы увидите сайт, размещенный в контейнере (рис. 15.5).



**Рис. 15.5.** Сайт ASP.NET Core, размещенный в контейнере Docker

9. В Docker Desktop проверьте, что контейнер работает. Он будет запущен с автоматически генерированным именем (рис. 15.6).



**Рис. 15.6.** Запущенный контейнер с рандомным именем

10. Закройте браузер и завершите работу контейнера в терминале.  
 11. Вернитесь в Docker Desktop. Контейнер будет автоматически удален благодаря ключу `--rm`, однако сам образ останется сохраненным на диске.

## Иерархия и слои Docker

Образы Docker строятся на основе многоуровневой архитектуры. Когда вы создаете образ, каждая инструкция в Dockerfile добавляет новый слой. Эти слои образуют иерархию: каждый новый слой опирается на предыдущий (рис. 15.7).

Слой 12: ENTRYPOINT [ "dotnet", "YourApp.dll" ]
Слой 11: COPY --from=publish /app/publish .
Слой 10: WORKDIR /app
Слой 9: RUN FROM mcr.microsoft.com/dotnet/runtime:8.0 AS final
Слой 8: RUN dotnet publish -c Release -o /app/publish
Слой 7: RUN FROM build AS publish
Слой 6: RUN dotnet build -c Release -o /app/build
Слой 5: COPY . ./
Слой 4: RUN dotnet restore
Слой 3: COPY \*.csproj ./
Слой 2: FROM WORKDIR /app
Слой 1: FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base

**Рис. 15.7.** Инструкции Docker и их слои

Каждый слой в образе Docker — это набор отличий по сравнению с предыдущим слоем. Слои накладываются друг на друга: если вы меняете файл, Docker обновляет только соответствующий слой, а не пересобирает весь образ. Благодаря этому создание и передача образов становится быстрее, а их хранение — эффективнее.

Все слои в образе неизменяемы: после создания их нельзя изменить. При внесении изменений создается новый слой, который записывает отличия, сохраняя предыдущую историю нетронутой.

В основании иерархии находится базовый образ — первый слой, на котором строится все остальное. Обычно базовый образ включает минимально необходимую среду, например операционную систему Ubuntu или Alpine.

Поверх базового образа добавляются пользовательские слои, содержащие код приложения, библиотеки, зависимости и переменные среды. Именно эти слои делают образ уникальным и адаптированным под конкретное приложение или сервис.

Рассмотрим Dockerfile для простого приложения ASP.NET Core:

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 # Базовый образ
COPY . /app # Копирование кода приложения
WORKDIR /app # Установка рабочего каталога
```

```
RUN dotnet restore # Восстановление зависимостей
RUN dotnet publish -c Release -o out # Сборка приложения
ENTRYPOINT ["dotnet", "out/myapp.dll"] # Установка команды запуска
```

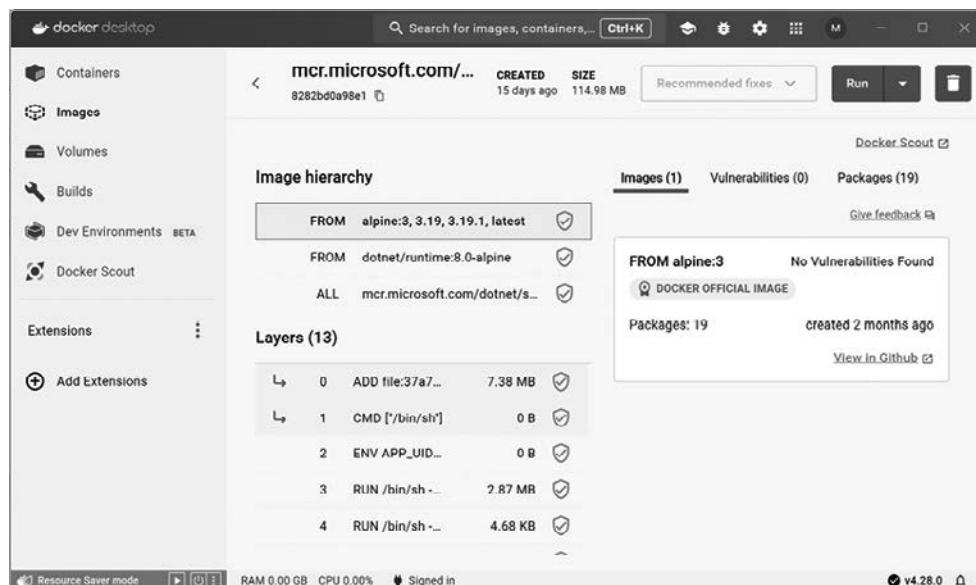
- Инструкция `FROM` задает базовый образ, содержащий среду выполнения .NET.
- Инструкция `COPY` добавляет новый слой с кодом вашего приложения.
- Инструкции `WORKDIR`, `RUN` и `ENTRYPOINT` создают дополнительные слои: устанавливают рабочий каталог, восстанавливают зависимости и определяют команду запуска.

Каждая инструкция добавляет слой поверх предыдущего, формируя иерархию, которая завершается созданием итогового образа. Когда вы запускаете контейнер, Docker объединяет все слои в общую целостную файловую систему.

Такое устройство позволяет Docker эффективно хранить и передавать образы: при изменениях передаются только обновленные слои, что значительно сокращает объем данных.

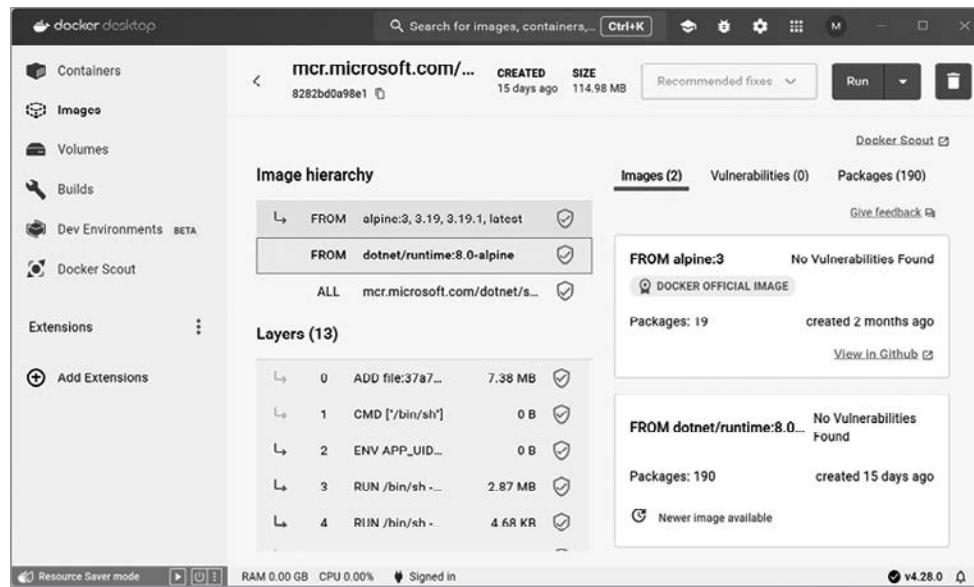
Теперь, когда у нас есть образ Docker для работы, давайте подробнее разберемся, как создаются образы и из чего они состоят.

1. В Docker Desktop откройте представление `Images` (Образы) и выберите образ `ASP.NET Core samples`.
2. В разделе `Image hierarchy` (Иерархия образов) выберите первый базовый образ — `alpine:3`. Это официальный образ Docker, содержащий 19 пакетов. Он включает первые два из 13 слоев, пронумерованных 0 и 1 (рис. 15.8).



**Рис. 15.8.** Иерархия образов и слои для `dotnet/samples:aspnetapp`

3. В том же разделе выберите второй образ — `dotnet/runtime:8.0-alpine`. Этот образ содержит 190 пакетов и состоит из слоев со 2-го по 6-й (рис. 15.9).



**Рис. 15.9.** Второй образ и его слои для `dotnet/samples:aspnetapp`

4. Выберите слой 2, перейдите на вкладку **Command** (Команда) и посмотрите инструкции **ENV**, где установлены переменные среды:

```
ENV APP_UID=1654 ASPNETCORE_HTTP_PORTS=8080 DOTNET_RUNNING_IN_CONTAINER=true DOTNET_SYSTEM_GLOBALIZATION_INVARIANT=true
```

5. Выберите слой 6, чтобы увидеть инструкцию **RUN**, которая загружает среду выполнения .NET для Linux:

```
RUN /bin/sh -c wget -O dotnet.tar.gz https://dotnetcli.azureedge.net/dotnet/Runtime/$DOTNET_VERSION/dotnetruntime-$DOTNET_VERSION-linux-musl-x64.tar.gz && dotnet_sha512='bbed0cf924d103e15d07e069522fc89d921e8d91adccbd4e161345b52fc8bdb26837a18c83d06cccd092d14d3df0e6acbe3b8d348e0825822807a1cbc1c8f549f' && echo "$dotnet_sha512 dotnet.tar.gz" | sha512sum -c - && mkdir -p /usr/share/dotnet && rm dotnet.tar.gz && ln -s /usr/share/dotnet/dotnet /usr/bin/dotnet # используется движок сборки BuildKit
```

6. Затем в разделе **Image hierarchy** выберите третий и последний образ — `mcr.microsoft.com/dotnet/samples:aspnetapp`. Он содержит 331 пакет и состоит из слоев с 7-го по 12-й.

7. Выберите слой 7, чтобы увидеть инструкцию ENV, устанавливающую переменную среды для версии ASP.NET Core:

```
ENV ASPNET_VERSION=8.0.3
```

8. Выберите слой 8, чтобы увидеть инструкцию RUN, которая загружает среду выполнения ASP.NET Core для Linux:

```
RUN /bin/sh -c wget -O aspnetcore.tar.gz https://dotnetcli.azureedge.net/dotnet/aspnetcore/Runtime/$ASPNET_VERSION/aspnetcore-runtime-$ASPNET_VERSION-linux-musl-x64.tar.gz && aspnetcore_sha512='438ed9f5fef9cc63bae18f52af4209a80d8265ef6f9c7b92661e5276538b76163a79e6c59f5fe3d40133e8cdbed7ba50135ce365194358f4abe9df9231a124a5'&& echo "$aspnetcore_sha512 aspnetcore.tar.gz" | sha512sum -c - && tar -oxzf aspnetcore.tar.gz -C /usr/share/dotnet ./shared/Microsoft.AspNetCore.App&& rm aspnetcore.tar.gz # используется движок сборки BuildKit
```

9. Выберите слой 9, чтобы увидеть инструкцию EXPOSE, которая сообщает контейнеру об открытии внутреннего порта 8080:

```
EXPOSE map[8080/tcp:{}]
```

С помощью Docker разработчики .NET могут создавать более портативные, масштабируемые и удобные в управлении приложения. Благодаря контейнерам процесс разработки, развертывания и поддержки приложений становится проще и надежнее.

Теперь посмотрим, как создать новый проект .NET с нуля и контейнеризировать его.

## Контейнеризация проектов .NET

В этой главе мы познакомились с основами контейнеризации и инструментами Docker. Теперь давайте посмотрим, как применить эти знания на практике при разработке приложений .NET.

В этом разделе мы рассмотрим контейнеризацию двух типов проектов:

- консольного приложения;
- сайта ASP.NET Core.

### Контейнеризация проекта консольного приложения

Приступим.

1. В редакторе кода создайте проект Class Library (Библиотека классов) / `classlib`, присвойте ему имя `EnvironmentLib` и добавьте в решение `Chapter15`.
2. Переименуйте файл `Class1.cs` в `EnvironmentInfo.cs`.

3. В файле `EnvironmentInfo.cs` удалите содержимое и добавьте код для сбора сведений о текущей среде:

```
using System.Net; // Подключение класса Dns
using System.Runtime.InteropServices; // Подключение класса RuntimeInformation

namespace EnvironmentLib;

public class EnvironmentInfo
{
    public string UserName { get; } = Environment.UserName;
    public string HostName { get; } = Dns.GetHostName();
    public string DotNet { get; } = RuntimeInformation.FrameworkDescription;
    public string OS { get; } = RuntimeInformation.OSDescription;
    public string Architecture { get; } =
        RuntimeInformation.OSArchitecture.ToString();
    public int Processors { get; } = Environment.ProcessorCount;
    public bool InContainer { get; } = Environment.GetEnvironmentVariable
        ("DOTNET_RUNNING_IN_CONTAINER") is not null;
}
```

4. Выполните сборку проекта `EnvironmentLib`.
5. В редакторе кода создайте проект `Console App` (Консольное приложение) / `console`, присвойте ему имя `AboutMyEnvironment` и добавьте в решение `Chapter15`.
6. По желанию сделайте проект `AboutMyEnvironment` стартовым проектом:
- на панели `Solution Explorer` (Обозреватель решений) найдите проект `AboutMyEnvironment`;
  - щелкните на проекте правой кнопкой мыши;
  - в контекстном меню выберите команду `Set as Startup Project` (Установить как стартовый проект).
7. В файле `AboutMyEnvironment.csproj`:
- статически и глобально импортируйте класс `Console`;
  - добавьте ссылку на пакет `Spectre.Console`;
  - добавьте ссылку на проект `EnvironmentLib`.
- ```
<ItemGroup>
    <Using Include="System.Console" Static="true" />
</ItemGroup>

<ItemGroup>
    <PackageReference Include="Spectre.Console" Version="0.49.1" />
</ItemGroup>

<ItemGroup>
    <ProjectReference Include=".\\EnvironmentLib\\EnvironmentLib.csproj" />
</ItemGroup>
```

8. В файле `Program.cs` удалите все содержимое и напишите код для вывода свойств экземпляра класса `EnvironmentInfo` в таблицу с помощью `Spectre.Console`:

```
using Spectre.Console; // Подключение виджета Table

EnvironmentLib.EnvironmentInfo info = new();
Table t = new();
t.AddColumn("Property");
t.AddColumn("Value");

t.AddRow("User", info.UserName);
t.AddRow("Host", info.HostName);
t.AddRow("OS", info.OS);
t.AddRow("Architecture", info.Architecture);
t.AddRow("Platform", info.DotNet);
t.AddRow("Processors", info.Processors.ToString());
t.AddRow("In a container", info.InContainer.ToString());

AnsiConsole.Write(t);

WriteLine("I will output the time every five seconds.");
WriteLine("Press Ctrl + C to stop.");

while (true)
{
    await Task.Delay(TimeSpan.FromSeconds(5));
    WriteLine(DateTime.Now.ToString("HH:mm:ss"));
}
```

9. Запустите проект `AboutMyEnvironment` и изучите результат, который будет выглядеть примерно так:

Property	Value
User	markj
Host	DESKTOP-J1PQHR7
OS	Microsoft Windows 10.0.22631
Architecture	X64
Platform	.NET 8.0.3
Processors	8
In a container	False

I will output the time every five seconds.

Press Ctrl + C to stop.

16:04:59

16:05:04

16:05:09

10. Нажмите сочетание клавиш `Ctrl+C`, чтобы остановить выполнение консольного приложения.

## Публикация проекта в контейнере Docker

Теперь развернем этот проект в контейнере.

1. В терминале откройте папку **Chapter15** и опубликуйте оба проекта в конфигурации **Release** (Релиз):

```
dotnet publish -c Release
```

2. Результаты публикации показаны ниже:

```
MSBuild version 17.10.0-01+07fd5d51f for .NET
Restore complete (1.0s)
  EnvironmentLib succeeded (0.2s) → EnvironmentLib\bin\Release\net8.0\publish\
  AboutMyEnvironment succeeded (2.5s) → AboutMyEnvironment\bin\Release\net8.0\
    publish\
```

3. В папке **Chapter15** создайте файл с именем **Dockerfile**.
4. В этом файле добавьте инструкции для создания образа: сначала для сборки проектов с помощью .NET SDK, затем для запуска консольного приложения **AboutMyEnvironment**:

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build-env
WORKDIR /Chapter15

# Копирование всех данных
COPY . .
# Восстановление в виде отдельных слоев
RUN dotnet restore
# Создание и публикация релиза
RUN dotnet publish -c Release -o out

# Создание образа среды выполнения
FROM mcr.microsoft.com/dotnet/runtime:8.0
WORKDIR /Chapter15
COPY --from=build-env /Chapter15/out .
ENTRYPOINT ["dotnet", "AboutMyEnvironment.dll"]
```



Инструкция **ENTRYPOINT** задает **dotnet** в качестве хоста для запуска сборки. Вместо этого можно использовать сам исполняемый файл приложения:

```
ENTRYPOINT ["./AboutMyEnvironment"]
```

При таком подходе приложение выполняется напрямую и не зависит от хоста приложений или базовой операционной системы.

5. В терминале создайте контейнер в папке **Chapter15**:

```
docker build -t aboutmyenvironment-image -f Dockerfile .
```



Имена образов репозиториев должны быть записаны строчными буквами.

6. Фрагмент вывода с результатами операции показан ниже:

```
[+] Building 9.0s (14/14) FINISHED
docker:default
=> [internal] load build definition from Dockerfile          0.0s
=> => transferring dockerfile: 442B                          0.0s
=> [internal] load metadata for mcr.../runtime:8.0           0.4s
...
=> [build-env 2/5] WORKDIR /Chapter15                         0.1s
=> [build-env 3/5] COPY . ./                                0.1s
=> [build-env 4/5] RUN dotnet restore                         3.1s
=> [build-env 5/5] RUN dotnet publish -c Release -o out      5.0s
=> [stage-1 3/3] COPY --from=build-env /Chapter15/out .     0.0s
=> exporting to image 0.0s

...
View build details: docker-desktop://dashboard/build/default/default/
lms97oh3e670otwuv89jexa21
List the Docker images, as shown in the following command:
docker images
Note the results, as shown in the following output:
REPOSITORY                      TAG      IMAGE ID      SIZE
aboutmyenvironment-image        latest   8e046d9d8358  194MB
mcr.microsoft.com/dotnet/samples  aspnetapp  8282bd0a98e1  115MB
```



Образ будет отображаться в Docker Desktop.

7. Создайте контейнер с именем `ame` из образа `aboutmyenvironment-image`:

```
docker create --name ame aboutmyenvironment-image
```

8. Система вернет значение GUID, например `1124bbf17c58`.

9. Выведите список всех контейнеров:

```
docker ps -a
```

10. В списке контейнеров появится пункт `ame`:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
1124bbf17c58	aboutmyenvironment-image	"dotnet AboutMyEnvir..."	25 seconds ago
ago	Created	ame	



Контейнер будет отображаться в Docker Desktop.

11. Запустите контейнер `ame`, подключив его стандартный ввод и вывод к терминалу:

```
docker start ame -ai
```



Ключи `-ai` — это сокращение от `--attach` и `--interactive`. Они подключают вывод `STDOUT/STDERR`, передают сигналы и присоединяют входной поток `STDIN` контейнера. Благодаря этому в терминале отображается вывод консольного приложения и принимается ввод с клавиатуры.

12. Будет запущено приложение, выводящее информацию о среде:

Property	Value
User	root
Host	1124bbf17c58
OS	Debian GNU/Linux 12 (bookworm)
Architecture	X64
Platform	.NET 8.0.3
Processors	8
In a container	True

I will output the time every five seconds.

Press Ctrl + C to stop.

16:51:15

16:51:20

13. В терминале нажмите сочетание клавиш `Ctrl+C`, чтобы остановить приложение и контейнер.
14. Запустите контейнер без указания имени. Будет создан контейнер со случайным именем, например `agitated-mahavira` (рис. 15.10).

```
docker run -it --rm aboutmyenvironment-image
```

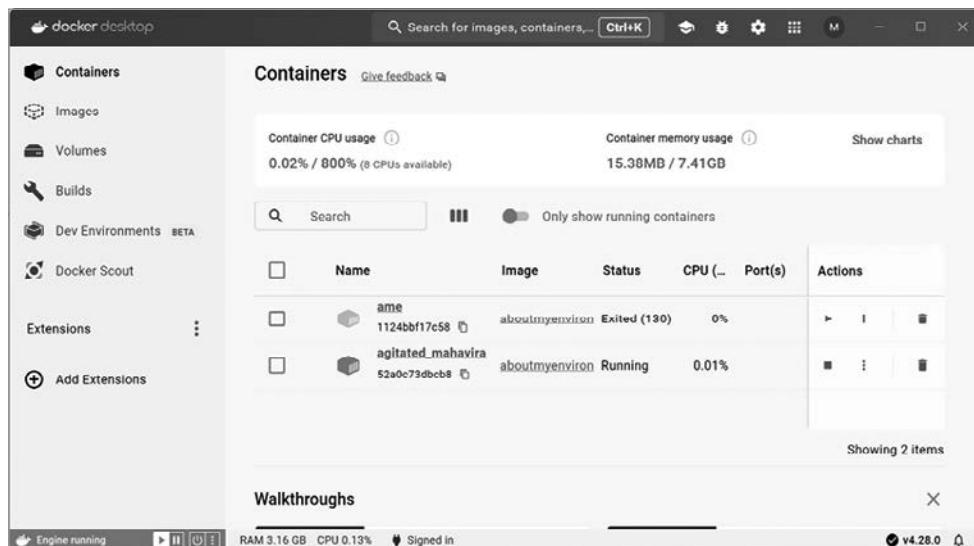
15. В терминале нажмите сочетание клавиш `Ctrl+C`, чтобы остановить консольное приложение. После остановки приложения контейнер будет автоматически удален благодаря ключу `--rm`.
16. В терминале запустите контейнер (без консольного приложения) и подключитесь к нему, переопределив точку входа:

```
docker run -it --rm --entrypoint "bash" aboutmyenvironment-image
```

17. В терминале отобразится имя пользователя и текущий каталог:

```
root@f01f9d2cda94:/Chapter15#
```

18. Выведите список содержимого каталога командой `ls` или `dir`. Результат будет следующим:



**Рис. 15.10.** Контейнер со случайно сгенерированным именем

AboutMyEnvironment	AboutMyEnvironment.pdb
EnvironmentLib.dll	
AboutMyEnvironment.deps.json	AboutMyEnvironment.runtimeconfig.json
EnvironmentLib.pdb	
AboutMyEnvironment.dll	EnvironmentLib.deps.json
Spectre.Console.dll	

19. Запустите консольное приложение командой `dotnet`:

```
dotnet AboutMyEnvironment.dll
```

20. По прошествии не менее десяти секунд нажмите сочетание клавиш `Ctrl+C`, чтобы остановить приложение.
21. Запустите консольное приложение через его исполняемый файл:

```
./AboutMyEnvironment
```

22. По прошествии не менее десяти секунд нажмите сочетание клавиш `Ctrl+C`, чтобы остановить консольное приложение.
23. Перейдите в родительский каталог с помощью команды `cd`. Затем выведите список содержимого командой `ls` или `dir`:

```
Chapter15 bin boot dev etc home lib lib64 media mnt opt proc
root run sbin srv sys tmp usr var
```

24. Завершите работу в оболочке Bash. Контейнер будет автоматически удален:

```
exit
```

Теперь рассмотрим более сложный пример.

## Контейнеризация проекта ASP.NET Core

Публикация простого консольного приложения в Docker — полезный первый шаг для освоения работы с контейнерами в .NET. Однако на практике чаще всего контейнеризируются проекты ASP.NET Core, поэтому рассмотрим их далее.

1. Откройте решение `Chapter15` в редакторе кода.
2. Добавьте проект сайта MVC с учетными записями проверки подлинности, хранящимися в базе данных. Используйте следующие настройки:
  - выберите шаблон C#-проекта `ASP.NET Core Web App (Model-View-Controller)` (Веб-приложение ASP.NET Core (модель-представление-контроллер)) / `mvc`;
  - в поле `Project name` (Имя проекта) укажите значение `Northwind.Mvc`;
  - в поле `Solution name` (Имя решения) укажите значение `Chapter15`;
  - в раскрывающемся списке `Authentication type` (Тип проверки подлинности) выберите пункт `Individual Accounts` (Индивидуальные учетные записи) / `--auth Individual`;
  - установите флажок `Configure for HTTPS` (Настроить для HTTPS);
  - установите флажок `Enable container support` (Включить поддержку контейнера);
  - в раскрывающемся списке `Container OS` (ОС контейнера) выберите пункт `Linux`;
  - в раскрывающемся списке `Container build type` (Тип сборки контейнера) выберите пункт `Dockerfile`;
  - снимите флажок `Do not use top-level statements` (Не использовать операторы верхнего уровня).



Если вы работаете в редакторе Code, перейдите в папку `Chapter15` и выполните команды:

```
dotnet new mvc --auth Individual -o Northwind.Mvc
dotnet sln add Northwind.Mvc
```

В JetBrains Rider щелкните правой кнопкой мыши на решении `Chapter15`, выберите в контекстном меню команду `Add > New Project` (Добавить > Новый проект). В диалоговом окне `New Project` (Новый проект) выберите шаблон `ASP.NET Core Web Application` (Веб-приложение ASP.NET Core), в раскрывающемся списке `Type` (Тип) выберите значение `Web App (Model-View-Controller)` (Веб-приложение (модель-представление-контроллер)), в раскрывающемся списке `Auth` (Проверка подлинности) выберите пункт `Individual authentication` (Индивидуальная проверка подлинности), затем нажмите кнопку `Create` (Создать).

3. Выберите команду меню View ▶ Output (Вид ▶ Вывод).
4. На панели Output (Вывод) в раскрывающемся списке Show output from: (Показать вывод из:) выберите вариант Container Tools (Инструменты контейнера). Visual Studio проверит наличие установленного Docker Desktop, загрузит необходимые образы и создаст образ Docker с именем northwindmvc для проекта Northwind.Mvc с тегом dev.



Вывод в книге не показан, поскольку он занял бы несколько страниц, да и не содержит особенно интересных деталей. Рекомендуется просмотреть вывод на своем компьютере: в нем отображается процесс сборки образа на основе Dockerfile, содержимое которого рассмотрено в предыдущем разделе.

5. В файле Northwind.Mvc.csproj проекта добавьте ссылку на проект библиотеки классов EnvironmentLib:

```
<ItemGroup>
  <ProjectReference Include="..\EnvironmentLib\EnvironmentLib.csproj" />
</ItemGroup>
```

6. Выполните сборку проекта Northwind.Mvc.
7. Откройте файл HomeController.cs в папке Controllers проекта Northwind.Mvc. В методе Index добавьте выражения для получения информации о среде и передачи ее представлению в качестве модели:

```
public IActionResult Index()
{
    EnvironmentLib.EnvironmentInfo model = new();
    return View(model);
}
```

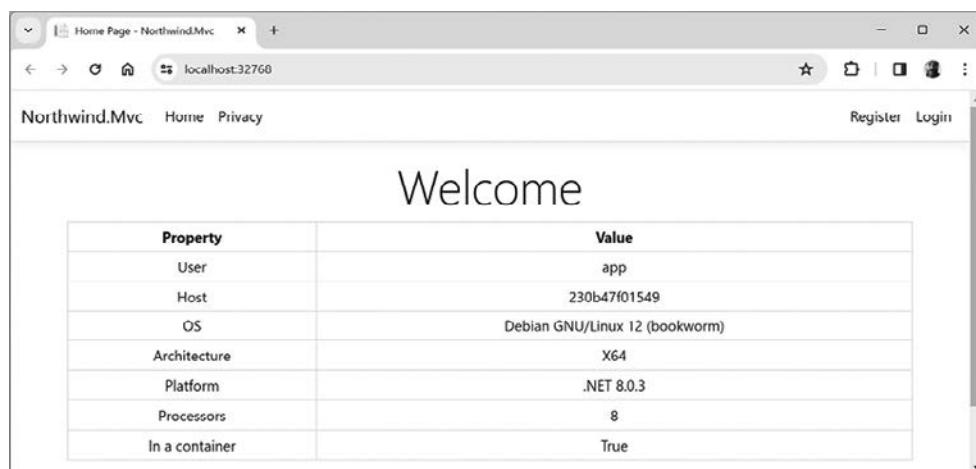
8. Откройте файл Index.cshtml в папке Views\Home проекта Northwind.Mvc. Замените абзац со ссылкой на документацию на разметку для отображения информации о среде:

```
@model EnvironmentLib.EnvironmentInfo
 @{
    ViewData["Title"] = "Home Page";
}


<h1 class="display-4">Welcome</h1>
    <table class="table table-bordered table-sm">
        <tr><th>Property</th><th>Value</th></tr>
        <tr><td>User</td><td>@Model.UserName</td></tr>
        <tr><td>Host</td><td>@Model.HostName</td></tr>
        <tr><td>OS</td><td>@Model.OS</td></tr>
        <tr><td>Architecture</td><td>@Model.Architecture</td></tr>
        <tr><td>Platform</td><td>@Model.DotNet</td></tr>
        <tr><td>Processors</td><td>@Model.Processors</td></tr>
        <tr><td>In a container</td><td>@Model.InContainer</td></tr>
    </table>
</div>

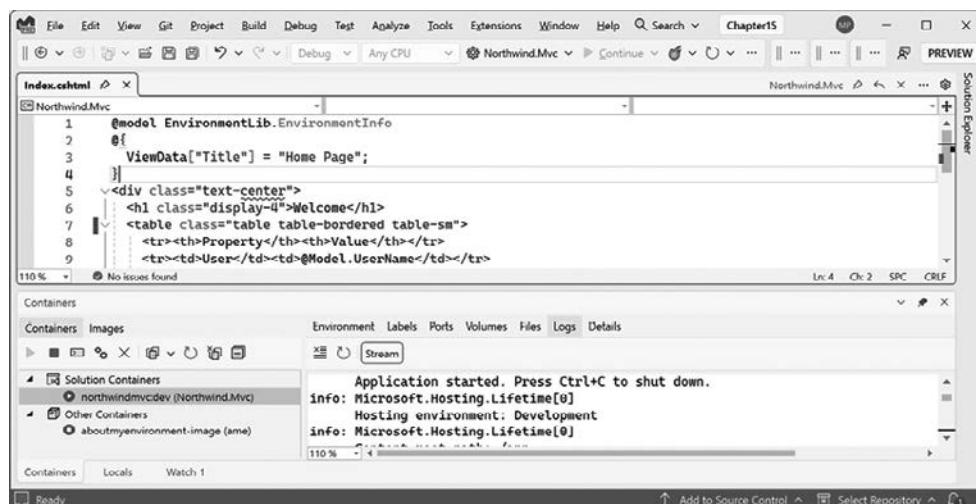

```

9. В Visual Studio выберите команду меню Debug ▶ Start Debugging (Отладка ▶ Начать отладку), нажмите клавишу F5 или кнопку Container (Dockerfile) (Контейнер (Dockerfile)) на панели инструментов. В браузере откроется домашняя страница сайта Northwind.Mvc (рис. 15.11).



**Рис. 15.11.** Сайт Northwind.Mvc, размещенный в Docker

10. В Visual Studio откройте панель Containers (Контейнеры). В списке будет отображаться запущенный контейнер northwindmvc:dev, а на вкладке Logs (Журналы) будет доступен его вывод, как показано на рис. 15.12.



**Рис. 15.12.** Окно контейнеров в Visual Studio

Проект можно также опубликовать без использования Dockerfile:

```
dotnet publish $PROJECT_PATH --os linux --arch arm64 /t:PublishContainer  
/p:ContainerImageTags=$CI_JOB_DATE -c Release
```

На этом этапе рассмотрены практические примеры публикации консольного приложения и проекта ASP.NET Core в контейнеры Docker, а также их выполнения.

Теперь перейдем к тестовым контейнерам, которые помогают улучшить процесс тестирования.

## Работа с тестовыми контейнерами

Testcontainers – это библиотека для интеграционного тестирования, которая использует контейнеры Docker для имитации внешних сервисов и систем. Изначально она была разработана для Java, но благодаря мощной концепции и гибкости появилась поддержка и для других языков программирования, включая .NET. Testcontainers для .NET – это управляемая реализация, позволяющая создавать легкие, одноразовые экземпляры баз данных, браузеров или других сервисов, которые могут быть запущены в контейнере Docker, прямо из тестового кода на .NET.

### Принцип работы Testcontainers для .NET

Основная концепция Testcontainers – использовать контейнеры Docker для внешних зависимостей, которые нужны интеграционным тестам. Это могут быть:

- базы данных, например SQL Server, PostgreSQL, MongoDB;
- кэши, например Redis;
- системы обмена сообщениями, например RabbitMQ;
- браузеры для Playwright и других автоматизированных веб-тестов;
- любой другой сервис, который можно запустить в контейнере.

Благодаря этому тесты проходят в среде, очень похожей на реальную. Так вы можете получать более надежные результаты.

Каждый тест запускает отдельный экземпляр нужного сервиса. Тесты изолированы друг от друга и не мешают работе соседних тестов. Все зависимости можно описать прямо в коде тестов, чтобы сразу видеть, что именно требуется для работы.

Поскольку сервисы запускаются в контейнерах, тесты можно переносить на любое устройство, где установлен Docker. Testcontainers поддерживает много разных сервисов, поэтому библиотека подходит для самых разных сценариев.

## Пример использования

Рассмотрим простой пример применения Testcontainers в проекте .NET для тестирования базы данных SQL Server.

Сначала добавьте ссылки на нужные пакеты Testcontainers:

```
<PackageReference Include="Testcontainers" Version="3.8.0" />
<PackageReference Include="Testcontainers.MsSql" Version="3.8.0" />
```

Затем напишите интеграционные тесты:

```
using Testcontainers.MsSql; // Подключение класса MsSqlContainer

namespace NorthwindTests;

public class DatabaseIntegrationTests
{
    private MsSqlContainer _sqlContainer;

    private async Task SetUp()
    {
        _sqlContainer = new(new(
            database: "Northwind",
            username: "sa",
            password: "s3cret-n1nj@"
        ));

        await _sqlContainer.StartAsync();
    }

    [Fact]
    public async Task TestDatabaseConnection()
    {
        await SetUp();

        // Использование свойства _sqlContainer.ConnectionString
        // для подключения к SQL Server

        await TearDown();
    }

    private async Task TearDown()
    {
        await _sqlContainer.StopAsync();
    }
}
```

В этом примере класс `MsSqlContainer` из библиотеки `Testcontainers` управляет контейнером SQL Server. Перед каждым тестом он запускает новый экземпляр базы данных с нужной конфигурацией. Тесты подключаются к этой базе и работают в известной изолированной среде. После выполнения тестов контейнер останав-

ливается и удаляется. Это исключает сохранение состояния между запусками и повышает надежность тестирования.

Testcontainers для .NET — отличное решение для разработчиков, которые хотят улучшить свои стратегии интеграционного тестирования с реальными зависимостями, не тратя время на их настройку вручную.



**В копилку знаний:** подробнее о тестовых контейнерах можно узнать на сайте [dotnet.testcontainers.org](http://dotnet.testcontainers.org).

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 15.1. Дополнительные онлайн-материалы

Электронная книга о контейнеризации в .NET доступна на сайте [learn.microsoft.com/ru-ru/dotnet/architecture/microservices](https://learn.microsoft.com/ru-ru/dotnet/architecture/microservices).

### Упражнение 15.2. Тренировочные упражнения

Интерфейс Docker Desktop подробно описан на странице [docs.docker.com/desktop/use-desktop/](https://docs.docker.com/desktop/use-desktop/).

### Упражнение 15.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Почему контейнеры считаются более «легкими» по сравнению с виртуальными машинами?
2. Как связаны Docker и Kubernetes?
3. В чем разница между реестром Docker, образом и контейнером?
4. Чем отличаются команды `docker start` и `docker run`?
5. Как задать базовый образ в файле `Dockerfile`?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 15.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-15---container-ization-using-docker](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-15---container-ization-using-docker).

## Резюме

В этой главе мы разобрали:

- концепции, связанные с контейнеризацией;
- основные инструменты и технологии для работы с Docker;
- приемы управления контейнерами с помощью Docker;
- способы контейнеризации консольных приложений и проектов ASP.NET Core;
- принципы работы с тестовыми контейнерами.

В следующей главе вы познакомитесь со стеком .NET Aspire. В нем реализованы похожие возможности, что и в Testcontainers, но значительно более широкие. Вы узнаете, как .NET Aspire встроен в .NET 8 и более поздние версии, и о поддержке со стороны Microsoft.

# 16 Облачная разработка с помощью .NET Aspire

В этой главе мы познакомимся с разработкой облачных решений с помощью .NET Aspire — новой функциональности в .NET 8 и более поздних версиях. Она помогает сделать локальную разработку распределенных облачных решений удобнее и эффективнее.

О релизе .NET Aspire было объявлено 21 мая 2024 года на конференции Microsoft Build. Реакция разработчиков на эту новинку оказалась очень теплой. Например, Pramesh KC — обычный разработчик .NET, такой же как и вы, — назвал .NET Aspire настоящим переломным моментом в развитии .NET 8. Он отметил, что Aspire быстро стал его любимым инструментом. Полный текст его статьи можно прочитать на сайте [dev.to/prameshkc/discovering-the-gem-in-net-8-introducing-net-aspire-1mlo](https://dev.to/prameshkc/discovering-the-gem-in-net-8-introducing-net-aspire-1mlo).

Другой разработчик, Creyke, поделился своими впечатлениями в канале Aspire на Discord-сервере Microsoft DotNetEvolution. Вот что он написал:

*«Из-за Aspire пропустил вечеринку, но пользуюсь уже два дня и хочу сказать — это действительно впечатляет. Абстракции очень продуманные, писать собственные расширения — настоящее удовольствие, и все работает с первого раза. Отличные настройки по умолчанию! Если Aspire будет правильно продвигаться на рынке, получит хорошие инструменты и поддержку сообщества, это станет настоящей гордостью для всех, кто его создавал».*

## Знакомство с Aspire

Что такое .NET Aspire? В анонсе на сайте Microsoft Dev Blogs ([devblogs.microsoft.com/dotnet/introducing-dotnet-aspire-simplifying-cloud-native-development-with-dotnet-8/](https://devblogs.microsoft.com/dotnet/introducing-dotnet-aspire-simplifying-cloud-native-development-with-dotnet-8/)) написано:

*«.NET Aspire — это определенный стек для создания устойчивых, наблюдаемых и настраиваемых облачных приложений на базе .NET».*

Каждое слово в этом описании выбрано не случайно, и стоит остановиться на ключевых идеях.

- **Определенный стек.** Одной из самых сложных задач при создании современных распределенных решений остается слишком большой выбор. Для каждой

функции можно найти десятки компонентов, а для каждого компонента — несколько способов настройки. Aspire предлагает собственное мнение о том, какие компоненты стоит использовать и как их лучше настраивать. Подробно об этом рассказывается в разделе «Компоненты Aspire» дальше в главе. Если вам подходит такой метод, работать с Aspire будет особенно удобно. Если нет — всегда можно отменить настройки по умолчанию и использовать свои.

- **Устойчивость.** Aspire разработан с опорой на лучшие практики построения облачных решений, чтобы обеспечить отказоустойчивость. В частности, поддерживаются такие подходы, как кэширование, организация очередей сообщений и использование библиотеки Polly для реализации механизмов автоматического восстановления при сбоях.



О том, как использовать Polly для повышения отказоустойчивости приложений и сервисов, вы найдете информацию в моей книге «.NET 8: приложения и сервисы». Документация по Polly доступна на сайте [www.pollydocs.org](http://www.pollydocs.org). Если вы работаете с Aspire, вам не нужно отдельно настраивать Polly — этот компонент уже входит в состав Aspire и предварительно настроен.



В .NET Aspire 8.2 произошли изменения, нарушающие совместимость. Чтобы использовать эту версию, убедитесь, что:

- установлена последняя версия рабочей нагрузки (workload);
- в файл проекта AppHost добавлена ссылка на пакет Aspire.Hosting.AppHost версии 8.2.0 или новее:

```
<PackageReference Include="Aspire.Hosting.AppHost" Version="8.2.0" />
```

Иначе при сборке может появиться ошибка, указывающая на отсутствие нужной ссылки в проекте AppHost.

Кроме того, начиная с версии 8.2 в Aspire были переименованы сущности: Components теперь называются Integrations. Подробнее об изменениях — на сайте Microsoft: <https://devblogs.microsoft.com/dotnet/announcing-dotnet-aspire-8-2/>.

- **Наблюдаемость.** Aspire помогает реализовать трассировку, журналирование и сбор метрик с помощью OpenTelemetry — ведущего фреймворка телеметрии для .NET и других платформ. Aspire также включает информационную панель для мониторинга активности на всех уровнях решения, даже в самых сложных сценариях.
- **Настраиваемость.** Aspire предлагает разумные настройки по умолчанию для всех компонентов. При необходимости их можно изменить. Если ваши требования существенно отличаются от стандартных, возможно, вы не получите от Aspire столько пользы, как те разработчики, которые используют рекомендованные компоненты с наиболее популярной конфигурацией. Но гибкость настройки все равно остается.

- **Работа в облаке.** Проекты Aspire изначально ориентированы на развертывание в облаке в контейнерах. Для локальной разработки они используют Docker Desktop или Podman. Про Docker мы подробно говорили в главе 15. Чтобы узнать больше о Podman, можно посетить сайт [podman.io](https://podman.io).

Aspire изначально включает такие возможности, как обнаружение сервисов, телеметрия, отказоустойчивость и проверка состояния. Он упрощает работу локальных разработчиков, позволяя быстро находить, получать и настраивать необходимые зависимости.



Предварительные версии Aspire впервые стали доступны вместе с выпуском .NET 8 в ноябре 2023 года. Версия Aspire 1.0 была официально представлена на конференции Microsoft Build 21 мая 2024 года. Она получила номер версии 8.0, поскольку Aspire входит в состав .NET 8. Следующая версия Aspire была выпущена вместе с .NET 9 в ноябре 2024 года и получила номер версии 9.0. В дальнейшем Aspire будет развиваться параллельно с основными релизами .NET. Ожидается, что следующая версия Aspire появится в ноябре 2025 года вместе с .NET 10, который станет выпуском с долгосрочной поддержкой (LTS).

## Что говорит команда Aspire

Чтобы лучше понять цели Aspire, полезно обратиться к комментариям самой команды разработчиков. Приведу несколько цитат, прозвучавших на публичной встрече, доступной для просмотра по адресу [www.youtube.com/watch?v=OsF7\\_ZxRlvw](https://www.youtube.com/watch?v=OsF7_ZxRlvw).

### Дэвид Фаулер:

*«С помощью проектов Aspire можно смоделировать и воспроизвести инфраструктуру приложения или его части прямо на локальном компьютере. Aspire запускает исполняемые файлы и контейнеры и следит за тем, чтобы все работало вместе без лишних усилий».*

### Дэмиен Эдвардс:

*«Когда Aspire начинает действительно помогать? Как только в приложении появляется больше одного узла. Например, веб-приложение, которое общается с базой данных. Или веб-приложение, которое делает запросы к API. Или любое приложение, где несколько компонентов работают вместе. Как только нужно развернуть больше одного проекта – Aspire становится очень полезным».*

### Джон Гэллоуэй:

*«Когда я начал работать с Aspire, сразу заметил: даже в простых случаях – когда приложение подключается к API или базе данных – наличие информационной панели и быстрый доступ ко всем адресам узлов сильно упростили жизнь. Больше не нужно метаться между окнами терминала и гадать, что запущено, а что нет, – все видно сразу».*

Лично для меня важно, что каждый разработчик .NET почувствует преимущества Aspire, как только начнет работать над решением с несколькими проектами, базами данных или сервисами вроде Redis. У Microsoft богатый опыт в корпоративном секторе именно потому, что компания понимает потребности разработчиков и создает действительно полезные инструменты, облегчающие повседневную работу.

## Редактор кода и поддержка CLI для Aspire

Поддержка Aspire встроена в Visual Studio 2022 начиная с версии 17.10. Пользователи других редакторов, в том числе Visual Studio Code, могут использовать инструменты CLI, а со временем ожидается расширение поддержки.



Подробнее о поддержке Aspire в Visual Studio Code можно узнать на странице: [devblogs.microsoft.com/dotnet/may-release-of-csharp-dev-kit/](https://devblogs.microsoft.com/dotnet/may-release-of-csharp-dev-kit/).

Чтобы пользоваться инструментарием Aspire CLI, нужно установить рабочую нагрузку Aspire .NET SDK:

```
dotnet workload install aspire
```

Чтобы использовать актуальную версию инструментария Aspire CLI, обновите все рабочие нагрузки .NET SDK:

```
dotnet workload update
```

Чтобы проверить версию инструментов Aspire CLI, выведите список рабочих нагрузок .NET SDK:

```
dotnet workload list
```



О плагине Aspire для среды Rider можно узнать на странице [blog.jetbrains.com/dotnet/2024/02/19/jetbrains-rider-and-the-net-aspire-plugin/](https://blog.jetbrains.com/dotnet/2024/02/19/jetbrains-rider-and-the-net-aspire-plugin/).

## Запуск решения Aspire

Чтобы запустить решение Aspire, выберите подходящий вариант в зависимости от редактора кода.

- В программе Visual Studio, Visual Studio Code или Rider нажмите клавишу F5, чтобы подключить отладчик ко всем проектам. При необходимости можно использовать горячую перезагрузку для внесения изменений в код во время работы.
- В Visual Studio, Visual Studio Code или Rider нажмите сочетание клавиш Ctrl+F5, чтобы запустить решение без отладки. При изменении проектов Visual Studio автоматически перезапустит их.

- В терминале введите команду `dotnet watch`. Команда позволяет следить за изменениями в проектах и перезапускать их при необходимости. Команда разработчиков Aspire продолжает улучшать поддержку горячей перезагрузки.

## Типы проектов Aspire

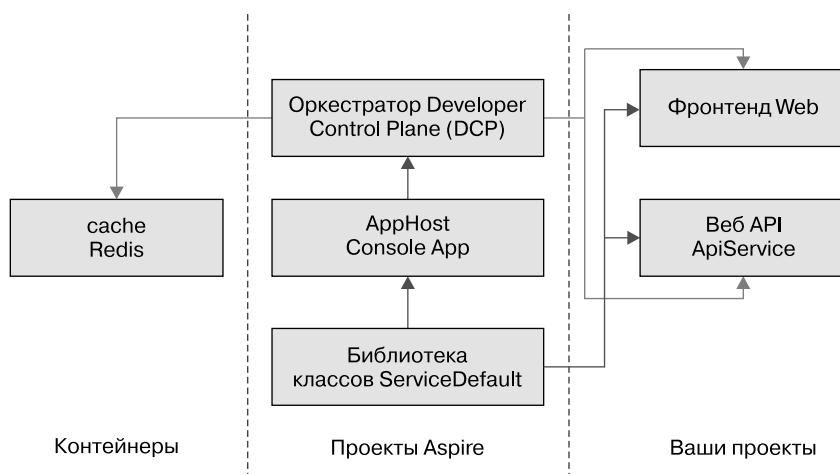
Решение Aspire состоит из двух специальных проектов.

- **AppHost** — консольное приложение, которое запускает все остальные проекты и отвечает за правильную настройку ресурсов и конечных точек.
- **ServiceDefaults** — библиотека классов, где сосредоточена общая конфигурация всех ресурсов Aspire, включая базы данных и проекты .NET.

Проект **AppHost** запускает все проекты .NET, контейнеры и исполняемые файлы, которые нужны для работы облачного распределенного приложения. Если вы используете программу Visual Studio, отладчик автоматически подключится ко всем проектам, что позволяет заходить внутрь каждого сервиса и работать с ним напрямую.

Проект **ServiceDefaults** хранит общую логику для настройки сервисов и компонентов. Здесь задаются параметры для обнаружения сервисов, телеметрии, проверки работоспособности и других важных аспектов. Это удобное централизованное расположение, куда можно обратиться, чтобы изменить или дополнить конфигурацию проекта Aspire.

Все решения Aspire включают два специальных проекта, о которых мы только что говорили, а также любые другие проекты, составляющие ваше распределенное приложение, как показано на рис. 16.1.



**Рис. 16.1.** DCP, проекты Aspire, пользовательские проекты и контейнеры в модели приложения Aspire



Developer Control Plane (DCP) — это оркестратор с закрытым исходным кодом, который используется в Aspire. Команда Aspire предпочитает не акцентировать на нем внимание, поскольку он недоступен для сообщества разработчиков. Все остальные части Aspire спроектированы так, чтобы быть открытыми и расширяемыми.

Проект AppHost считывает конфигурацию из проекта `ServiceDefaults` и с помощью DCP управляет запуском контейнеров и всех ваших проектов. Ваши собственные проекты также должны использовать настройки из `ServiceDefaults`.

## Типы ресурсов Aspire

В Aspire предусмотрены следующие встроенные типы ресурсов.

- **Project** — проект .NET, например веб-сервис или сайт ASP.NET Core.
- **Container** — контейнерный образ, например образ Docker для Redis или RabbitMQ.
- **Executable** — исполняемый файл.

Для контейнеров Aspire поддерживает два варианта: можно запустить контейнер в процессе локальной разработки или подключиться к уже работающему экземпляру с помощью строки подключения.

## Модель и оркестрация приложений Aspire

Оркестрация в Aspire упрощает настройку и управление связями между всеми компонентами облачного решения. Aspire помогает решать несколько ключевых задач.

- **Состав решения.** Проект AppHost определяет, из каких ресурсов состоит приложение — это могут быть проекты .NET, контейнеры, исполняемые файлы и облачные ресурсы.
- **Обнаружение сервисов.** Проект AppHost описывает, как разные ресурсы взаимодействуют между собой.

Оркестрация Aspire во многом напоминает подход ASP.NET Core, где с помощью класса `WebApplication` формируется конфигурация сайта или сервиса. Аналогично в проектах Aspire AppHost используется класс `DistributedApplication`:

```
var builder = DistributedApplication.CreateBuilder(args);
```

Затем этот объект `builder` вызывает методы для сопоставления ресурсов в распределенной модели приложения Aspire, как показано в табл. 16.1.

Например, чтобы добавить контейнер Redis, нужно вызвать метод `AddRedis`:

```
var cache = builder.AddRedis("cache");
```

**Таблица 16.1.** Методы сопоставления ресурсов в распределенной модели приложения Aspire

Метод	Описание
AddContainer	Добавляет ресурс контейнера в модель приложения Aspire. Если указать ссылку на пакет Aspire, например <code>Aspire.Hosting.Redis</code> , можно использовать методы расширения, такие как <code>AddRedis</code> , которые автоматически добавляют контейнер Redis
AddExecutable	Добавляет исполняемый ресурс в модель приложения Aspire
AddProject	Добавляет ресурс проекта .NET в модель приложения Aspire. При этом автоматически считывается конфигурация из файла <code>launchSettings.json</code> , например URL-адрес проекта
AddResource	Добавляет произвольный ресурс в модель приложения Aspire

Чтобы добавить проект веб-API на ASP.NET Core, нужно использовать метод `AddProject`:

```
var apiService = builder.AddProject<Projects.AspireStarter_ApiService>("apiservice");
```



**Дельный совет:** когда вы добавляете ресурс в Aspire, вы задаете ему короткое имя, например `cache` или `apiservice`. Эти имена затем используются в других проектах для динамических ссылок на конечные точки ресурсов, например `http://apiservice`. Это не настоящий URL-адрес — фактический URL и номера портов Aspire подставит автоматически во время обнаружения сервисов.

Помимо методов построения модели приложения Aspire, существуют методы для добавления настроек, как показано в табл. 16.2.

**Таблица 16.2.** Методы добавления настроек для компонентов Aspire

Метод	Описание
AddConnectionString	Добавляет строку подключения, если вы хотите использовать внешние ресурсы, уже настроенные за пределами узла приложений
AddParameter	Считывает значения из раздела <code>Parameters</code> конфигурации хоста приложения и передает их проекту при локальном запуске. При развертывании такие параметры будут запрашиваться отдельно. Подробнее о внешних параметрах можно узнать на сайте <a href="https://learn.microsoft.com/ru-ru/dotnet/aspire/fundamentals/external-parameters">learn.microsoft.com/ru-ru/dotnet/aspire/fundamentals/external-parameters</a>

Чтобы подключить ресурсы в модели приложения Aspire с помощью механизма обнаружения сервисов, нужно вызвать один из методов `With`, как показано в табл. 16.3.

**Таблица 16.3.** Способы подключения и настройки компонентов Aspire

Метод	Описание
<code>WithReference</code>	Связывает ресурсы, чтобы они могли использовать правильную конфигурацию для взаимодействия друг с другом, передавая информацию об обнаружении сервисов для указанных проектов
<code>WithExternalHttpEndpoints</code>	Настраивает веб-проект на внешние конечные точки HTTP, которые будут автоматически настроены для всех связанных компонентов и проектов Aspire
<code>WithHttpEndpoint</code> , <code>WithHttpsEndpoint</code>	Используют код для настройки именованных конечных точек
<code>WithArgs</code>	Добавляет обратный вызов, который будет выполнен с аргументами из командной строки при запуске ресурса контейнера
<code>WithEnvironment</code>	Добавляет переменную среды к ресурсу

Например, чтобы добавить проект веб-фронтенда, который будет вызывать веб-сервис и контейнер Redis, нужно вызвать метод `AddProject`, а затем подключить зависимости, используя метод `WithReference`:

```
builder.AddProject<Projects.AspireStarter_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
    .WithReference(apiService);
```

Все распределенные решения нуждаются во взаимодействии с удаленными сервисами. В Aspire для этого была создана библиотека `Microsoft.Extensions.ServiceDiscovery`. Она упрощает поиск сервисов на стороне клиента и балансировку нагрузки, предоставляя основную абстракцию и несколько вариантов реализации. Библиотека интегрируется с `HttpClientFactory` и прокси YARP (*Yet Another Reverse Proxy*, [microsoft.github.io/reverse-proxy/](https://microsoft.github.io/reverse-proxy/)) для локальной разработки, а также с Kubernetes в развернутых средах.



**В копилку знаний:** подробнее о механизме обнаружения сервисов можно узнать по адресу [learn.microsoft.com/ru-ru/dotnet/aspire/service-discovery/overview](https://learn.microsoft.com/ru-ru/dotnet/aspire/service-discovery/overview).

## Шаблоны проектов Aspire

В Aspire для .NET 8 доступно пять шаблонов проектов.

- **.NET Aspire Starter Application** (Начальное приложение .NET Aspire) / `aspire-starter`. Решение с четырьмя или пятью проектами и дополнительным контейнером Redis, в зависимости от выбранных параметров. Этот шаблон подходит для знакомства с Aspire.
- **.NET Aspire Application** (Пустое приложение .NET Aspire) / `aspire`. Решение с двумя обязательными проектами — AppHost и ServiceDefaults. Используется для создания новых решений с применением Aspire.
- **.NET Aspire App Host** (Узел приложений .NET Aspire) / `aspire-apphost`. Проект с минимальной настройкой для размещения приложений. Применяется, когда нужно добавить Aspire в существующее решение без использования средств Visual Studio.
- **.NET Aspire Service Defaults** (Параметры службы .NET Aspire по умолчанию) / `aspire-servicedefaults`. Проект с минимальной настройкой для хранения общих параметров сервисов. Используется для интеграции Aspire в существующее решение без применения Visual Studio.
- **.NET Aspire Test Project (xUnit)** (Тестовый проект .NET Aspire (xUnit)) / `aspire-xunit`. Проект для добавления интеграционных тестов в решение Aspire.



В Aspire для .NET 9, скорее всего, появятся два дополнительных шаблона проектов для nUnit и MSTest. Подробнее об этом можно узнать на странице [github.com/dotnet/aspire/tree/main/src/Aspire.ProjectTemplates/templates](https://github.com/dotnet/aspire/tree/main/src/Aspire.ProjectTemplates/templates).

Чтобы вывести список шаблонов проектов Aspire в терминале, выполните команду `dotnet new list aspire`.

Помимо шаблонов проектов, в Visual Studio предусмотрена инструментальная поддержка для добавления оркестрации Aspire и двух обязательных проектов в существующее решение. Эту возможность мы рассмотрим позже в этой главе.

Теперь создадим начальный проект на основе шаблона и разберем ключевые концепции и особенности Aspire.

## Изучение начального шаблона Aspire

Шаблон проекта .NET Aspire Starter Application (Начальное приложение .NET Aspire) помогает быстро познакомиться с рабочим решением на базе Aspire. Это отличный способ увидеть, как все части Aspire взаимодействуют друг с другом.

Решение, созданное с помощью шаблона, включает четыре или пять проектов, а также дополнительный контейнер Redis.

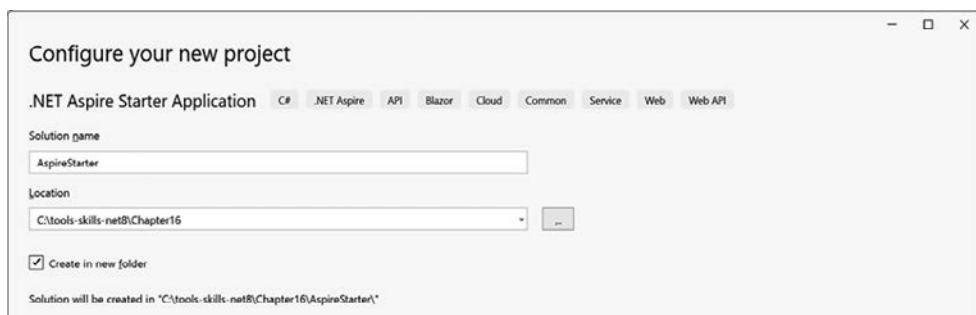
- Проект Blazor Web Application (Веб-приложение Blazor) с именем <Имя\_решения>.Web, который служит внешним пользовательским интерфейсом.
- Проект ASP.NET Core Web API (Веб-API ASP.NET Core) с именем <Имя\_решения>.ApiService, предоставляющий внутренний сервис информации о погоде (на базе минимального API).
- Проект .NET Aspire App Host (Узел приложений .NET Aspire) с именем <Имя\_решения>.AppHost, управляющий запуском всех сервисов.
- Проект .NET Aspire Service Defaults (Параметры службы .NET Aspire по умолчанию) с именем <Имя\_решения>.ServiceDefaults, содержащий общие настройки для всех сервисов.
- При необходимости — проект xUnit Test (Тестовый проект xUnit) с именем <Имя\_решения>.Tests для интеграционного тестирования.
- Контейнер Redis, работающий через Docker (по умолчанию) или через Podman.

Теперь, когда мы разобрались, из чего состоит шаблон начального проекта Aspire, посмотрим, как он работает на практике.

## Создание начального приложения Aspire

Создадим начальное приложение Aspire с помощью шаблона проекта.

1. В редакторе кода создайте проект .NET Aspire Starter Application (Начальное приложение .NET Aspire) / `aspire-starter`, присвойте решению имя `AspireStarter` и разместите во вложенной папке каталога `Chapter16` (рис. 16.2).

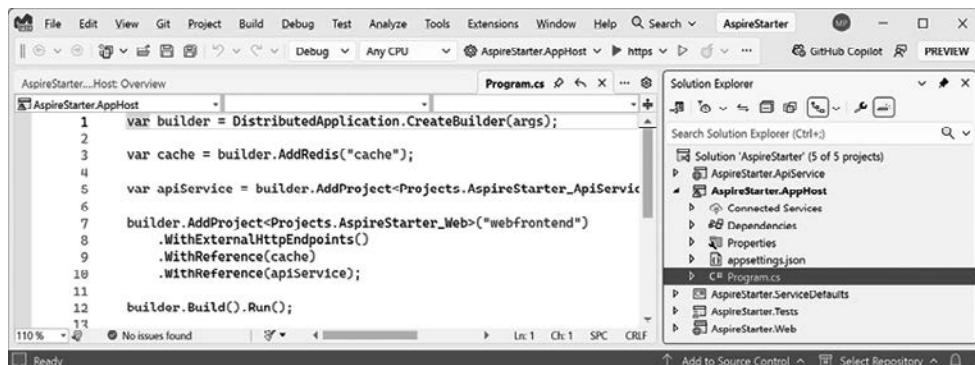


**Рис. 16.2.** Настройка нового проекта для начального шаблона Aspire



**Внимание!** Этот шаблон проекта не позволяет задать имя для отдельных проектов — вместо этого указывается имя решения. Оно задается в качестве префикса для всех проектов внутри решения. В этой главе используется папка `Chapter16`, где будет несколько решений, каждое со своими проектами.

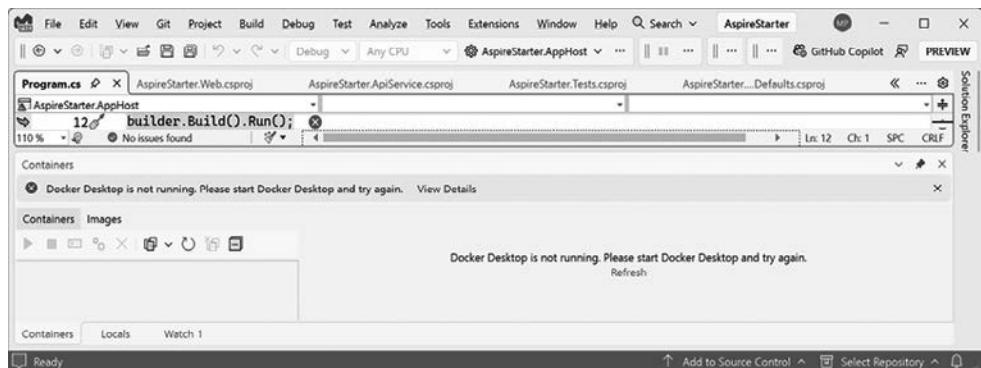
2. Нажмите кнопку **Next** (Далее).
  3. На этапе ввода **Additional information** (Дополнительные сведения) выберите следующие параметры:
    - установите флагок **Configure for HTTPS** (Настроить для HTTPS);
    - установите флагок **Use Redis for caching** (Использовать Redis для кэширования) (будет добавлен контейнер Redis);
    - установите флагок **Create a tests project** (Создать тестовый проект) (будет добавлен проект для интеграционных тестов).
  4. На панели **Solution Explorer** (Обозреватель решений) будет создано пять проектов. Проект **AspireStarter.AppHost** автоматически устанавливается стартовым (выделен жирным шрифтом) (рис. 16.3).



**Рис. 16.3.** Решение начального приложения Aspire с пятью проектами

5. Запустите проект `AspireStarter.AppHost`. Для этого можно выбрать команду меню `Debug ▶ Start Debugging` (Отладка ▶ Начать отладку), нажать кнопку `https` на панели инструментов или клавишу `F5`.
  6. Если приложение Docker Desktop не запущено, появится ошибка (см. рис. 16.4 и вывод в терминале).

```
info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 8.0.0+6596fdc41a8d419876a6bf4abc17b7c66b9ef63a
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is: C:\tools-skills-net8\Chapter16\
AspireStarter\AspireStarter.AppHost
fail: Microsoft.Extensions.Hosting.Internal.Host[11]
      Hosting failed to start
      Aspire.Hosting.DistributedApplicationException: Container runtime
'docker' was found but appears to be unhealthy. The error from the container
runtime check was error during connect: this error may indicate that the
docker daemon is not running:
```



**Рис. 16.4.** Visual Studio выдает ошибку, если не запущен демон Docker



Если приложение Docker Desktop не запущено, появится ошибка (см. рис. 16.4 и вывод в терминале).

7. Остановите отладку в Visual Studio.
8. Запустите приложение Docker Desktop. Вместе с ним будет запущен демон Docker.
9. Снова запустите проект `AspireStarter.AppHost`. Выберите команду меню `Debug > Start Debugging` (Отладка > Начать отладку), нажмите кнопку `https` или клавишу `F5`.
10. Консольное приложение хоста приложений и информационная панель Aspire должны запуститься успешно. Информационная панель откроется на случайном порту, например 17 143 (см. вывод ниже):

```
info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 8.0.0+6596fdc41a8d419876a6bf4abc17b7c66b9ef63a
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is: C:\tools-skills-net8\Chapter16\
AspireStarter\AspireStarter.AppHost
info: Aspire.Hosting.DistributedApplication[0]
      Now listening on: https://localhost:17143
info: Aspire.Hosting.DistributedApplication[0]
      Login to the dashboard at https://localhost:17143/
login?t=88c817f3679614a06778de42009aa624
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application started. Press Ctrl+C to shut down.
```



В некоторых случаях может потребоваться разрешить работу без защищенного HTTPS-протокола. Для этого установите переменную среды `ASPIRE_ALLOW_UNSECURED_TRANSPORT` равной `true`. Подробнее об этом можно узнать на сайте [learn.microsoft.com/ru-ru/dotnet/aspire/troubleshooting/allow-unsecure-transport](http://learn.microsoft.com/ru-ru/dotnet/aspire/troubleshooting/allow-unsecure-transport).

## Структура начального решения Aspire

Теперь, когда начальное решение Aspire создано и успешно запущено, изучим его подробнее.

- На информационной панели в браузере видно, что запущены Redis-контейнер `cache`, а также два проекта — сервис веб-API с именем `apiservice` и клиент Blazor с именем `webfrontend` (рис. 16.5).

The screenshot shows the 'Resources' section of the AspireStarter interface. On the left, there are navigation links for 'Console', 'Structured', 'Traces', and 'Metrics'. The main table lists three resources:

Type	Name	State	Start time	Source	Endpoints	Logs	Data...
Container	cache	Running	8:53:17 AM	docker.io/library/re...	tcp://localhost:58709	<a href="#">View</a>	<a href="#">View</a>
Project	apiservice	Running	8:53:17 AM	AspireStarter.ApiSer...	+2	<a href="#">View</a>	<a href="#">View</a>
Project	webfrontend	Running	8:53:18 AM	AspireStarter.Web.c...	<a href="https://localhost:7031">https://localhost:7031</a> +1	<a href="#">View</a>	<a href="#">View</a>

**Рис. 16.5.** На информационной панели Aspire отображаются один контейнер и два проекта

- В строке `apiservice` в столбце `Endpoints` (Конечные точки) нажмите кнопку `+2`, чтобы раскрыть список двух конечных точек с номерами портов. Затем щелкните на ссылке `https`, чтобы открыть веб-сервис, и изучите отображенный прогноз погоды (рис. 16.6).

The screenshot shows a browser window with the URL `localhost:7403/weatherforecast`. The page displays a JSON response under 'Pretty print':

```
[{"date": "2024-05-01", "temperatureC": 34, "summary": "Hot", "temperatureF": 93}, {"date": "2024-05-02", "temperatureC": 44, "summary": "Freezing", "temperatureF": 111}, {"date": "2024-05-03", "temperatureC": 46, "summary": "Cool", "temperatureF": 114}, {"date": "2024-05-04", "temperatureC": 44, "summary": "Freezing", "temperatureF": 111}, {"date": "2024-05-05", "temperatureC": -11, "summary": "Sweltering", "temperatureF": 13}]
```

**Рис. 16.6.** Фиктивный прогноз погоды из веб-сервиса

- В строке `webfrontend` в столбце `Endpoints` (Конечные точки) щелкните на ссылке `https`, чтобы открыть клиентское приложение Blazor. Перейдите на страницу `Weather` (Погода) и изучите таблицу с прогнозами (рис. 16.7).
- Вернитесь на панель Aspire, перейдите на вкладку `Console` (Консоль) и по очереди выберите каждый из трех ресурсов, чтобы изучить их журналы работы (рис. 16.8).



Можно также перейти по ссылкам `View` (Просмотр) в столбце `Logs` (Журналы) для каждой строки ресурса, чтобы сразу открыть журналы соответствующего ресурса.

The screenshot shows a Blazor application interface. On the left is a sidebar with navigation links: Home, Counter, and Weather (which is currently selected). The main content area has a title "Weather" and a subtitle "This component demonstrates showing data loaded from a backend API service." Below this is a table with five rows of weather data:

Date	Temp. (C)	Temp. (F)	Summary
01/05/2024	-3	27	Scorching
02/05/2024	31	87	Sweltering
03/05/2024	-14	7	Chilly
04/05/2024	9	48	Chilly
05/05/2024	49	120	Hot

Рис. 16.7. Веб-фронтенд Blazor

The screenshot shows the Aspire information panel with the "Console logs" tab selected. On the left is a sidebar with icons for Resources, Console, Structured, Traces, and Metrics. The main area shows a dropdown menu titled "Select a resource" with three options: "apiservice", "cache", and "webfrontend".

Рис. 16.8. Журналы консоли на информационной панели Aspire

5. В журналах консоли для `webfrontend` видно, что приложение успешно подключилось к контейнеру Redis и выполнило успешный запрос к веб-сервису, используя библиотеку Polly для повторных попыток. Вот пример вывода:

```
2024-04-30T08:58:13.2905685 info: StackExchange.Redis.
ConnectionMultiplexer[0]
    Connecting (sync) on .NET 8.0.4 (StackExchange.Redis: v2.7.27.49176)
...
2024-04-30T08:58:13.4358566 info: StackExchange.Redis.
ConnectionMultiplexer[0]
    localhost:58709/Interactive: Connected
...
2024-04-30T09:06:57.1032079 info: System.Net.Http.HttpClient.
WeatherApiClient.ClientHandler[100]
    Sending HTTP request GET https://localhost:7403/weatherforecast
2024-04-30T09:06:57.1305086 info: System.Net.Http.HttpClient.
WeatherApiClient.ClientHandler[101]
    Received HTTP response headers after 18.2274ms - 200
2024-04-30T09:06:57.1549674 info: Polly[3]
    Execution attempt. Source: '-standard//Standard-Retry', Operation Key: '',
    Result: '200', Handled: 'False', Attempt: '0', Execution Time: '68.7295'
```

```
2024-04-30T09:06:57.1775001 info: System.Net.Http.HttpClient.
WeatherApiClient.LogicalHandler[101]
End processing HTTP request after 178.3335ms - 200
```

6. Перейдите на вкладку Structured (Структурированные), чтобы просмотреть структурированные журналы (рис. 16.9).

Resource	Level	Timestamp	Message	Trace	Data...
webfront...	Information	9:06:57.10...	Sending HTTP request GET https://localhost:7403/weatherforecast	ff113d4	<a href="#">View</a>
webfront...	Information	9:06:57.13...	Received HTTP response headers after 18.2274ms - 200	ff113d4	<a href="#">View</a>
webfront...	Information	9:06:57.16...	Execution attempt. Source: 'standard//Standard-Retry', Operation Key:...	ff113d4	<a href="#">View</a>
webfront...	Information	9:06:57.18...	End processing HTTP request after 178.3335ms - 200	ff113d4	<a href="#">View</a>

Total: 58 results found

Рис. 16.9. Структурированные журналы на информационной панели Aspire

7. В столбце Traces (Трассировки) щелкните кнопкой мыши на GUID-значении. В трассировке видно, что маршрут /weather в приложении Blazor сначала обратился к кэшу Redis, затем к веб-сервису, поскольку данные еще не были кэшированы, а затем снова к Redis для записи данных в кэш (рис. 16.10).

Name	0μs	93.01ms	186.03ms	279.04ms	372.05ms	Details
- webfrontend GET /weather						<a href="#">View</a>
- webfrontend Browser L...	921.4μs					<a href="#">View</a>
- webfrontend cache DA...		7.59ms				<a href="#">View</a>
- webfrontend GET			15.67ms			<a href="#">View</a>
apiservice GET /weatherfo...				1.49ms		<a href="#">View</a>
- webfrontend cache DA...					3.33ms	<a href="#">View</a>

Рис. 16.10. Детали трассировки с вызовами между ресурсами проекта Aspire

8. Перейдите на вкладку Metrics (Метрики) и изучите доступные графики и таблицы. Можно увидеть, например, такие показатели, как `process.runtime.dot-net.thread_pool.threads.count` и `http.server.request.duration`.
9. Закройте браузер. В терминале узла приложения Aspire нажмите сочетание клавиш `Ctrl+C`, чтобы завершить работу распределенного приложения. Теперь давайте более подробно рассмотрим функционал Aspire.

## Подробный разбор возможностей Aspire

В стеке Aspire собрано множество компонентов. Давайте разберемся, какую роль играет каждая часть.

### Информационная панель разработчика для мониторинга

Когда запускается проект Aspire, автоматически открывается информационная панель разработчика. Это один из важнейших инструментов для отладки распределенных приложений: он предоставляет единое представление о сервисах вместе с их журналами, метриками и трассировками.

С помощью панели можно просматривать журналы всех проектов, а также распределенные трассировки запросов, например обращение к странице прогноза погоды. Трассировки особенно полезны для диагностики проблем в распределенных системах.

Информационная панель разработчика построена на открытых стандартах, которые используются и в рабочих системах мониторинга, таких как Grafana и Prometheus.

Панель отображается во время работы проекта `AppHost` и запускается автоматически при старте решения. Левая навигационная панель содержит ссылки на разные разделы панели.

- **Resources (Ресурсы).** Главная страница панели, на которой перечислены все проекты, контейнеры, исполняемые файлы и другие ресурсы решения Aspire. Здесь видно состояние каждого ресурса, а также ссылки на веб-сервисы и пользовательские веб-интерфейсы. Ошибки выделяются отдельно, чтобы вы могли быстро перейти к подробной информации и найти причину проблемы.
- **Console (Консоль).** Отображает журналы всех компонентов решения Aspire, выводимые в консоль или в стандартный текстовый поток.
- **Structured (Структурированные).** Позволяет работать с фильтруемыми журналами. В отличие от консольных журналов, где все свойства объединены в одну строку, здесь каждое свойство хранится отдельно, что упрощает поиск и анализ событий.
- **Traces (Трассировки).** Показывает путь выполнения одного действия через все уровни распределенного решения в виде цепочки вызовов. Трассировки помогают выявить ошибки, найти узкие места и лучше понять поведение системы.
- **Metrics (Метрики).** Раздел для просмотра метрик, собираемых с вашего приложения: от количества потоков пула до времени обработки HTTP-запросов.



**В копилку знаний:** подробнее об информационной панели Aspire можно прочитать на сайте Microsoft по адресу [learn.microsoft.com/ru-ru/dotnet/aspire/fundamentals/dashboard/overview](https://learn.microsoft.com/ru-ru/dotnet/aspire/fundamentals/dashboard/overview).

О том, как запустить информационную панель отдельно от решения Aspire, можно узнать на сайте [learn.microsoft.com/ru-ru/dotnet/aspire/fundamentals/dashboard/standalone](https://learn.microsoft.com/ru-ru/dotnet/aspire/fundamentals/dashboard/standalone).

## Проект AppHost для оркестрации ресурсов

Проект AppHost позволяет задать все потребности вашего решения и управлять его запуском на локальном компьютере разработчика. Aspire предлагает удобные абстракции, благодаря которым можно настроить обнаружение сервисов, переменные среды и параметры контейнеров, не погружаясь в детали вручную — если только вы сами этого не захотите.

Посмотрим, как все это устроено, на примере начального шаблона Aspire.

1. В файле `AspireStarter.AppHost.csproj` проекта видно, что проект хоста приложений — это консольное приложение, настроенное как узел Aspire. В нем указаны ссылки на два функциональных проекта, а также добавлены пакеты `Aspire.Hosting.AppHost` и `Aspire.Hosting.Redis`:

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

<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net8.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
  <IsAspireHost>true</IsAspireHost>
  <UserSecretsId>1d713f01-3bf2-4a15-9910-799f70fe499f</UserSecretsId>
</PropertyGroup>

<ItemGroup>
  <ProjectReference Include=".\\AspireStarter.ApiService\\AspireStarter.ApiService.csproj" />
  <ProjectReference Include=".\\AspireStarter.Web\\AspireStarter.Web.csproj" />
</ItemGroup>

<ItemGroup>
  <PackageReference Include="Aspire.Hosting.AppHost" Version="8.0.1" />
  <PackageReference Include="Aspire.Hosting.Redis" Version="8.0.1" />
</ItemGroup>

</Project>
```



Элемент `<ProjectReference>` в проекте AppHost работает иначе, чем обычная ссылка на библиотеку классов. Вместо того чтобы собирать проект и копировать его файл .dll в папку bin, эта ссылка просто сообщает инструментарию Aspire, как управлять соединениями между проектами: какие конечные точки использовать, какие порты и пути настраивать. Aspire при этом учитывает информацию из файлов launchSettings.json. Мы еще вернемся к этому подробнее позже в текущей главе.

2. В папке Properties в файле launchSettings.json прописаны URL-адреса:
  - для доступа к информационной панели (параметр applicationUrl);
  - для конечной точки сервиса Open Telemetry Protocol (OTLP), который собирает телеметрию;
  - и для конечной точки gRPC, через которую информационная панель получает данные.

Вот как это выглядит:

```
{  
  "$schema": "https://json.schemastore.org/launchsettings.json", "profiles": {  
    "https": {  
      "commandName": "Project",  
      "dotnetRunMessages": true,  
      "launchBrowser": true,  
      "applicationUrl": "https://localhost:17143;http://localhost:15099",  
      "environmentVariables": {  
        "ASPNETCORE_ENVIRONMENT": "Development",  
        "DOTNET_ENVIRONMENT": "Development",  
        "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "https://localhost:21184",  
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "https://localhost:22057"  
      }  
    },  
  },
```

3. В файле Program.cs организуется подключение контейнера для Redis и двух веб-проектов:

```
var builder = DistributedApplication.CreateBuilder(args);  
  
var cache = builder.AddRedis("cache");  
  
var apiService = builder.AddProject<Projects.AspireStarter_  
ApiService>("apiservice");  
  
builder.AddProject<Projects.AspireStarter_Web>("webfrontend")  
  .WithExternalHttpEndpoints()  
  .WithReference(cache)  
  .WithReference(apiService);  
  
builder.Build().Run();
```

4. Щелкните кнопкой мыши на типе AspireStarter\_ApiService и нажмите клавишу F12 (или щелкните правой кнопкой мыши и выберите команду Go To Definition (Перейти к определению)). В коде видно, что для проекта задано свойство ProjectPath, указывающее на путь к проекту, а также сгенерирован случайный номер порта:

```
// <auto-generated/>  
  
namespace Projects;
```

```
[global::System.CodeDom.Compiler.GeneratedCode("Aspire.Hosting", null)]
[global::System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverage(Justification =
    "Generated code.")]
[global::System.Diagnostics.DebuggerDisplay("Type = {GetType().Name},nq",
ProjectPath = {ProjectPath}")]
public class AspireStarter_ApiService : global::Aspire.Hosting.IProjectMetadata
{
    public string ProjectPath => """C:\tools-skills-net8\Chapter16\AspireStarter\
        AspireStarter.ApiService\AspireStarter.ApiService.csproj""";
}
```



Когда вы добавляете ссылку на проект в AppHost, Aspire автоматически создает вспомогательный код. Этот код сохраняет путь к подключенному проекту, чтобы инструментарий мог прочитать его настройки и правильно настроить взаимодействие — например, подключить конечные точки на основе конфигурации из `launchSettings.json`.

Важно понимать: добавление проекта в AppHost — это *не* обычная ссылка между двумя проектами .NET! «За кадром» SDK Aspire обрабатывает это особым образом и автоматически генерирует код — чтобы ваш проект появился на информационной панели и стал полноценной частью распределенного решения.



Ресурсом Aspire может быть не только проект на .NET. Это может быть любой исполняемый файл или любой другой тип проекта.

Когда вы вызываете метод `AddProject`, Aspire динамически читает файл `launchSettings.json`, чтобы определить такие параметры, как URL-адреса и номера портов. Сам проект AppHost тоже имеет свой `launchSettings.json`, благодаря чему он может управлять несколькими URL-адресами сразу — для всех подключенных проектов.

При добавлении проекта через `AddProject` можно указать дополнительный параметр `launchProfileName`. Например, `launchProfileName: "https"`. Если передать `null`, запуск произойдет без определенного профиля, а порт будет выбран случайным образом:

```
.WithEndpoint("https", e => e.Port = null) // Генерация случайного номера порта
```

## Проект `ServiceDefaults` для централизованной настройки

Проект `ServiceDefaults` служит общей точкой для настройки всех сервисов и компонентов Aspire. Посмотрим, как он устроен.

1. В файле `AspireStarter.ServiceDefaults.csproj` видно, что проект — это библиотека классов, настроенная как общий проект Aspire. В нем подключен

фреймворк `Microsoft.AspNetCore.App` и добавлены несколько пакетов для реализации отказоустойчивости, обнаружения сервисов и инструментов OpenTelemetry:

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

<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
  <IsAspireSharedProject>true</IsAspireSharedProject>
</PropertyGroup>

<ItemGroup>
  <FrameworkReference Include="Microsoft.AspNetCore.App" />

  <PackageReference Include="Microsoft.Extensions.Http.Resilience"
    Version="8.3.0" />
  <PackageReference Include="Microsoft.Extensions.ServiceDiscovery"
    Version="8.0.0" />
  <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol"
    Version="1.8.0" />
  <PackageReference Include="OpenTelemetry.Extensions.Hosting"
    Version="1.8.0" />
  <PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore"
    Version="1.8.1" />
  <PackageReference Include="OpenTelemetry.Instrumentation.Http"
    Version="1.8.1" />
  <PackageReference Include="OpenTelemetry.Instrumentation.Runtime"
    Version="1.8.0" />
</ItemGroup>

</Project>
```

2. В файле `Extensions.cs` определен класс `Extensions` в пространстве имен `Microsoft.Extensions.Hosting`. В этом классе созданы методы расширения для интерфейса `IApplicationBuilder`:

- `AddServiceDefaults` — базовый метод, который следует вызывать в каждом проекте, участвующем в распределенном приложении (например, в веб-сервисах и сайтах Blazor). Внутри вызывает следующий метод;
- `ConfigureOpenTelemetry` — включает OpenTelemetry с разумными настройками по умолчанию: настройка журналирования, метрик для ASP.NET Core, HTTP-клиентов и среды выполнения .NET, а также трассировки;
- `AddOpenTelemetryExporters` — настраивает экспортеры OTLP, считывая параметры из конфигурации, например из переменных среды;
- `AddDefaultHealthChecks` — добавляет базовую проверку работоспособности сервиса;
- `MapDefaultEndpoints` — настраивает конечные точки `/health` для проверки состояния сервиса и `/alive` для проверки работоспособности.



**Дельный совет:** если хотите добавить другую общую функциональность, лучше создать для нее отдельную библиотеку классов. Проект ServiceDefaults рекомендуется использовать только для настройки сервисов и компонентов Aspire.

## Как подключаются функциональные проекты в Aspire

Теперь давайте разберемся, что особенного в наших функциональных проектах — веб-сервисе и веб-интерфейсе пользователя — и какие настройки позволяют им работать вместе в распределенном решении Aspire.

1. В файле проекта `ApiService` добавлена ссылка на проект `ServiceDefaults`. С ее помощью вызывается метод `AddServiceDefaults`.
2. В файле `Program.cs` этого проекта добавлен вызов метода `AddServiceDefaults`:

```
// Добавление ServiceDefaults и компонентов Aspire
builder.AddServiceDefaults();
```

3. В файле веб-проекта также есть ссылка на проект `ServiceDefaults`, чтобы можно было использовать метод `AddServiceDefaults`.
4. Там же добавлена ссылка на пакет `Aspire.StackExchange.Redis.OutputCaching`.



Перед вами пример компонента .NET Aspire. Такие компоненты — это библиотеки классов-оберток, которые настраивают отдельные функции, например Redis, для работы в облачной среде. Позже в этой главе мы рассмотрим их подробнее.

5. В файле `Program.cs` веб-проекта выполнено подключение `AddServiceDefaults` и добавление Redis для кэширования вывода:

```
// Добавление ServiceDefaults и компонентов Aspire
builder.AddServiceDefaults();
builder.AddRedisOutputCache("cache");
```



Имя `cache` появилось потому, что Redis был зарегистрирован как ресурс Aspire в проекте хоста приложения через вызов `builder.AddRedis("cache")`.

6. В файле `Program.cs` веб-проекта настроено использование веб-фронтенда для вызова API веб-сервиса погоды:

```
builder.Services.AddHttpClient<WeatherApiClient>(client =>
{
    // В этом URL используется "https+http://", чтобы указать, что HTTPS
    // предпочтительнее HTTP.
    // Подробнее о разрешении схем обнаружения сервисов можно узнать
    // на сайте aka.ms/dotnet/sdschemes
    client.BaseAddress = new("https+http://apiservice");
});
```



Имя `apiservice` возникло при регистрации веб-сервиса в проекте хоста приложения с помощью команды `builder.AddProject<Projects.AspireStarter_ApiService>("apiservice")`.

Aspire запускает ваши проекты вместе с их зависимостями и автоматически настраивает их для взаимодействия. Благодаря этому разработчикам не нужно заботиться о портах и строках подключения. С помощью механизма обнаружения сервисов Aspire позволяет обращаться к ресурсам через логические имена, такие как `apiservice`, вместо IP-адресов и портов при выполнении HTTP-запросов.

В URL-адресах, которые вы используете для HTTP-вызовов через `IHttpClientFactory`, можно указывать такие имена, как `apiservice`. Эти вызовы автоматически обрабатываются через библиотеку Polly, настроенную Aspire для повышения отказоустойчивости — в том числе с поддержкой повторных попыток при временных сбоях.

Теперь посмотрим, как интегрировать Redis в проекты: сначала вручную, а затем — автоматически с помощью Aspire.

## Настройка Redis

Посмотрим, что обычно требуется от разработчика для настройки Redis.

1. Добавить ссылку на пакет клиента Redis.
2. Добавить ссылку на библиотеку проверок функционирования, чтобы приложение могло реагировать на недоступность Redis.



**Дельный совет:** добавление проверок функционирования для подтверждения доступности ключевых ресурсов, например Redis, делает решения более отказоустойчивыми. Однако этот шаг часто упускают. В Aspire он встроен автоматически. Вы увидите это позже в этой главе, на примерах с RabbitMQ в подразделе «Обзор эталонного приложения eShop», на этапах 9–11. Принцип работы для Redis будет аналогичным.

3. Настроить строку подключения к Redis.
4. Настроить клиента Redis так, чтобы он отправлял журналы в систему telemetry.
5. Настроить сбор журналов и метрик для Redis.
6. Продумать отказоустойчивость: реализовать политику повторных попыток с помощью библиотеки вроде Polly, чтобы обеспечить надежную работу вызовов Redis.



**Дельный совет:** реализация отказоустойчивости с помощью политики повторных попыток — один из важных, но часто забываемых аспектов. Без нее приложение может ломаться из-за временных сбоев, которых легко избежать. Подробнее о стратегиях повтора можно узнать в моей книге «.NET 8: приложения и сервисы» или на странице [github.com/App-vNext/Polly/blob/main/docs:strategies/retry.md](https://github.com/App-vNext/Polly/blob/main/docs:strategies/retry.md).

А теперь посмотрим, что нужно сделать при использовании Aspire.

1. Добавить ссылку на пакет компонента Aspire для Redis.
2. Вызвать метод `AddRedis`

И все! Больше никаких шагов.

Компонент Aspire Redis автоматически настраивает клиент Redis на использование строки подключения, предоставленной системой. Это помогает избежать множества распространенных ошибок, которые часто возникают при ручной настройке локальной среды.

При желании можно изменить стандартную конфигурацию, но по умолчанию компонент Aspire Redis уже обеспечивает оптимальные настройки, подходящие для работы в реальной эксплуатации. При этом Aspire не скрывает оригинальный клиент Redis — ваше приложение использует ту же самую библиотеку Redis без дополнительной обертки, так что производительность остается прежней.



**Внимание!** Недавно лицензионное соглашение Redis изменилось. Подробнее об этом можно прочитать на сайте [redis.com/blog/redis-adopts-dual-source-available-licensing/](https://redis.com/blog/redis-adopts-dual-source-available-licensing/). В рабочих средах стоит рассмотреть совместимые альтернативы, например Dragonfly ([www.dragonflydb.io/redis-alternative](https://www.dragonflydb.io/redis-alternative)) или KeyDB ([docs.keydb.dev/](https://docs.keydb.dev/)).

Рассмотрев на примере Redis, как Aspire упрощает настройку, давайте разберемся, что собой представляют компоненты Aspire.

## Компоненты Aspire

Компоненты Aspire помогают преодолеть одну из главных трудностей облачной разработки: необходимость настраивать множество параметров, когда непонятно, с чего начать и что действительно важно. Aspire облегчает задачу, потому что для каждого компонента заранее определено, что он должен предоставлять. Причем каждый компонент обязан по умолчанию обеспечивать отказоустойчивость, проверку состояния, установку телеметрии и интеграцию с контейнером внедрения зависимостей (DI).

В частности, для компонентов Aspire предусмотрены такие требования.

- **Схема JSON.** Компонент должен предоставлять схему, чтобы в `appsettings.json` работало автодополнение и стало проще настраивать параметры.

- **Отказоустойчивость.** Компонент обязан включать готовые механизмы, такие как повторные попытки, тайм-ауты и автоматические выключатели, чтобы приложение оставалось доступным даже при сбоях. При этом настройки можно изменить под свои задачи.
- **Проверка состояния.** Для отслеживания работоспособности каждый компонент должен поддерживать проверки состояния.
- **Телеметрия.** Компонент должен сразу работать с журналами, метриками и трассировками через стандартные средства .NET – такие как `ILogger`, `Meter` и `Activity`.
- Интеграция с DI.** Компонент должен предоставлять методы расширения, чтобы легко регистрировать сервисы в контейнере внедрения зависимостей с правильным временем жизни.

Сейчас в Aspire доступны следующие компоненты (табл. 16.4).

**Таблица 16.4.** Компоненты Aspire

Компонент	Описание
SQL Server EF Core	Для работы с базами данных SQL Server через Entity Framework Core
SQL Server	Для прямого доступа к базам данных SQL Server
PostgreSQL EF Core	Для работы с базами данных PostgreSQL через Entity Framework Core
PostgreSQL	Для прямого доступа к базам данных PostgreSQL
RabbitMQ	Для взаимодействия с системой обмена сообщениями RabbitMQ
Redis Distributed Caching	Для использования Redis в качестве распределенного кэша
Redis Output Caching	Для кэширования вывода с использованием Redis
Redis	Для прямого доступа к Redis
Apache Kafka	Для получения и обработки сообщений через брокер Apache Kafka
MongoDB Driver	Для доступа к базам данных MongoDB
Oracle EF Core	Для работы с базами данных Oracle через Entity Framework Core
MySQLConnector	Для подключения к базам данных MySQL

В Aspire также предусмотрены отдельные компоненты для работы с сервисами Azure (табл. 16.5).

**Таблица 16.5.** Специфические компоненты Aspire для Azure

Компонент	Описание
Azure AI OpenAI	Для взаимодействия с сервисами Azure AI OpenAI или с функциональностью OpenAI
Azure Blob Storage	Для работы с хранилищами BLOB-объектов Azure Blob Storage
Azure Cosmos DB EF Core	Для доступа к базам данных Azure Cosmos DB через Entity Framework Core
Azure Cosmos DB	Для прямого доступа к базам данных Azure Cosmos DB
Azure Key Vault	Для работы с хранилищем ключей Azure Key Vault
Azure Service Bus	Для использования очередей и топиков Azure Service Bus
Azure Storage Queues	Для работы с очередями Azure Storage Queues



Актуальный список всех доступных компонентов Aspire опубликован на сайте [learn.microsoft.com/ru-ru/dotnet/aspire/fundamentals/components-overview#available-components](https://learn.microsoft.com/ru-ru/dotnet/aspire/fundamentals/components-overview#available-components).

## Журналирование, трассировка и сбор метрик для обеспечения наблюдаемости

Чтобы распределенное решение было действительно наблюдаемым, важно соблюдать несколько условий.

- Каждый компонент должен предоставлять данные в удобной для обработки форме — это касается как самой платформы .NET, так и подключаемых библиотек и кода вашего проекта.
- Все собранные данные нужно отправлять в хранилище, к которому можно получить доступ для дальнейшей работы.
- Должны быть настроены инструменты для просмотра, запроса и анализа этих данных.

Решения на основе Aspire изначально спроектированы так, чтобы быть наблюдаемыми. Благодаря этому вы сможете получать полную картину происходящего в приложении — по журналам, метрикам и трассировкам, собираемым со всех работающих ресурсов.

Команды разработчиков .NET делают ставку на OpenTelemetry как на основной формат данных: они приняли его структуру и правила именования, а для передачи данных используют протокол OTLP. Вы уже знакомились с OpenTelemetry в главе 5.

Aspire автоматически включает поддержку OpenTelemetry через проект Service-Defaults. Вместо того чтобы жестко вшивать реализацию прямо в проекты, Aspire применяет подход с совместно используемым кодом. Это позволяет настраивать такие соглашения, как имена конечных точек для сбора телеметрии, если возникает такая необходимость. Альтернативный способ — создание отдельной библиотеки с параметрами настройки — оказался бы более сложным.

Все журналы, метрики и трассировки вашего решения Aspire выводятся на информационной панели разработчика. Раздел **Traces** (Трассировки) помогает быстро находить проблемные места — например, действия пользователей, которые вызывают неэффективные или медленные пути в приложении.

Благодаря трассировке вы легко обнаружите потенциальные проблемы вроде лишних вызовов базы данных или сервисов, замедляющих работу системы. Без таких данных найти и устранить подобные узкие места было бы гораздо сложнее.

## Docker и Podman для контейнеров

В начальной версии Aspire контейнеры можно размещать двумя способами: через Docker (по умолчанию) или Podman.

Преимущества Docker:

- развитая экосистема;
- широкое сообщество;
- множество сторонних интеграций;
- поддержка таких функций, как Docker Compose, для управления многоконтейнерными решениями.

Преимущество Podman:

- более низкая стоимость лицензий.

Если нет серьезных причин использовать Podman (например, если ваша команда уже выбрала его в качестве стандарта), лучше отдать предпочтение Docker. Именно его выбрала команда Aspire в качестве основного варианта, и на это есть веские причины, которые перечислены выше.

## Ожидание готовности контейнеров

Если вы уже работали с инструментом Docker Compose, то, возможно, знаете об одной распространенной проблеме. Когда используется инструкция `depends_on`, Docker Compose ожидает только запуска контейнера, а не его полной готовности. Это может привести к ошибкам: например, если база данных еще не завершила инициализацию, другой сервис может попытаться подключиться к ней слишком рано.

На момент написания книги я столкнулся с похожей ситуацией. Вызов метода `WithReference` действует примерно так же, как `depends_on` в Docker Compose: он ожидает запуска контейнера, но не проверяет, готов ли ресурс к работе.

В .NET Aspire 9, выпущенном в ноябре 2024 года, была реализована долгожданная функция — возможность ожидания готовности зависимых ресурсов перед запуском основного приложения. Теперь с помощью методов `WaitFor` и `WaitForCompletion` можно указать, что один ресурс должен дождаться полной готовности другого. Это особенно полезно для предотвращения ошибок, возникающих при попытке подключения к еще не готовым сервисам, таким как базы данных или очереди сообщений.

Механизм ожидания готовности основан на стандартных проверках работоспособности .NET (*health checks*). Если ресурс предоставляет такие проверки, `WaitFor` будет использовать их для определения готовности. В противном случае приложение будет ожидать, пока ресурс перейдет в состояние «*Running*». Это позволяет более надежно управлять порядком запуска компонентов распределенного приложения.

Таким образом, начиная с версии Aspire 9 разработчики получили встроенный инструмент для управления зависимостями между ресурсами, что значительно упрощает разработку и отладку сложных распределенных систем.

Подробнее см. на сайте <https://learn.microsoft.com/ru-ru/dotnet/aspire/whats-new/dotnet-aspire-9>.



Можно также воспользоваться методом `WaitFor`, который написал Дэвид Фаулер. Ознакомиться с его кодом можно в репозитории GitHub по адресу [github.com/davidfowl/WaitForDependenciesAspire](https://github.com/davidfowl/WaitForDependenciesAspire).

## А как же инструменты Dapr, Orleans и Project Tye?

У компании Microsoft есть и другие родственные проекты, которые похожи на Aspire или могут использоваться вместе с ним. Иногда это может немного запутывать, поэтому коротко разберемся, что это за технологии и как они соотносятся с Aspire.

### Dapr

*Dapr* (*Distributed Application Runtime*) — это проект с открытым исходным кодом, созданный для упрощения разработки отказоустойчивых приложений на основе микросервисов. Он помогает решать типичные задачи, такие как управление состоянием, взаимодействие между сервисами и организация наблюдаемости, предоставляя для этого удобные API.

Dapr построен по архитектуре *Sidecar*: к каждому микросервису добавляется отдельный процесс Dapr, который берет на себя задачи проксирования, работы с внешними системами и межсервисного взаимодействия, при этом изменять код самого микросервиса не требуется.

Dapr поддерживает разные языки программирования и фреймворки, взаимодействие с ним происходит через HTTP или gRPC. Поэтому он так удобен для многих разработчиков, независимо от выбранной платформы.

Проект развивается благодаря поддержке большого сообщества и ведущих компаний отрасли, а не только Microsoft. Это помогает Dapr оставаться современным и учитывающим реальные сценарии использования. Вы можете использовать Dapr вместе с Aspire. В частности, готовый пример доступен на сайте GitHub по адресу [github.com/dotnet/aspire-samples/tree/main/samples/AspireWithDapr](https://github.com/dotnet/aspire-samples/tree/main/samples/AspireWithDapr).



**В копилку знаний:** подробнее о Dapr можно узнать на сайте [dapr.io](https://dapr.io).

## Orleans

*Orleans* – это фреймворк для создания распределенных и масштабируемых вычислительных приложений. Его разработали в подразделении Microsoft Research, и сегодня Orleans широко используется внутри компании – например, он обеспечивает работу сервисов для таких игр, как Halo, а также других крупных проектов.

Orleans основан на акторной модели, но вводит собственную концепцию – *виртуальные акторы*, которые называются «зернами» (*grains*). Зерна становятся основными единицами вычислений и хранения состояния в приложении. Orleans скрывает всю сложность управления их жизненным циклом – разработчикам не нужно заботиться о создании, уничтожении или размещении акторов вручную.

Фреймворк включает встроенные механизмы обработки сбоев, например автоматическую активацию резервных зерен при отказах серверов. Писать код с Orleans удобно: можно думать обо всех зернах как об элементах, находящихся на одном сервере, не вникая в детали их размещения или маршрутизации запросов.

Orleans поддерживает:

- сохранение состояния зерен;
- потоки событий – зерна могут публиковать события и подписываться на них, что упрощает построение событийно-ориентированных архитектур;
- таймеры и напоминания – зерна могут выполнять действия через заданные интервалы времени.

Orleans особенно хорошо подходит для приложений, которым нужны высокая пропускная способность и минимальные задержки, например игры в реальном времени, интернет вещей (IoT) и аналитика в реальном времени. В целом Orleans помогает строить масштабируемые распределенные системы, предлагая более гибкий подход по сравнению с традиционной микросервисной архитектурой.

Команда разработчиков Aspire также подготовила пример интеграции Aspire и Orleans – его можно найти в репозитории образцов Aspire по адресу [github.com/dotnet/aspire-samples/tree/main/samples/OrleansVoting](https://github.com/dotnet/aspire-samples/tree/main/samples/OrleansVoting).



**В копилку знаний:** подробнее об Orleans можно узнать на сайте [learn.microsoft.com/ru-ru/dotnet/orleans/](https://learn.microsoft.com/ru-ru/dotnet/orleans/).

## Project Tye

Project Tye – это экспериментальный инструмент для разработчиков, созданный для упрощения процесса разработки, тестирования и развертывания микросервисов и распределенных приложений. Он помогал решать типичные задачи, с которыми сталкиваются команды при работе с несколькими сервисами одновременно, например:

- оркестрация множества сервисов в процессе локальной разработки;
- упрощение развертывания микросервисов в рабочей среде.

Можно сказать, что Aspire вырос из идей Tye. Project Tye был своего рода экспериментальной площадкой, а Aspire стал зрелой производственной реализацией тех же принципов. Например, информационная панель разработчика, которая теперь встроена в Aspire, изначально появилась именно в рамках Project Tye.



**В копилку знаний:** историю завершения эксперимента Project Tye можно прочитать на сайте [github.com/dotnet/tye/issues/1622](https://github.com/dotnet/tye/issues/1622).

## Выбор между Dapr, Orleans и Aspire

Во время одного из публичных собраний команда Aspire поделилась своими мыслями о выборе между этими технологиями:

*«Есть ответ. А есть то, что люди хотят услышать. Эти вопросы подсказывают мне, что люди хотят услышать: “Какой из них использовать?” Верно? Но такого прямого ответа вы от меня не получите. Есть много вариантов, и важно понимать, какого результата можно добиться с каждой техникой. Orleans отлично подходит для виртуальных акторов. Dapr очень хорош, когда у вас несколько языков в системе. Оркестрация Aspire прекрасно сочетается с обеими концепциями. Aspire не пытается скрыть облако за абстракциями. Он работает с любыми клиентскими библиотеками и существующими системами».*

Дэвид Фаулер

Один из разработчиков также поделился впечатлением в чате:

*«Работать с Aspire в связке с Dapr – одно удовольствие».*

Лоран Кемпе



Другие комментарии и мнения можно прочитать по адресу [devblogs.microsoft.com/dotnet/introducing-dotnet-aspire-simplifying-cloud-native-development-with-dotnet-8/](https://devblogs.microsoft.com/dotnet/introducing-dotnet-aspire-simplifying-cloud-native-development-with-dotnet-8/).

## Применение Aspire в новых и существующих решениях

Теперь, когда мы познакомились с начальным шаблоном и получили общее представление о частях Aspire и их совместной работе, давайте перейдем к более реальным задачам. Мы посмотрим, как можно создавать новые решения Aspire и как добавить Aspire в уже существующие проекты.

### Создание решения Aspire

На этом этапе мы уже изучили полноценное решение Aspire с несколькими распространенными проектами и компонентами — такими как веб-сервис, пользовательский интерфейс и контейнер Redis. Если вам нужно создать совершенно новое распределенное решение, можно воспользоваться шаблоном проекта **.NET Aspire Application** (Пустое приложение .NET Aspire).

Представьте ситуацию: вы хотите создать систему, которая будет использовать базы данных PostgreSQL и SQL Server, а также веб-интерфейс на ASP.NET Core.

Создадим начальное решение Aspire для задуманной системы.

1. В редакторе кода создайте проект **.NET Aspire Application** (Пустое приложение .NET Aspire) / `aspire` с решением `AspireNew` во вложенной папке каталога `Chapter16`.



**Внимание!** Убедитесь, что создаете отдельное новое решение, а не добавляете его в уже существующее. Путь должен быть таким: `C:\tools-skills-net8\Chapter16\AspireNew\`.

2. После создания решения обратите внимание: в нем должны быть два проекта с именами `AspireNew.AppHost` и `AspireNew.ServiceDefaults`.
3. В проекте `AspireNew.ServiceDefaults` откройте файл `Extensions.cs` и изучите реализацию метода `AddServiceDefaults`. Он настраивает проверки работоспособности, включает `OpenTelemetry`, задает параметры отказоустойчивости с помощью библиотеки `Polly` и настраивает обнаружение сервисов для HTTP-клиентов:

```
public static IApplicationBuilder AddServiceDefaults(
    this IApplicationBuilder builder)
{
    builder.ConfigureOpenTelemetry();

    builder.AddDefaultHealthChecks();

    builder.Services.AddServiceDiscovery();

    builder.Services.ConfigureHttpClientDefaults(http =>
    {
```

```
// Добавление отказоустойчивости по умолчанию
http.AddStandardResilienceHandler();

// Добавление обнаружения сервисов по умолчанию
http.AddServiceDiscovery();
});

return builder;
}
```

4. В этом файле обратите внимание на реализацию метода `MapDefaultEndpoints`, который настраивает конечные точки для проверки работоспособности при разработке:

```
public static WebApplication MapDefaultEndpoints(
    this WebApplication app)
{
    // Добавление конечных точек проверки функциональности в средах,
    // не связанных с разработкой, имеет последствия для безопасности.
    // Подробнее: aka.ms/dotnet/aspire/healthchecks
    if (app.Environment.IsDevelopment())
    {
        // Все проверки должны быть пройдены, чтобы приложение считалось
        // готовым к приему трафика после запуска.
        app.MapHealthChecks("/health");

        // Только проверки с тегом "live" необходимы для состояния "работает"
        app.MapHealthChecks("/alive", new HealthCheckOptions
        {
            Predicate = r => r.Tags.Contains("live")
        });

        // Для включения конечной точки Prometheus (требуется пакет
        // OpenTelemetry.Exporter.Prometheus.AspNetCore) раскомментируйте
        // следующую строку:
        // app.MapPrometheusScrapingEndpoint();
    }
    return app;
}
```

5. Открыв в проекте `AspireNew.AppHost` файл `Program.cs`, вы увидите, что пока создается только пустой распределенный сборщик приложений, без добавления конкретных ресурсов:

```
var builder = DistributedApplication.CreateBuilder(args);

builder.Build().Run();
```

6. Теперь добавим в решение `AspireNew` проект веб-API со следующими настройками:
- выберите шаблон проекта ASP.NET Core Web API (Веб-API ASP.NET Core) / `webapi`;
  - в поле `Solution name` (Имя решения) укажите значение `AspireNew`;
  - в поле `Project name` (Имя проекта) укажите значение `Northwind.WebApi`;

- в раскрывающемся списке Authentication type (Тип проверки подлинности) выберите пункт None (Отсутствует);
- установите флажок Configure for HTTPS (Настроить для HTTPS);
- снимите флажок Enable Docker (Включить поддержку контейнера);
- снимите флажок Enable OpenAPI support (Включить поддержку OpenAPI);
- снимите флажок Do not use top-level statements (Не использовать операторы верхнего уровня);
- снимите флажок Use controllers (Использовать контроллеры);
- установите флажок Enlist in .NET Aspire orchestration (Включить в список проект оркестрации .NET Aspire).



**Внимание!** Программа Rider на момент чтения книги может не иметь полноценной поддержки оркестрации .NET Aspire. В этом случае рекомендуется создать проект с помощью команды dotnet new, а затем вручную добавить его в решение в программе Rider.

7. В проекте Northwind.WebApi откройте файл Program.cs и обратите внимание на вызовы методов AddServiceDefaults и MapDefaultEndpoints, которые настраивают сервисы в контейнере:

```
var builder = WebApplication.CreateBuilder(args);

builder.AddServiceDefaults();

// Добавление сервисов в контейнер

var app = builder.Build();

app.MapDefaultEndpoints();
```

8. В проекте AspireNew.AppHost откройте файл Program.cs и обратите внимание, что в построителе распределенного приложения добавлен ресурс проекта для нашего веб-API:

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddProject<Projects.Northwind_WebApi>("northwind-webapi");

builder.Build().Run();
```

9. На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши по проекту AspireNew.AppHost и выберите в контекстном меню команду Add ▶ .NET Aspire package (Добавить ▶ Пакет .NET Aspire). Обратите внимание, что менеджер пакетов NuGet применяет фильтр (owner:Aspire tags:hosting), отображая только совместимые пакеты для проектов типа AppHost.
10. В поле фильтра после существующего текста введите postgresql, выберите пакет Aspire.Hosting.PostgreSQL, установите последнюю доступную версию

и нажмите кнопку **Install** (Установить). При необходимости примите лицензионное соглашение.

11. После установки пакета для PostgreSQL измените фильтр, заменив `postgresql` на `sqlserver`, выберите пакет `Aspire.Hosting.SqlServer`, установите его последнюю версию и нажмите кнопку **Install** (Установить). При необходимости также примите лицензионное соглашение.
12. В файле `AspireNew.AppHost.csproj` проекта проверьте, что пакеты для размещения PostgreSQL и SQL Server добавлены корректно:

```
<ItemGroup>
  <PackageReference Include="Aspire.Hosting.AppHost" Version="8.0.1" />
  <PackageReference Include="Aspire.Hosting.PostgreSQL" Version="8.0.1"/>
  <PackageReference Include="Aspire.Hosting.SqlServer" Version="8.0.1" />
</ItemGroup>
```

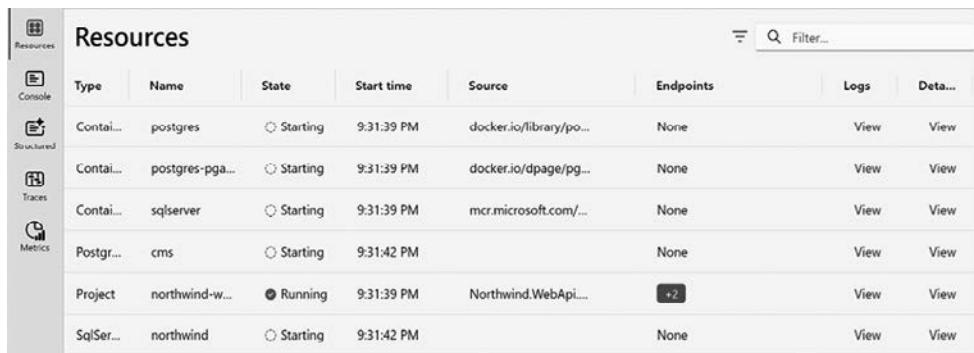
13. Выполните сборку проекта `AspireNew.AppHost`, чтобы восстановить все зависимости.
14. В файле `Program.cs`, сразу после создания конструктора, добавьте выражения для настройки двух баз данных:

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddPostgres("postgres")
  .WithPgAdmin()
  .AddDatabase("cms");

builder.AddSqlServer("sqlserver")
  .AddDatabase("northwind");
```

15. Запустите приложение Docker Desktop.
16. Запустите проект `AspireNew.AppHost`.
17. Когда в браузере откроется панель разработчика Aspire, обратите внимание, что запуск двух серверов баз данных может занять немного больше времени (рис. 16.11).

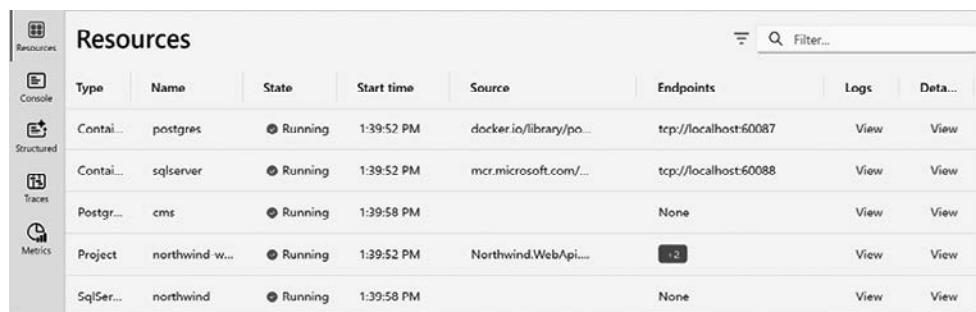


The screenshot shows the Aspire Resources panel. On the left, there is a sidebar with icons for Resources, Console, Structured, Traces, and Metrics. The main area is titled "Resources" and contains a table with the following data:

Type	Name	State	Start time	Source	Endpoints	Logs	Data...
Container	postgres	Starting	9:31:39 PM	docker.io/library/postgres	None	View	View
Container	postgres-pgadmin	Starting	9:31:39 PM	docker.io/dpage/pgadmin4	None	View	View
Container	sqlserver	Starting	9:31:39 PM	mcr.microsoft.com/mssql/server	None	View	View
PostgreSQL	cms	Starting	9:31:42 PM		None	View	View
Project	northwind-webapi	Running	9:31:39 PM	Northwind.WebApi...	+2	View	View
SqlServer	northwind	Starting	9:31:42 PM		None	View	View

**Рис. 16.11.** Два сервера баз данных все еще запускаются

18. Через некоторое время оба сервера баз данных будут запущены, их конечные точки станут активными, и базы данных будут доступны (рис. 16.12).



The screenshot shows the 'Resources' section of the Aspire interface. On the left is a sidebar with icons for Console, Structured, Traces, and Metrics. The main area has a table titled 'Resources' with columns: Type, Name, State, Start time, Source, Endpoints, Logs, and Data... . There are two rows for PostgreSQL servers:

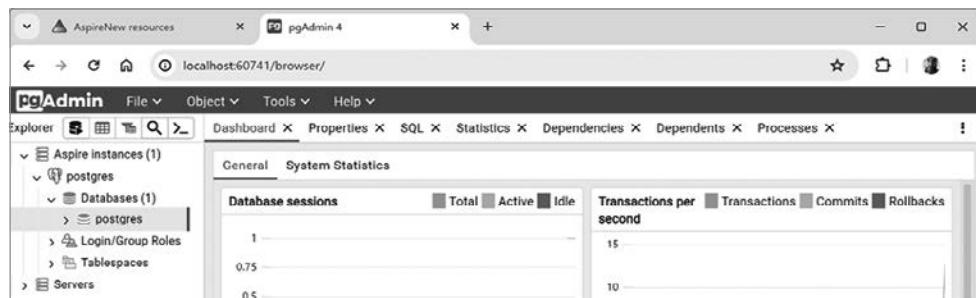
Type	Name	State	Start time	Source	Endpoints	Logs	Data...
Container	postgres	Running	1:39:52 PM	docker.io/library/postgres:13.1-alpine	tcp://localhost:60087	View	View
Container	sqlserver	Running	1:39:52 PM	mcr.microsoft.com/azure-sql-edge:1.0.1	tcp://localhost:60088	View	View
PostgreSQL	cms	Running	1:39:58 PM		None	View	View
Project	northwind	Running	1:39:52 PM	Northwind.WebApi:5001	+3	View	View
SqlServer	northwind	Running	1:39:58 PM		None	View	View

**Рис. 16.12.** Работающие серверы и их базы данных



Когда вы вызываете метод `AddSqlServer`, по умолчанию используется случайный порт. Если необходимо задействовать стандартный порт 1433, нужно задать его явно.

19. На панели инструментов Aspire щелкните кнопкой мыши на конечной точке `postgres-pgadmin`. В браузере откроется веб-интерфейс для управления сервером PostgreSQL и его базами данных (рис. 16.13).



**Рис. 16.13.** Веб-интерфейс пользователя pgadmin

20. Закройте браузер и завершите работу распределенного приложения Aspire.

Итак, вы создали новое решение Aspire и запустили его!

Теперь более подробно разберем, как Aspire работает с базой данных PostgreSQL. Эти приемы одинаково применимы как к новым, так и к существующим решениям.

## Aspire и PostgreSQL

В проекте AppHost можно зарегистрировать сервер PostgreSQL как ресурс для локальной разработки. Пример регистрации:

```
var postgress = builder.AddPostgres("my-pgsql")
    .WithPgAdmin()
    .AddDatabase("mydb");
```

Здесь `my-pgsql` — имя сервера PostgreSQL, а `mydb` — имя базы данных. Aspire использует эти имена для идентификации ресурсов, поэтому они должны быть уникальными в пределах решения.

По умолчанию Aspire принимает, что имя строки подключения совпадает с именем базы данных (в данном случае оба значения — `mydb`). Например:

```
{
  "ConnectionStrings": {
    "mydb": "Host=localhost;Port=5432;Database=myDataBase;User
              ID=root;Password=myPassword; "
  },
}
```

В сервисе, которому требуется доступ к базе данных PostgreSQL, для работы с ней нужно зарегистрировать контекст данных EF Core:

```
// Поиск строки подключения с именем "my-pgsql"
builder.AddNpgsqlDbContext<NorthwindContext>("my-pgsql");
```

Рекомендуется согласовать имя строки подключения с именем базы данных. Например, `mydb`:

```
// Поиск строки подключения с именем "mydb"
builder.AddNpgsqlDbContext<NorthwindContext>("mydb");
```

При необходимости можно переопределить имя строки подключения, явно указав его в параметрах настройки:

```
// Переопределение имени строки подключения вручную
builder.AddNpgsqlDbContext<NorthwindContext>("my-pgsql", o =>
    o.ConnectionString = builder.Configuration.GetConnectionString("mydb")
);
```

Рассмотрим, какие типы объектов возвращаются методами регистрации ресурсов. Сводка приведена в табл. 16.6.



Метод `WithReference` принимает необязательный параметр `connectionName`, который задает имя строки подключения.

**Таблица 16.6.** Конструктор Aspire для PostgreSQL

Метод	Возвращает	Описание
AddPostgres	PostgresServerResource	Добавляет сервер PostgreSQL в модель приложения. Для локальной разработки используется контейнер
AddDatabase	PostgresDatabaseResource	Добавляет базу данных PostgreSQL. Если имя базы данных не указано, используется имя сервера

## Использование томов данных и настройка постоянного пароля

Когда используется том данных, случайно сгенерированный пароль сохраняется внутри тома. При следующем запуске решения Aspire будет сгенерирован новый пароль, и если это не учесть, произойдет сбой подключения. Эта проблема описана на странице [github.com/dotnet/aspire/issues/3669](https://github.com/dotnet/aspire/issues/3669).

Если вы применяете метод `WithDataVolume`, нужно обязательно установить постоянный пароль, а не полагаться на случайную генерацию при каждом запуске. Самый простой способ это сделать — сохранить пароль в пользовательских секре-тах проекта `AppHost`, используя ключ `Parameters:my-pgsql-password`.



Эта проблема была решена в версии Aspire 8.0.1 и более поздних: теперь Aspire поддерживает сохранение стабильных паролей с помощью механизма секретов. Подробнее об истории решения этой проблемы можно узнать на странице [github.com/dotnet/aspire/issues/1151](https://github.com/dotnet/aspire/issues/1151).

## Добавление Aspire к существующему решению

Aspire можно использовать и в существующих приложениях, постепенно внедряя его возможности. Aspire поддерживается только в .NET 8 и более поздних версиях. Если ваши проекты используют более ранние версии .NET, их необходимо обновить как минимум до версии .NET 8, прежде чем подключать какие-либо компоненты Aspire. Для работы с инструментарием Visual Studio потребуется версия программы 17.10 или новее.

Добавим Aspire в существующее решение.

1. В редакторе кода создайте проект ASP.NET Core Web API (Веб-API ASP.NET Core) / `webapi`, присвойте ему имя `Northwind.WebApi`, а решению — имя `AspireExisting`. Проект разместите во вложенной папке каталога `Chapter16`.

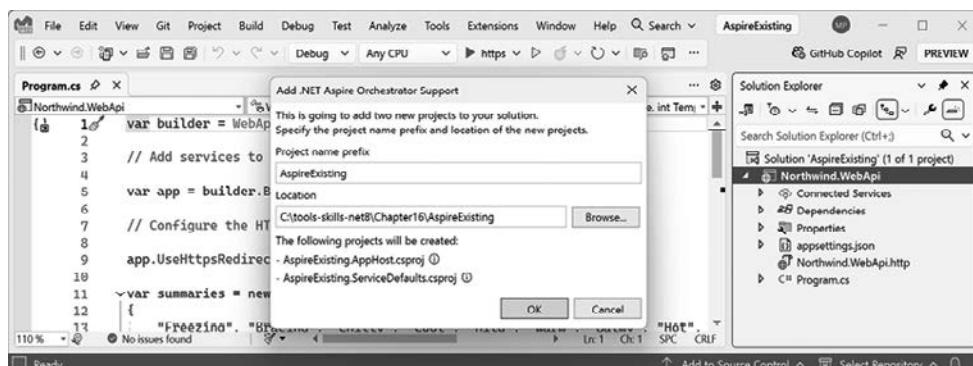


Не устанавливайте флажок Enlist in .NET Aspire orchestration (Включить в список проект оркестрации .NET Aspire)! В этом разделе важно начать с проекта, в который Aspire пока не добавлен, чтобы наглядно увидеть процесс его интеграции.



**Внимание!** Убедитесь, что создаете новое отдельное решение, а не добавляете проект в существующее. Проект нужно создать в каталоге C:\tools-skills-net8\Chapter16\AspireExisting\Northwind.WebApi.

2. Запустите проект без отладки.
3. Убедитесь, что веб-сервис возвращает несколько фиктивных прогнозов погоды в формате JSON.
4. Закройте браузер и остановите веб-сервис.
5. На панели Solution Explorer (Обозреватель решений) щелкните правой кнопкой мыши по проекту и в контекстном меню выберите команду Add ▶ .NET Aspire Orchestrator Support (Добавить ▶ Поддержка оркестратора .NET Aspire).
6. В диалоговом окне Add .NET Aspire Orchestrator Support (Добавить поддержку оркестратора .NET Aspire) просмотрите параметры, затем нажмите кнопку OK (рис. 16.14).



**Рис. 16.14.** Добавление поддержки оркестратора .NET Aspire

7. В файле Northwind.WebApi.csproj проекта появится ссылка на проект Aspire Service Defaults.
8. В файле Program.cs проекта Northwind.WebApi добавлены вызовы методов AddServiceDefaults и MapDefaultEndpoints.
9. В файле Program.cs проекта AspireExisting будет настроено подключение веб-сервиса через метод AddProject:

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddProject<Projects.Northwind_WebApi>("northwind-webapi");

builder.Build().Run();
```

## 10. Закройте решение.



Если вы работаете не в Visual Studio, проекты AppHost и ServiceDefaults можно создать вручную с помощью команды dotnet new, а затем добавить в решение. Однако их придется привязывать к проекту самостоятельно.

## Переход на компоненты Aspire

Если в ваших проектах используются пакеты NuGet, для которых есть аналоги в Aspire (например, Redis, MySQL, PostgreSQL и др.), вы можете перейти на компоненты Aspire. Это упростит настройку и поможет избежать явного задания некоторых параметров.



Компоненты Aspire можно использовать не только в решениях Aspire. Их можно подключить к любому приложению на .NET 8. Так вы получите преимущества вроде встроенной отказоустойчивости и удобной базовой настройки выбранного компонента. Однако такие возможности, как панель управления, автоматическое обнаружение сервисов, настройка портов и строк подключения, в этом случае недоступны.

## Обзор эталонного приложения eShop

Эталонное приложение eShop представляет собой вымышленный сайт под названием Northern Mountains. Это сайт электронной коммерции на платформе .NET, построенный с использованием архитектуры на основе сервисов (рис. 16.15).

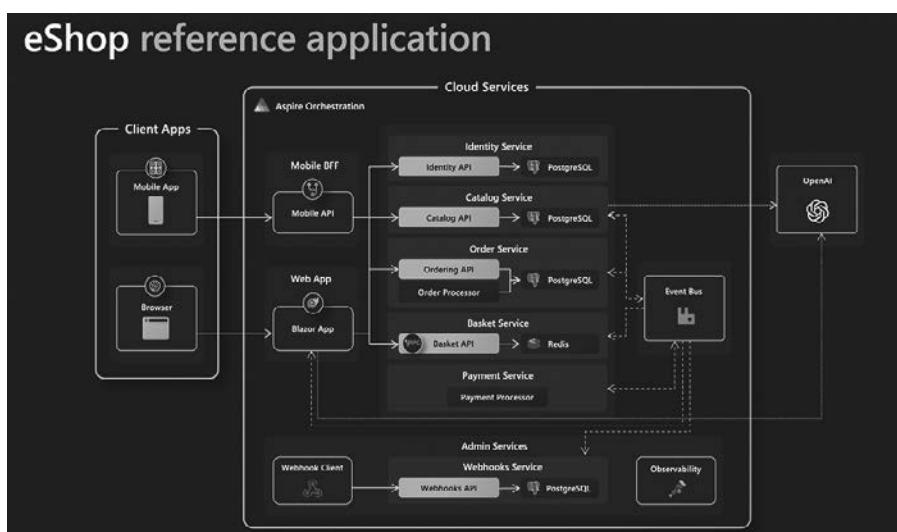


Рис. 16.15. Диаграмма архитектуры эталонного приложения eShop из репозитория Microsoft



Более крупную версию схемы можно посмотреть по ссылке [raw.githubusercontent.com/dotnet/eShop/main/img/eshop\\_architecture.png](https://raw.githubusercontent.com/dotnet/eShop/main/img/eshop_architecture.png).

Данные каталога товаров определены в файле `catalog.json`. Названия товаров, их описания и наименования брендов сгенерированы с помощью модели GPT-3.5 Turbo, а изображения товаров — с помощью модели DALL-E 3.

Посмотрим, как работает eShop с Aspire.

1. Перейдите в репозиторий eShop по адресу [github.com/dotnet/eShop](https://github.com/dotnet/eShop).
2. Загрузите или клонируйте проект eShop.
  - Чтобы загрузить проект, нажмите зеленую кнопку **Code (Код)**, выберите **Download ZIP (Загрузить архив ZIP)** и распакуйте архив, например, в папку `C:\eShop\`.
  - Чтобы клонировать проект, нажмите зеленую кнопку **Code (Код)** и выберите **Open with GitHub Desktop (Открыть с помощью GitHub Desktop)**.
3. Откройте файл решения `eShop.sln` в редакторе кода. Загрузка может занять некоторое время, поскольку в решении 25 проектов (20 проектов в папке `src` и 5 в папке `tests`).
4. В файле `eShop.AppHost.csproj` проекта можно увидеть ссылки на пакеты компонентов Aspire для RabbitMQ, Redis, PostgreSQL и Azure Cognitive Services, а также на десять проектов .NET:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <IsAspireHost>true</IsAspireHost>
    <IsPackable>false</IsPackable>
    <UserSecretsId>b99dbce4-17d4-41d2-858a-2b0529d60bb8</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Aspire.Hosting.AppHost" />
    <PackageReference Include="Aspire.Hosting.RabbitMQ" />
    <PackageReference Include="Aspire.Hosting.Redis" />
    <PackageReference Include="Aspire.Hosting.PostgreSQL" />
    <PackageReference Include="Aspire.Hosting.Azure.CognitiveServices" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\Mobile.Bff.Shopping\Mobile.Bff.Shopping.csproj" />
    <ProjectReference Include="..\Basket.API\Basket.API.csproj" />
    <ProjectReference Include="..\Catalog.API\Catalog.API.csproj" />
    <ProjectReference Include="..\Identity.API\Identity.API.csproj" />
    <ProjectReference Include="..\Ordering.API\Ordering.API.csproj" />
  </ItemGroup>
```

```

<ProjectReference Include="..\OrderProcessor\OrderProcessor.csproj" />
<ProjectReference Include="..\PaymentProcessor\PaymentProcessor.csproj" />
<ProjectReference Include="..\Webhooks.API\Webhooks.API.csproj" />
<ProjectReference Include="..\WebApp\WebApp.csproj" />
<ProjectReference Include="..\WebhookClient\WebhookClient.csproj" />
</ItemGroup>
</Project>

```

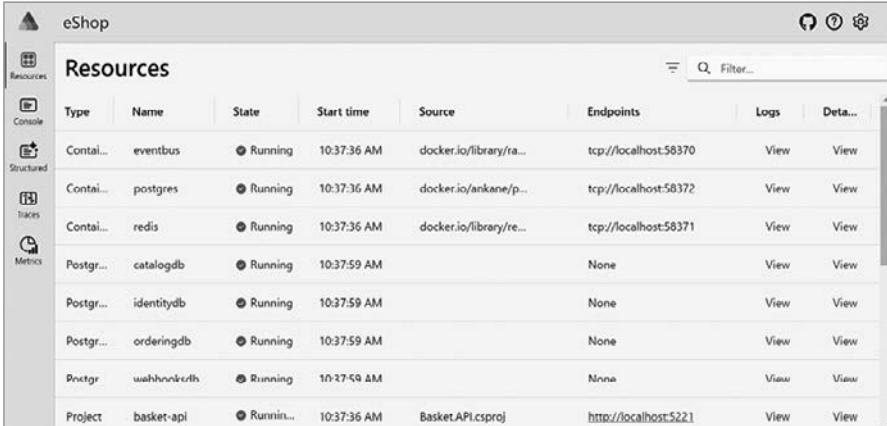
5. Выполните сборку решения.
6. Запустите приложение Docker Desktop.
7. Запустите проект **AppHost** без отладки.
8. В терминале в числе прочего вывода отобразится сообщение о запуске распределенного приложения. В нем будет указан URL-адрес информационной панели на случай, если браузер не откроется автоматически:

```

info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 8.0.0+6596fdc41a8d419876a6bf4abc17b7c66b9ef63a
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is: C:\GitHub\eshop\src\eShop.AppHost
info: Aspire.Hosting.DistributedApplication[0]
      Now listening on: https://localhost:19888
info: Aspire.Hosting.DistributedApplication[0]
      Login to the dashboard at https://localhost:19888/
login?t=42930028c8398321487a942b8aeb6cd0
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application started. Press Ctrl+C to shut down.

```

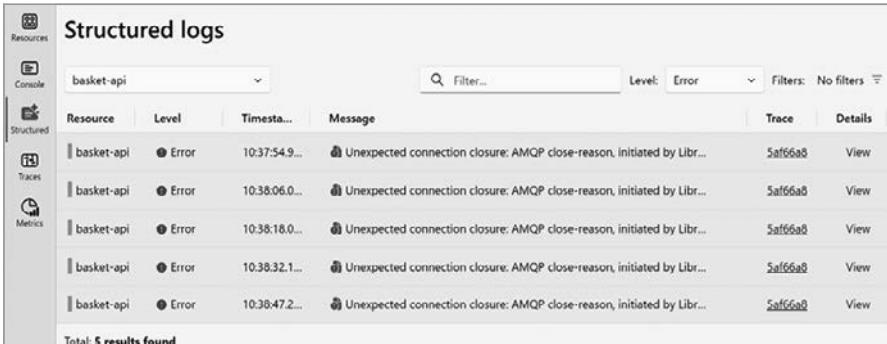
9. Дождитесь, пока на информационной панели Aspire все ресурсы не будут запущены и в столбце **State** (Состояние) появится значение **Running** (Запущено). Вы должны увидеть (рис. 16.16):
  - три контейнера: **eventbus** (RabbitMQ), **postgres** и **redis**;
  - четыре базы данных PostgreSQL: **catalogdb**, **identitydb**, **orderingdb** и **webhooksdb**;
  - десять проектов: **basket-api**, **catalog-api**, **identity-api**, **mobile-bff**, **order-processor**, **ordering-api**, **payment-processor**, **webapp**, **webhooks-api** и **webhooksclient**.
10. Сначала в журналах вы, скорее всего, увидите ошибки, потому что ресурсам вроде RabbitMQ требуется время, чтобы полностью запуститься и начать принимать запросы. Это видно на рис. 16.17.
11. В столбце **Traces** (Трассировки) на панели разработчика Aspire щелкните кнопкой мыши на значении GUID. Вы увидите подробности о серии попыток обращения к RabbitMQ: сначала запросы завершались неудачей, а примерно через минуту, когда брокер был готов, стали успешно выполняться (рис. 16.18).



The screenshot shows the 'Resources' section of the eShop application. It lists 15 resources across various service types:

Type	Name	State	Start time	Source	Endpoints	Logs	Data...
Container	eventbus	Running	10:37:36 AM	docker.io/library/fa...	tcp://localhost:58370	View	View
Container	postgres	Running	10:37:36 AM	docker.io/kanane/p...	tcp://localhost:58372	View	View
Container	redis	Running	10:37:36 AM	docker.io/library/re...	tcp://localhost:58371	View	View
PostgreSQL	catalogdb	Running	10:37:59 AM		None	View	View
PostgreSQL	identitydb	Running	10:37:59 AM		None	View	View
PostgreSQL	orderingdb	Running	10:37:59 AM		None	View	View
Project	wahhnnnkrth	Running	10:47:49 AM		None	View	View
Project	basket-api	Running	10:37:36 AM	BasketAPI.csproj	http://localhost:5221	View	View

Рис. 16.16. Семнадцать запущенных ресурсов на информационной панели решения eShop



The screenshot shows the 'Structured logs' section for the 'basket-api' service. It displays five log entries all reporting 'Error' level messages related to unexpected connection closures on the AMQP broker.

Resource	Level	Timestamp	Message	Trace	Details
basket-api	Error	10:37:54.9...	Unexpected connection closure: AMQP close-reason, initiated by Libr...	Saf66a8	View
basket-api	Error	10:38:06.0...	Unexpected connection closure: AMQP close-reason, initiated by Libr...	Saf66a8	View
basket-api	Error	10:38:18.0...	Unexpected connection closure: AMQP close-reason, initiated by Libr...	Saf66a8	View
basket-api	Error	10:38:32.1...	Unexpected connection closure: AMQP close-reason, initiated by Libr...	Saf66a8	View
basket-api	Error	10:38:47.2...	Unexpected connection closure: AMQP close-reason, initiated by Libr...	Saf66a8	View

Total: 5 results found

Рис. 16.17. Пять ошибок AMQP (RabbitMQ) от веб-сервиса basket-api

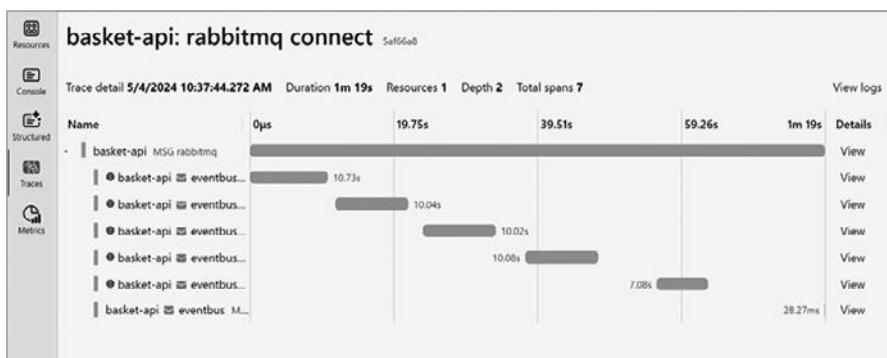
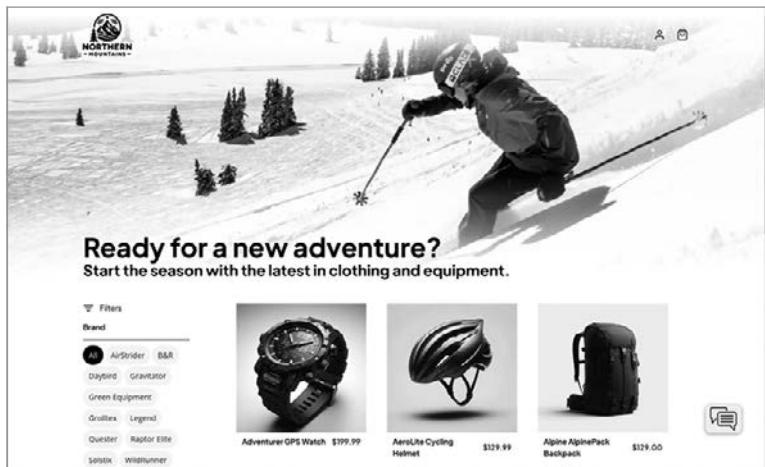


Рис. 16.18. Пять неудачных обращений к RabbitMQ в течение примерно одной минуты, пока брокер не был готов



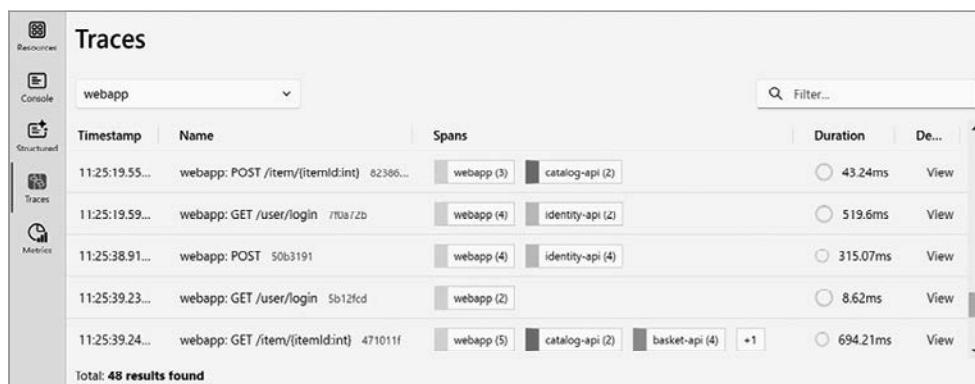
Одно из главных преимуществ Aspire в том, что вам не нужно писать специальный код для ожидания готовности таких ресурсов, как RabbitMQ или Redis. Aspire обрабатывает такие ситуации автоматически.

- Вернитесь к разделу Resources (Ресурсы). В строке webapp щелкните на ссылке <https> и посмотрите на главную страницу сайта Northern Mountains (рис. 16.19).



**Рис. 16.19.** Главная страница сайта Northern Mountains

- Откройте карточку любого товара, войдите в систему под пользователем `alice` или `bob` с паролем `Pass123$`, добавьте товар в корзину. Затем вернитесь на информационную панель Aspire, откройте последние трассировки для ресурса `webapp` и посмотрите, как используются сервисы (рис. 16.20).



**Рис. 16.20.** Трассировки на информационной панели Aspire для сайта Northern Mountains

14. В строке ресурса `basket-api` щелкните на ссылке View (Просмотреть) и обратите внимание на диаграмму последовательности вызовов между сервисами при добавлении товара в корзину (рис. 16.21).



**Рис. 16.21.** Последовательная работа ресурсов Aspire при добавлении товара в корзину

15. Закройте браузер и завершите работу распределенного приложения Aspire. Теперь посмотрим, как развернуть решение Aspire в рабочей среде.

## Развертывание с помощью Aspire

Проект AppHost поддерживает два режима выполнения.

- **Запуск** — используется в процессе разработки на локальном компьютере.
- **Публикация** — создает файл манифеста, который статически описывает модель приложения и может использоваться в сценариях развертывания.



**Важно!** Сам проект AppHost не предназначен для развертывания или запуска за пределами локальной среды разработки и тестирования.

Aspire позволяет сформировать определение манифеста, в котором описаны связи и зависимости между ресурсами решения. На основе этого манифеста инструменты могут дополнительно обрабатывать модель и использовать ее для сборки и развертывания.

С помощью манифеста можно развернуть решение Aspire в Azure через сервис Azure Container Apps — быстро и удобно. Aspire тесно интегрирован с интерфейсом Azure Developer CLI, что позволяет в один шаг развернуть как само приложение, так и все необходимые ресурсы в облаке. Кроме того, Azure Developer CLI может на основе манифеста создать файл Bicep, который затем можно использовать для аудита инфраструктуры или настройки процессов автоматизированного развертывания.



Висер — это предметно-ориентированный язык (Domain-Specific Language, DSL), в котором используется декларативный синтаксис для описания ресурсов Azure при развертывании.

Сегодня Aspire лучше всего подходит для развертывания на платформе Azure, но стек открыт и для других систем развертывания.

В итоге работа с Aspire приводит к созданию стандартных приложений и конфигураций .NET, которые можно развернуть в любой облачной среде. Поскольку Aspire ориентирован на контейнеризацию, встроенные контейнерные сборки в .NET SDK оказываются очень удобным инструментом для подготовки приложений к публикации в контейнерах.

Хотя сам стек Aspire напрямую не предоставляет прямого механизма развертывания приложений на конечные платформы, созданная вами модель приложения содержит всю необходимую информацию о зависимостях, конфигурациях и связях между ресурсами распределенного решения.



**В копилку знаний:** подробнее о развертывании приложений Aspire в Azure Container Apps можно узнать на странице [devblogs.microsoft.com/dotnet/how-to-deploy-dotnet-aspire-apps-to-azure-container-apps/](https://devblogs.microsoft.com/dotnet/how-to-deploy-dotnet-aspire-apps-to-azure-container-apps/).

Еще один вариант развертывания — инструмент Aspir8, который умеет генерировать YAML-манифесты для проектов .NET Aspire AppHost. Однако стоит учитывать, что Aspir8 все еще находится в стадии предварительной версии и для его использования нужно устанавливать предварительные сборки. Подробная информация о проекте доступна на сайте [prom3theu5.github.io/aspirational-manifests/getting-started.html](https://prom3theu5.github.io/aspirational-manifests/getting-started.html). Репозиторий Aspir8 размещен на сайте GitHub: [github.com/prom3theu5/aspirational-manifests](https://github.com/prom3theu5/aspirational-manifests).

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

## Упражнение 16.1. Дополнительные онлайн-материалы

С официальными материалами Microsoft можно ознакомиться по следующим ссылкам.

- **Анонсы:** [devblogs.microsoft.com/dotnet/introducing-dotnet-aspire-simplifying-cloud-native-development-with-dotnet-8/](https://devblogs.microsoft.com/dotnet/introducing-dotnet-aspire-simplifying-cloud-native-development-with-dotnet-8/).
- **Общедоступные релизы:** [devblogs.microsoft.com/dotnet/dotnet-aspire-general-availability/](https://devblogs.microsoft.com/dotnet/dotnet-aspire-general-availability/).
- **Документация:** [learn.microsoft.com/ru-ru/dotnet/aspire/get-started/aspire-overview](https://learn.microsoft.com/ru-ru/dotnet/aspire/get-started/aspire-overview).
- **Примеры кода:** [github.com/dotnet/aspire-samples](https://github.com/dotnet/aspire-samples).

Вы также можете найти полезные материалы об Aspire на сторонних сайтах.

- **Информационная панель:** [anthony simmon.com/dotnet-aspire-dashboard-best-tool-visualize-opentelemetry-local-dev/](https://anthonysimmon.com/dotnet-aspire-dashboard-best-tool-visualize-opentelemetry-local-dev/).
- **Часто задаваемые вопросы:** [learn.microsoft.com/ru-ru/dotnet/aspire/reference/aspire-faq](https://learn.microsoft.com/ru-ru/dotnet/aspire/reference/aspire-faq).

В книге мы намеренно избегаем привязки к ресурсам Azure или AWS, так как их использование связано с дополнительными затратами. Тем не менее полезно знать, что для Aspire существуют специальные методы настройки, ориентированные на облачные платформы. Пакеты облачного хостинга представлены в табл. 16.7.

**Таблица 16.7.** Пакеты облачного хостинга для Azure и AWS

Пакет	Ссылка
Azure Hosting	<a href="https://www.nuget.org/packages/Aspire.Hosting.Azure/">www.nuget.org/packages/Aspire.Hosting.Azure/</a>
AWS Hosting	<a href="https://www.nuget.org/packages/Aspire.Hosting.AWS/">www.nuget.org/packages/Aspire.Hosting.AWS/</a>



Подробнее о Dapr можно узнать из книги Practical Microservices with Dapr and .NET Давида Бедина.

## Упражнение 16.2. Практические задания

Для выполнения задач программирования, описанных в этой главе, вместо Docker можно использовать Podman.

Чтобы сделать это в Aspire, присвойте переменной среды `DOTNET_ASPIRE_CONTAINER_RUNTIME` значение `podman`. В PowerShell это можно сделать следующим образом:

```
$env:DOTNET_ASPIRE_CONTAINER_RUNTIME = "podman"
```

Много полезных статей и практических руководств по работе с Aspire вы найдете на сайте [aspireify.net](http://aspireify.net), созданном Джейфом Фритцем (известным также как @csharpfritz).

## Упражнение 16.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. Какие три типа ресурсов используются в Aspire?
2. Какую функцию выполняет проект `AppHost` в решении Aspire?
3. Какова роль проекта `ServiceDefaults` в решении Aspire?
4. Для чего используется метод `AddProject`?
5. Для чего используется метод `WithReference`?
6. Какие сведения отображаются на информационной панели разработчика Aspire?
7. Какие преимущества дает использование пакета компонента Aspire вместо обычного NuGet-пакета, например для Redis?
8. Чем Aspire отличается от решений вроде Dapr и Orleans?
9. Какие контейнерные технологии поддерживаются в Aspire?
10. Как проверить, что для доступа к базам данных, таким как PostgreSQL, используется надежный пароль?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 16.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-16---cloud-native-development-using-net-aspire](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-16---cloud-native-development-using-net-aspire).

## Резюме

В этой главе вы узнали:

- о ключевых концепциях Aspire;
- как создавать новые решения Aspire;
- как добавлять Aspire к существующим решениям;
- как развертывать решения Aspire.

В следующей главе мы перейдем к изучению паттернов и принципов проектирования.

# 17

## Паттерны и принципы проектирования

В этой главе речь идет о паттернах проектирования и принципах, помогающих находить проверенные решения для типичных задач. Благодаря таким рекомендациям разработчикам не приходится изобретать велосипед — это экономит время и силы, а также способствует созданию кода, который легче повторно использовать и сопровождать. Это преимущество касается всех подходов, описанных в этой главе.

Когда будете читать про различные советы и принципы, возможно, кое-что покажется очевидным — вроде банального «здравого смысла». И все же на практике здравый смысл встречается куда реже, чем хотелось бы. Поэтому полезно иметь под рукой короткий список устойчивых и понятных рекомендаций.

К тому же знание всех терминов, аббревиатур и концепций из этой главы поможет вам чувствовать себя увереннее — например, на собеседовании не придется мямлить: «Ой, а я о таком не слышал». Паттерны и принципы формируют общий язык, на котором говорят разработчики. Когда кто-то из команды упоминает определенный паттерн или принцип, остальные сразу понимают, какой подход он имеет в виду. Такой общий словарь упрощает совместную работу и помогает избежать недоразумений.

Кроме того, паттерны и принципы проектирования поддерживают единый стиль между разными проектами. Поэтому переход с одного проекта на другой менее болезненный, а требования к качеству кода — более стабильны.



Как и в любом другом аспекте разработки, здесь важно сохранять баланс. Примеры передовых методов, паттерны проектирования и принципы, приведенные в этой главе, действительно полезны — но не стоит относиться к ним как к догмам. Разработка ПО требует гибкости, умения учитывать контекст и принимать практические решения. Хотя эти подходы помогают ориентироваться и принимать более обоснованные решения, ни один из них не заменяет здравого инженерного суждения. Каждый проект уникален — и это всегда нужно учитывать.

## Принципы SOLID

Один из самых известных акронимов, с которым сталкиваются разработчики .NET, — *SOLID*. Это набор из пяти принципов, каждый из которых имеет собственную расшифровку.

- Принцип единой ответственности (Single Responsibility Principle, SRP).
- Принцип открытости/закрытости (Open/Closed Principle, OCP).
- Принцип подстановки Лисков (Liskov Substitution Principle, LSP).
- Принцип разделения интерфейса (Interface Segregation Principle, ISP).
- Принцип инверсии зависимостей (Dependency Inversion Principle, DIP).

Принципы SOLID были разработаны в первую очередь для объектно-ориентированного программирования, но их идеи вполне применимы и в других стилях разработки — например, в функциональной или процедурной парадигме.



Следовать принципам сложнее, чем правилам. Правила четкие и конкретные, а принципы — более обобщенные. Чтобы действительно применять их на практике, нужно не просто запомнить формулировки, а разобраться, как адаптировать их к своему проекту и насколько строго стоит их придерживаться.

Разберем каждый из этих пяти принципов по отдельности.

### Принцип единой ответственности (SRP)

*Принцип единой ответственности (Single Responsibility Principle, SRP)* формулируется так: *у класса должна быть только одна причина для изменения*. Иными словами, класс должен выполнять лишь одну задачу — отвечать за одну конкретную область поведения. Соблюдение этого принципа помогает создавать более понятный, гибкий и удобный в сопровождении код.

Когда в один класс попадает сразу несколько обязанностей, они начинают пересекаться и мешать друг другу. Любое изменение в одной части может непредсказуемо повлиять на другую. Это усложняет модульное тестирование, увеличивает риск ошибок и затрудняет развитие кода. Если разработчик не стремится осознанно следовать SRP, то такие проблемы могут долго оставаться незаметными — до тех пор, пока не начнут мешать работе всерьез.

### Пример соблюдения SRP

Начнем с простого примера, в котором принцип единой ответственности соблюден. Здесь есть два класса: один представляет данные профиля пользователя, другой — управляет их сохранением.

```

public class UserProfile
{
    public string UserName { get; set; }
    public string Email { get; set; }
}

public class UserProfileManager
{
    public void SaveUserProfile(UserProfile user)
    {
        // Сохраняем профиль пользователя в базе данных
    }
}

```

В этом примере `UserProfile` – это простой класс, содержащий информацию о пользователе. Класс `UserProfileManager` отвечает только за сохранение объектов класса `UserProfile` в базе данных. Каждый класс сосредоточен на своей задаче, и в этом заключается суть SRP: одна причина для изменения, одна зона ответственности.

## Пример нарушения SRP

А теперь посмотрим на пример, в котором SRP нарушается: здесь один класс выполняет сразу несколько обязанностей:

```

public class UserOperations
{
    public void SaveUser(string userName, string email)
    {
        // Сохраняем пользователя в базе данных
    }

    public void SendEmail(string email, string message)
    {
        // Отправляем письмо пользователю
    }
}

```

Класс `UserOperations` управляет как сохранением данных, так и отправкой уведомлений по электронной почте. У него две причины для изменений: если нужно изменить способ хранения данных или процесс рассылки писем. Этот код нарушает принцип SRP, поскольку класс берет на себя слишком многое.

## Рефакторинг с соблюдением SRP

Чтобы привести предыдущий пример в соответствие с принципом единой ответственности, нужно разделить обязанности между двумя отдельными классами. Каждый из них будет отвечать за свою, четко определенную задачу:

```

public class UserSaver
{
    public void SaveUser(string userName, string email)
}

```

```

    {
        // Сохраняем пользователя в базе данных
    }
}

public class EmailSender
{
    public void SendEmail(string email, string message)
    {
        // Отправляем письмо пользователю
    }
}

```

Благодаря такому разделению код становится проще сопровождать, расширять и тестировать. Каждый класс отвечает за одну конкретную часть поведения, что и отражает суть принципа SRP.

## Распространенные ошибки SRP

Одна из самых типичных ошибок, связанных с SRP, — это перегрузка класса слишком большим количеством обязанностей. Чаще всего такое происходит не из-за халатности, а из-за недопонимания: что именно считается «одной» ответственностью? Разработчики склонны воспринимать ее слишком широко — например, как «обработка пользовательских данных». В итоге в одном классе оказываются методы, которые на самом деле отвечают за совершенно разные вещи.

Иногда SRP нарушается не сразу, а спустя долгое время после написания кода. Давление сроков, стремление быстро решить проблему — и вот появляется соблазн добавить еще один метод в уже существующий класс. Ведь «там уже есть похожие методы, пусть будет и этот». Такой подход, основанный на удобстве, постепенно приводит к тому, что классы разрастаются и становятся все сложнее в сопровождении.

Когда мы проектируем класс, трудно заранее предсказать, как именно изменятся требования. И хотя сейчас может казаться, что все соответствует SRP, со временем класс может начать меняться по нескольким причинам сразу — и тогда принцип уже будет нарушен, даже если он соблюдался изначально.

Чтобы не попасть в такую ситуацию, стоит регулярно пересматривать код и выполнять небольшой рефакторинг с оглядкой на SRP. Вопрос вроде «Относится ли эта часть функциональности к основной задаче класса?» помогает держать фокус и не допустить лишнего.

Стоит также использовать практики, которые дают свежий взгляд — такие как *код-ревью* или *парное программирование*. Нередко именно другой разработчик замечает проблему, которую автор кода мог не распознать. Удивительно, но факт: мы лучше видим нарушения SRP в чужом коде, чем в своем!

## Выводы по SRP

Основные выводы по SRP.

- **Соблюдайте SRP**, стремясь к тому, чтобы каждый класс отвечал за одну четкую задачу. Такой подход делает код более модульным, гибким и понятным. Но не уходите в крайности: если в проекте появляются классы с единственным методом — это может быть признаком переусердствования. И тут тоже нужен баланс.
- **Рефакторинг с учетом SRP** — полезная привычка. Если класс со временем разросся и стал отвечать сразу за несколько вещей, стоит подумать о том, чтобы разделить его на более простые и специализированные.
- **Проектируйте с опорой на SRP**, чтобы избежать чрезмерной связности и заложить прочную основу для будущих изменений. Лучше учитывать это с самого начала, чем потом расчищать завалы.

## Принцип открытости/закрытости (OCP)

*Принцип открытости/закрытости (Open/Closed Principle, OCP)* формулируется так: *программные сущности, такие как классы и функции, должны быть открыты для расширения, но закрыты для изменения*. Это значит, что в систему можно добавлять новую функциональность, не затрагивая уже написанный код.

Такой подход помогает строить более модульные и гибкие решения, которые проще развивать и сопровождать в будущем. Вместо того чтобы постоянно переписывать существующие компоненты, вы просто добавляете новые — не ломая то, что уже работает.

### Пример соблюдения OCP

В следующем примере показана простая система генерации отчетов, которая соответствует принципу OCP:

```
public abstract class ReportGenerator
{
    public abstract void GenerateReport();
}

public class PDFReportGenerator : ReportGenerator
{
    public override void GenerateReport()
    {
        // Создание отчета в формате PDF
    }
}
```

```
public class ExcelReportGenerator : ReportGenerator
{
    public override void GenerateReport()
    {
        // Создание отчета в формате Excel
    }
}
```

Здесь `ReportGenerator` — абстрактный класс, который задает общий контракт для создания отчетов. Классы `PDFReportGenerator` и `ExcelReportGenerator` реализуют этот контракт, расширяя поведение без изменения базового кода. Если позже понадобится добавить поддержку, скажем, Word-отчетов — достаточно создать еще один класс, унаследованный от `ReportGenerator`. Код останется расширяемым, но неизменяемым — и это как раз соответствует OCP.

## Пример нарушения OCP

А вот пример, в котором принцип OCP нарушается:

```
public class ReportGenerator
{
    public void GenerateReport(string reportType)
    {
        if (reportType == "PDF")
        {
            // Создание отчета в формате PDF
        }
        else if (reportType == "Excel")
        {
            // Создание отчета в формате Excel
        }
    }
}
```

В этом случае класс `ReportGenerator` содержит логику, определяющую тип отчета на основе параметра. И если нужно добавить поддержку нового формата — например, Word, — придется лезть внутрь метода `GenerateReport` и менять его. Это нарушает принцип открытости/закрытости: каждый раз вы модифицируете существующий код, вместо того чтобы расширять систему.

Чтобы следовать OCP, стоит применять подход, показанный в первом примере, — с базовым интерфейсом или абстрактным классом и отдельными реализациями для каждого формата.

## Распространенные ошибки OCP

Частая ошибка при нарушении принципа открытости/закрытости — это отсутствие проектирования с учетом точек расширения. Если система не использует абстракции, такие как интерфейсы или абстрактные классы, то добавление новой

функциональности почти неизбежно приводит к правке существующего кода — а это и есть прямое нарушение ОСР. Когда код не построен на абстракциях, любое новое поведение приходится «вшивать» прямо в ядро логики, увеличивая риск ошибок и усложняя сопровождение.

Еще одна ошибка — недостаточное использование возможностей объектно-ориентированного программирования: наследования, делегирования, агрегирования. Когда разработчики не применяют эти механизмы, архитектура становится жесткой и негибкой. Любое новое поведение требует вмешательства в существующие компоненты, вместо того чтобы добавлять новое поведение через расширения.

Иногда в стремлении к эффективности и оптимизации «сразу и навсегда» код проектируется с сильной связанностью между компонентами. И хотя стремление к производительности похвально, преждевременная оптимизация часто приводит к тому, что архитектура становится негибкой — а значит, ее придется модифицировать при каждом изменении. Такой подход противоречит ОСР.

Наконец, еще одна ловушка — это гиперинженерный подход, когда разработчики стараются сделать *все* расширяемым. Это тоже ошибка. ОСР не означает, что абсолютно каждый компонент системы должен быть открыт для расширения. Принцип стоит применять прежде всего к тем частям приложения, которые действительно могут измениться в будущем. Иначе вместо гибкости получится излишняя сложность, которую никто не просил.

Чтобы избежать типичных ошибок, связанных с нарушением принципа открытости/закрытости, полезно придерживаться следующих подходов.

- **Работайте с абстракциями.** Проектируйте с опорой на интерфейсы и абстрактные классы, которые задают четкие контракты поведения. Это позволяет изменять реализацию без необходимости переписывать код, который от нее зависит.
- **Продумывайте точки расширения.** Постарайтесь заранее определить, какие части системы со временем, скорее всего, будут развиваться. Спроектируйте их так, чтобы расширять поведение можно было без вмешательства в существующий код.
- **Используйте агрегирование.** В тех случаях, когда оно обеспечивает большую гибкость, предпочтите агрегирование наследованию. Это особенно важно, если поведение компонента должно легко меняться или комбинироваться.
- **Находите баланс между гибкостью и простотой.** Не пытайтесь применять ОСР повсеместно. Сосредоточьтесь на тех частях системы, которые действительно могут меняться. Избыточная расширяемость может привести к усложнению архитектуры и затруднить ее поддержку.

Если с самого начала проектировать компоненты с учетом возможных изменений, получится система, которая будет проще в сопровождении, надежнее и гибче в долгосрочной перспективе.

## Выводы по OCP

Основные выводы по OCP.

- **Соблюдение OCP.** Стремитесь к тому, чтобы новая функциональность добавлялась через новый код, а не за счет изменения уже существующего. Такой подход делает систему стабильнее и предсказуемее при развитии.
- **Практика применения.** Используйте абстракции и полиморфизм — например, интерфейсы и абстрактные классы в C#, — чтобы строить гибкую архитектуру, готовую к росту и изменениям со временем.

Если подходить к проектированию классов и интерфейсов с учетом OCP, приложения становятся устойчивее к изменениям и легче адаптируются к новым требованиям — без необходимости переписывать уже отлаженный код.

## Принцип подстановки Лисков (LSP)

*Принцип подстановки Лисков (Liskov Substitution Principle, LSP)* был сформулирован Барбарой Лисков в 1987 году. Его суть в следующем: *объекты подкласса должны быть взаимозаменяемы с объектами суперкласса без нарушения корректности программы*.

Другими словами, если в коде используется базовый тип, то вместо него должен без проблем работать и любой его подкласс. Это позволяет строить более модульные и предсказуемые системы, где поведение остается стабильным при замене одного типа другим.

### Пример соблюдения LSP

Рассмотрим простой пример, в котором соблюдается принцип LSP:

```
public abstract class Shape
{
    public abstract double CalculateArea();
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double CalculateArea()
    {
        return Width * Height;
    }
}

public class Circle : Shape
{
```

```

public double Radius { get; set; }

public override double CalculateArea()
{
    return Math.PI * Radius * Radius;
}
}

```

В этом примере классы `Rectangle` и `Circle` наследуются от абстрактного базового класса `Shape` и реализуют метод расчета площади по-своему. Объекты `Rectangle` и `Circle` можно свободно использовать в контексте типа `Shape`, не нарушая корректности поведения. Каждый подкласс полностью сохраняет ожидания, заданные базовым классом, — это и есть суть LSP.

## Пример нарушения LSP

А вот случай, где принцип LSP нарушается:

```

public class Rectangle
{
    public virtual double Width { get; set; }
    public virtual double Height { get; set; }

    public double CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : Rectangle
{
    public override double Width
    {
        get => base.Width;
        set => base.Width = base.Height = value;
    }

    public override double Height
    {
        get => base.Height;
        set => base.Height = base.Width = value;
    }
}

```

На первый взгляд кажется логичным, что `Square` — это разновидность `Rectangle`. Но квадрат требует равенства ширины (`Width`) и высоты (`Height`), поэтому при изменении одного свойства должно автоматически меняться и другое. Такое поведение нарушает ожидания: код, работающий с `Rectangle`, предполагает, что ширину и высоту можно задавать независимо. Подстановка `Square` в таком контексте приводит к неожиданным результатам и нарушает принцип Лисков.

## Рефакторинг с соблюдением LSP

Чтобы не нарушать LSP, нужно отказаться от наследования в подобных случаях и реализовать классы `Rectangle` и `Square` независимо друг от друга, но с общим базовым типом:

```
public abstract class Shape
{
    public abstract double CalculateArea();
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public override double CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : Shape
{
    public double SideLength { get; set; }

    public override double CalculateArea()
    {
        return SideLength * SideLength;
    }
}
```

Теперь оба класса реализуют один и тот же интерфейс (`CalculateArea`), но их поведение не зависит от наследования. Это позволяет использовать любой из них там, где требуется `Shape`, не нарушая корректности программы. Такой подход соответствует LSP и делает архитектуру чище и предсказуемее.

## Распространенные ошибки LSP

Наиболее частая ошибка при нарушении принципа подстановки Лисков — это создание подклассов, поведение которых не соответствует ожиданиям, заданным в базовом классе. В результате при замене базового типа на производный программа может начать вести себя иначе — и это нарушает LSP.

Иногда разработчики переопределяют методы в подклассе так, что они ведут себя иначе, чем предполагает контракт базового класса. Например, метод может начать выбрасывать новые исключения, не предусмотренные исходной реализацией, либо его поведение существенно меняется: появляются побочные эффекты или меняется логика, на которую полагались пользователи базового класса.

Другая частая ошибка — ужесточение предварительных условий. Допустим, метод базового класса принимает любую непустую строку. Если в переопре-

деленной версии метод начинает требовать, чтобы строка была не короче пяти символов, он становится несовместимым с базовым — и нарушает принцип подстановки. Такой подкласс уже нельзя безопасно подставить туда, где ожидается базовый тип.

Как избежать типичных ошибок, связанных с LSP?

- **Понимайте контракт базового класса.** Прежде чем расширять класс, важно разобраться, какие обязательства он накладывает: как работают его методы, какие у них предварительные и последующие условия, какие инварианты должны сохраняться. Один из надежных способов лучше это понять — попробовать задокументировать метод. Чем точнее вы его опишете, тем яснее станет, чего от него ожидают.
- **Проектируйте с учетом подстановки.** Создавая подкласс, всегда задавайтесь вопросом: можно ли безопасно использовать его вместо базового? Если такая замена нарушает логику программы, это явный сигнал, что что-то пошло не так.
- **Агрегируйте, если наследование неуместно.** Не всегда стоит тянуть за уши наследование. Иногда проще и надежнее собрать объект из нужных компонентов, чем пытаться встроить в иерархию то, что в нее плохо вписывается.
- **Проверяйте подстановку в тестах.** Автоматизированные тесты, которые проверяют, можно ли подставить подкласс вместо базового без ошибок и неожиданного поведения, — хороший способ выявить нарушения LSP до того, как они обернутся проблемами в продакшене.

Если учитывать LSP на этапе проектирования и избегать этих ловушек, можно построить системы, которые легче расширять, сопровождать и которым проще доверять.

## LSP в контексте .NET

Классы потоков в .NET — отличный пример принципа подстановки Лисков в действии. В базовой библиотеке (BCL) есть целое семейство потоков для чтения и записи: из памяти, файлов, сети и других источников. Все они наследуются от общего базового типа `Stream` или реализуют соответствующий интерфейс. Благодаря этому можно, например, заменить один поток на другой, не меняя логику работы остальной программы.

Теперь представим, что разработчик создает свой `CustomStream`, который наследует `Stream`, но ведет себя иначе. Допустим, метод `Read` в этом классе игнорирует параметры `count` или неожиданно преобразует данные. Такой поток технически совместим с базовым типом, но на практике — нарушает его контракт. В результате код, который ожидает стандартное поведение `Stream`, может начать работать некорректно. Такой случай — классический пример нарушения LSP: формально все выглядит правильно, но замена приводит к изменению поведения, и программа больше не может полагаться на базовый интерфейс.

## Выводы по LSP

Основные выводы по LSP.

- **LSP способствует надежному проектированию.** Когда классы можно безопасно подставлять друг вместо друга, система становится гибче и легче в сопровождении.
- **Не переоценивайте наследование.** Отношение «является» (*is-a*) не всегда означает, что наследование уместно. Если поведение подкласса нарушает ожидания, заложенные в базовом классе, это сигнал, что стоит пересмотреть архитектуру.
- **Используйте LSP как инструмент анализа.** Принцип подстановки Лисков – хороший ориентир при оценке иерархий. Он помогает выявить слабые места в дизайне, где код становится хрупким и склонным к ошибкам при расширении.

## Принцип разделения интерфейса (ISP)

*Принцип разделения интерфейса (Interface Segregation Principle, ISP)* гласит: *клиент не должен зависеть от методов, которые он не использует*. Вместо одного большого интерфейса, охватывающего все подряд, стоит разбивать его на более узкие и целенаправленные. Такой подход делает систему удобнее в сопровождении и снижает связанность между компонентами.

### Пример соблюдения ISP

Представим систему, в которой устройства могут печатать, сканировать и отправлять документы по факсу. Вместо того чтобы собирать все функции в один интерфейс, мы разделим их на более узкие:

```
public interface IPrinter
{
    void Print(Document d);
}

public interface IScanner
{
    void Scan(Document d);
}

public interface IFax
{
    void Fax(Document d);
}

public class Document
{}
```

```

public class MultiFunctionPrinter : IPrинтер, IScanner, IFax
{
    public void Print(Document d)
    {
        // Реализация
    }

    public void Scan(Document d)
    {
        // Реализация
    }

    public void Fax(Document d)
    {
        // Реализация
    }
}

```

Разделив функциональность на отдельные интерфейсы (`IPrinter`, `IScanner`, `IFax`), мы позволяем клиентскому коду взаимодействовать *только с теми возможностями, которые ему реально нужны*. Например, если компоненту требуется только печатать — он зависит только от `IPrinter`, а не от всего комплекса функций.

## Пример нарушения ISP

Теперь взглянем на пример, в котором этот принцип нарушается:

```

public interface IMultiFunctionDevice
{
    void Print(Document d);
    void Scan(Document d);
    void Fax(Document d);
}

public class Document
{
}

public class MultiFunctionPrinter : IMultiFunctionDevice
{
    public void Print(Document d)
    {
        // Реализация
    }

    public void Scan(Document d)
    {
        // Реализация
    }

    public void Fax(Document d)
    {
        // Реализация
    }
}

```

В этом интерфейсе (`IMultiFunctionDevice`) объединены все функции устройства. Если клиенту нужна только печать, ему все равно придется зависеть от методов сканирования и факса — даже если он их не вызывает. Такой подход затрудняет поддержку, снижает гибкость и нарушает принцип ISP.

## Рефакторинг с соблюдением ISP

Чтобы устранить нарушение, достаточно разбить `IMultiFunctionDevice` на отдельные, более узкие интерфейсы — как показано в первом примере.

## Распространенные ошибки ISP

Одна из самых частых ошибок при нарушении принципа разделения интерфейса — это создание так называемых «жирных» интерфейсов. Такой интерфейс содержит больше методов, чем действительно требуется большинству клиентов.

Когда интерфейс не разделен, классы, реализующие его, вынуждены поддерживать методы, которые им на самом деле не нужны. Это приводит к загроможденным реализациям и усложняет сопровождение кода: в классе появляется логика, не связанная с его основной задачей. К тому же крупные интерфейсы усиливают связаннысть системы. Классы начинают зависеть от функциональности, которую они не используют, и любая правка в интерфейсе — даже не относящаяся к конкретному классу — может все равно повлечь за собой изменения. Система становится менее гибкой и хуже поддается расширению.

Интерфейсы изначально создаются как механизм повторного использования: они задают контракты, которые можно реализовывать разными способами. Но если интерфейс перегружен лишними методами, он теряет это качество. Классы, которым нужна только небольшая часть функциональности, могут вообще отказаться от его использования, потому что он не соответствует их задачам.

Как избежать типичных ошибок, связанных с ISP?

- **Разделяйте интерфейсы по сценарию использования.** Вместо одного обобщенного интерфейса создавайте несколько узких — таких, которые отражают реальные потребности клиента. Это соответствует принципу ISP: клиент зависит только от тех методов, которые действительно использует.
- **Анализируйте, кто и как использует интерфейс.** Если разные компоненты действуют разные группы методов, это повод задуматься о разделении интерфейса на более специфичные. Регулярный анализ поможет избежать избыточной связанности и упростит сопровождение.
- **Предпочитайте агрегирование, когда нужно собрать функциональность из разных источников.** Если классу требуется поведение из нескольких независимых областей, не стоит собирать все в один интерфейс. Гораздо надежнее агрегировать отдельные компоненты с нужной логикой.

- **Будьте готовы к рефакторингу.** По мере развития проекта интерфейсы могут устаревать или терять актуальность. Разделение интерфейсов, изменение клиентского кода, внедрение паттернов вроде Адаптер или Фасад — все это помогает привести систему в соответствие с принципом ISP и сделать ее гибче.

## ISP в контексте .NET

Иногда может показаться, что некоторые интерфейсы или классы в базовой библиотеке .NET (BCL) слишком объемные или недостаточно четко разделены по функциям. Однако это не обязательно означает нарушение принципа ISP. Напротив, такие решения чаще отражают сложность проектирования универсального фреймворка, предназначенного для работы в самых разных сценариях.

Класс `Stream` — хороший пример. Это абстрактный базовый тип, представляющий поток данных. Он включает методы для чтения, записи, поиска и других операций. При этом не все производные классы поддерживают полный набор этих возможностей. Например, `FileStream` обеспечивает позиционирование, а `NetworkStream` — нет. Для таких случаев в `Stream` предусмотрены свойства вроде `CanRead`, `CanWrite`, `CanSeek`, позволяющие проверить доступные функции. Теоретически можно было бы определить отдельные интерфейсы для потоков, поддерживающих только чтение или только запись. Такой подход, возможно, лучше соответствовал бы принципу ISP, особенно в системах, где требуется строгий контроль за поведением объекта. Но в рамках BCL это было бы менее удобно и привело бы к избыточной сложности — особенно с учетом обратной совместимости и разнообразия поддерживаемых сценариев.

В библиотеках общего назначения приходится искать баланс между строгостью принципов проектирования, удобством использования, производительностью и универсальностью. Иногда решение отступить от принципа — это осознанный компромисс, а не ошибка. Возможности для переосмысливания дизайна могут быть ограничены историей развития платформы и существующими зависимостями.

Важно понимать, что такие отклонения не обесценивают сам принцип. Наоборот — они дают возможность глубже разобраться в контексте, в котором применяются архитектурные решения. То, что в BCL не везде строго соблюден какой-либо принцип SOLID, не уменьшает его полезность — и не делает команду .NET неправой.

## Выводы по ISP

Основные выводы по ISP.

- **Снижение связанности.** Принцип разделения интерфейса помогает уменьшить связанность между компонентами, поскольку классы не зависят от методов, которые им не нужны.

- **Гибкость и удобство сопровождения.** Системы, построенные с учетом ISP, проще поддерживать и развивать: изменения в одной части кода реже затрагивают другие участки.
- **Осознанное использование интерфейсов.** В .NET и других объектно-ориентированных системах интерфейсы играют ключевую роль. Правильное их применение позволяет создавать более чистую архитектуру, модульный код и надежную основу для расширения системы.

## Принцип инверсии зависимостей (DIP)

*Принцип инверсии зависимостей (Dependency Inversion Principle, DIP)* помогает ослабить связь между высокоуровневыми и низкоуровневыми модулями путем введения слоя абстракции. Он основан на двух ключевых идеях.

- **Высокоуровневые модули не должны зависеть от низкоуровневых.** Оба типа должны зависеть от абстракций.
- **Абстракции не должны зависеть от деталей.** Напротив — детали должны зависеть от абстракций.

Следование принципу DIP помогает снизить связанность между компонентами, упростить сопровождение и сделать архитектуру более гибкой и расширяемой.

## Пример соблюдения DIP

Предположим, у нас есть высокоуровневый модуль `CustomerService`, отвечающий за бизнес-логику, и низкоуровневый модуль `CustomerRepository`, занимающийся доступом к данным:

```
public interface ICustomerRepository
{
    void Add(Customer customer);
    // Другие методы доступа к данным
}

public class CustomerRepository : ICustomerRepository
{
    public void Add(Customer customer)
    {
        // Детали реализации для добавления клиента
    }
}

public class CustomerService
{
    private readonly ICustomerRepository _customerRepository;

    public CustomerService(ICustomerRepository customerRepository)
    {
        _customerRepository = customerRepository;
    }
}
```

```

public void RegisterCustomer(Customer customer)
{
    _customerRepository.Add(customer);
    // Дополнительные операции высокого уровня
}
}

```

В этом примере и `CustomerService`, и `CustomerRepository` зависят от интерфейса `ICustomerRepository`. Благодаря этому `CustomerService` остается независимым от конкретной реализации и код становится гибким, тестируемым и легко расширяемым.

## Пример нарушения DIP

Теперь рассмотрим пример, где принцип нарушается:

```

public class CustomerRepository
{
    public void Add(Customer customer)
    {
        // Реализация сохранения клиента
    }
}

public class CustomerService
{
    private readonly CustomerRepository _customerRepository = new
CustomerRepository();

    public void RegisterCustomer(Customer customer)
    {
        _customerRepository.Add(customer);
        // Дополнительные действия высокого уровня
    }
}

```

Здесь высокоуровневый модуль `CustomerService` напрямую зависит от низкоуровневого модуля `CustomerRepository`, а не от абстракции. Таким образом, любое изменение в логике доступа к данным потребует правки кода в `CustomerService`. Такой подход повышает связанность и снижает гибкость архитектуры — именно то, чего DIP помогает избежать.

## Распространенные ошибки DIP

Одна из самых частых ошибок при нарушении принципа инверсии зависимостей — это жесткое создание зависимостей внутри класса. Такой подход приводит к тому, что класс зависит от конкретных реализаций, а не от абстракций, что делает его менее гибким, плохо тестируемым и тесно связанным со своими зависимостями.

Для соблюдения DIP используется внедрение зависимостей (DI) — техника, при которой зависимости передаются в класс извне, обычно через конструктор или свойства. Об этой технике подробнее говорилось в главе 10. Однако

важно помнить, что просто использовать DI-фреймворк недостаточно. Если вы по-прежнему передаете конкретные типы, а не абстракции, принцип DIP все равно остается нарушенным. Его суть — именно в использовании абстракций.

DIP часто учитывают внутри одного слоя приложения, например в бизнес-логике, но игнорируют при взаимодействии между слоями, например между пользовательским интерфейсом, бизнес-логикой и хранилищем данных. Чтобы обеспечить настоящее разделение и гибкость, каждый слой должен зависеть от абстракций, а не от конкретных реализаций соседнего слоя.

При применении DIP важно соблюдать баланс. Недостаток абстракций оставляет связанность слишком высокой. Но и чрезмерное увлечение абстрагированием может навредить: если в проекте появляется множество избыточных интерфейсов и абстрактных классов, структура становится громоздкой и теряет свою прозрачность. Вместо гибкости — сложность ради сложности.

Как избежать типичных ошибок, связанных с DIP?

- **Работайте с абстракциями.** Используйте интерфейсы или абстрактные классы вместо конкретных типов — и во внешних API, и во внутренних компонентах. Это основа гибкой архитектуры, соответствующей DIP.
- **Применяйте DIP на всех уровнях.** Не ограничивайтесь только внутренними слоями. DIP должен соблюдаться и между слоями приложения — например, между интерфейсом и бизнес-логикой или между бизнес-логикой и хранилищем данных.
- **Используйте DI-фреймворки осознанно.** Инструменты внедрения зависимостей полезны, но важно помнить: они не соблюдают DIP за вас. Внедряйте абстракции, а не конкретные реализации.
- **Находите разумный уровень абстракции.** Хорошая абстракция упрощает архитектуру и снижает связанность. Но если абстракций становится слишком много — система может усложниться без пользы. Важно находить баланс между гибкостью и ясностью.

## Выводы по DIP

Основные выводы по DIP.

- **Снижение связанности через абстракции.** DIP подчеркивает важность использования интерфейсов и абстрактных классов для отделения бизнес-логики от деталей реализации. Это позволяет строить модули, слабо зависящие друг от друга.
- **Гибкость и удобство сопровождения.** При соблюдении DIP изменения в низкоуровневых компонентах (например, замена базы данных) затрагивают минимум кода в высокоуровневых модулях, а часто не требуют изменений вовсе.
- **Улучшение тестируемости.** Благодаря зависимости от абстракций, а не от конкретных реализаций становится проще подменять компоненты в тестах. Это облегчает написание модульных и интеграционных тестов.

Эффективное применение DIP и других принципов SOLID в проектах .NET помогает создавать чистую модульную архитектуру, которую легче развивать, сопровождать и тестировать.

## Паттерны проектирования

Паттерны проектирования — это повторяемые решения типовых задач, с которыми разработчики сталкиваются при создании ПО. Это не готовые блоки кода, которые можно просто вставить в проект, а скорее рекомендации, которые помогают принять обоснованные архитектурные решения в определенном контексте.

Паттерны проектирования традиционно делят на три большие категории — порождающие, структурные и поведенческие.

- **Порождающие паттерны** (creational patterns) помогают управлять процессом создания объектов. Прямое использование `new` в коде может привести к излишней связанности и усложнению, особенно когда объект сложен или его состав зависит от контекста. Порождающие паттерны берут на себя заботу о создании экземпляров, делая этот процесс более контролируемым и гибким. Примеры таких паттернов: *Одиночка*, *Фабричный метод*, *Абстрактная фабрика*, *Строитель*, *Прототип*.
- **Структурные паттерны** (structural patterns) фокусируются на организации взаимосвязей между классами и объектами. Они помогают строить гибкую архитектуру, в которой изменения в одной части системы не требуют каскадных изменений в других. Среди наиболее известных: *Адаптер*, *Мост*, *Компоновщик*, *Декоратор*, *Фасад*, *Приспособленец*, *Заместитель*.
- **Поведенческие паттерны** (behavioral patterns) описывают шаблоны взаимодействия между объектами. Их цель — сделать связи между компонентами более понятными и управляемыми, особенно когда речь идет о сложной логике. К этой группе относятся паттерны *Наблюдатель*, *Посредник*, *Итератор*, *Стратегия*, *Команда*, *Хранитель*, *Состояние*, *Посетитель*, *Цепочка обязанностей*, *Шаблонный метод*.

Использование паттернов может ускорить разработку, поскольку они основаны на проверенных практиках. Такие подходы помогают не только решать распространенные задачи, но и заранее учитывать проблемы, которые могут проявиться на поздних этапах. Применение популярных паттернов снижает риск мелких ошибок, которые могут привести к более серьезным последствиям, и делает код понятнее для других разработчиков — особенно тех, кто знаком с этими концепциями.



**Дельный совет:** не стоит применять паттерны формально или «на всякий случай». Паттерн — это ответ на проблему, а не оправдание для ее поиска. Используйте их осознанно: избыточное применение может только усложнить код и затруднить его сопровождение.

Прежде чем перейти к подробному разбору, коротко ознакомимся с наиболее распространенными паттернами проектирования, а также примерами их использования — как в базовой библиотеке .NET, так и в ASP.NET Core и популярных сторонних библиотеках (табл. 17.1–17.3).

**Таблица 17.1.** Порождающие паттерны проектирования и их применение в .NET

Паттерн проектирования	Краткое описание	Примеры в .NET (BCL / ASP.NET Core / сторонние библиотеки)
<b>Одиночка</b>	Гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему	SqlConnection (BCL, для пула соединений), ILogger (в ASP.NET Core, при регистрации как синглтон)
<b>Фабричный метод</b>	Задает общий интерфейс для создания объекта, оставляя конкретный выбор класса подклассам	WebClient (BCL), HttpClientFactory (ASP.NET Core)
<b>Абстрактная фабрика</b>	Позволяет создавать семейства связанных объектов без привязки к конкретным реализациям	DbProviderFactory (BCL), ILogerFactory (ASP.NET Core)
<b>Строитель</b>	Управляет пошаговым созданием сложного объекта, отделяя процесс конструирования от результата	StringBuilder (BCL), WebHostBuilder (ASP.NET Core)
<b>Прототип</b>	Создает объекты путем копирования существующих	ICloneable (BCL)

**Таблица 17.2.** Структурные паттерны проектирования и их применение в .NET

Паттерн проектирования	Краткое описание	Примеры в .NET (BCL / ASP.NET Core / сторонние библиотеки)
<b>Адаптер</b>	Позволяет несовместимым интерфейсам работать вместе. Использует обертку для преобразования одного интерфейса в другой	StreamAdapter (BCL), ORM-адAPTERЫ (например, Dapper)
<b>Мост</b>	Отделяет абстракцию от ее реализации, позволяя развивать их независимо друг от друга	DbContext (BCL), ILogger (ASP.NET Core)
<b>Компоновщик</b>	Объединяет объекты в древовидные структуры, позволяя работать с отдельными элементами и группами единообразно	DirectoryInfo (BCL), IApplicationBuilder (ASP.NET Core)

Паттерн проектирования	Краткое описание	Примеры в .NET (BCL / ASP.NET Core / сторонние библиотеки)
<b>Декоратор</b>	Добавляет поведениециальному объекту без изменения других объектов того же класса	Stream (BCL, например, FileStream, MemoryStream), middleware в ASP.NET Core
<b>Фасад</b>	Предоставляет упрощенный интерфейс к сложной системе	HttpClient (BCL), WebClient (BCL)
<b>Приспособленец</b>	Снижает потребление ресурсов путем повторного использования схожих объектов	String.Intern (BCL)
<b>Заместитель</b>	Контролирует доступ к другому объекту, подменяя его или добавляя функциональность	WCF-прокси (BCL), IActionResult в ASP.NET Core

**Таблица 17.3.** Поведенческие паттерны проектирования и их применение в .NET

Паттерн проектирования	Краткое описание	Примеры в .NET (BCL / ASP.NET Core / сторонние библиотеки)
<b>Наблюдатель</b>	Устанавливает зависимость между объектами, чтобы при изменении одного остальные автоматически получали уведомление	IObservable<T> / IObserver<T> (BCL), INotifyPropertyChanged (BCL)
<b>Посредник</b>	Уменьшает связанность между взаимодействующими объектами, перенаправляя их взаимодействие через посредника	IMediator в MediatR (сторонний)
<b>Итератор</b>	Обеспечивает доступ к элементам коллекции без раскрытия ее внутренней структуры	IEnumerable / IEnumerator (BCL)
<b>Стратегия</b>	Инкапсулирует алгоритмы и делает их взаимозаменяемыми, позволяя выбирать поведение во время выполнения	IComparer / IComparable (BCL)
<b>Команда</b>	Представляет запрос как объект, позволяя выстраивать цепочки команд и настраивать их выполнение	ICommand в WPF (BCL), CommandHandler в MediatR
<b>Хранитель</b>	Позволяет сохранять и восстанавливать состояние объекта без нарушения инкапсуляции	DataContractSerializer (BCL)

*Продолжение ⇨*

**Таблица 17.3 (окончание)**

Паттерн проектирования	Краткое описание	Примеры в .NET (BCL / ASP.NET Core / сторонние библиотеки)
<b>Состояние</b>	Позволяет объекту менять поведение в зависимости от внутреннего состояния	StateServerMode в ASP.NET (BCL)
<b>Посетитель</b>	Отделяет алгоритмы от объектов, над которыми они выполняются, упрощая добавление новых операций	ExpressionVisitor (BCL)
<b>Цепочка обязанностей</b>	Передает запрос по цепочке обработчиков, каждый из которых может обработать его или передать дальше	DelegatingHandler в HttpClient (ASP.NET Core), middleware
<b>Шаблонный метод</b>	Задает скелет алгоритма в базовом классе, позволяя подклассам переопределять отдельные шаги	Stream (BCL, методы Read, Write), ControllerBase (ASP.NET Core)



Подробные описания всех паттернов проектирования доступны онлайн — в репозитории GitHub для этой книги по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/design-patterns/readme.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/design-patterns/readme.md). В эту книгу я добавил по одному паттерну из каждой группы — чтобы вы оценили структуру, в которой эти описания оформлены.

## Порождающие паттерны

С них чаще всего начинают знакомство с паттернами проектирования — и не случайно. Прежде чем приступить к работе, нужно создать хотя бы один объект. Порождающие паттерны помогают абстрагировать процесс создания объектов, инкапсулируя эту задачу внутри программы. Вместо того чтобы напрямую использовать `new`, эти паттерны позволяют делегировать создание экземпляров, что делает код более модульным, гибким и легче сопровождаемым. Они отделяют процесс инстанцирования от остальной логики, позволяя системе оставаться независимой от конкретных способов создания и компоновки объектов.



Полный список порождающих паттернов и их описания доступен в репозитории по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/design-patterns/creational.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/design-patterns/creational.md).

В печатной версии книги в качестве примера выбран паттерн Строитель. Он особенно актуален для проектов ASP.NET Core — а именно с ними .NET-разработчикам чаще всего приходится работать на практике.

## Паттерн Строитель

Основная идея паттерна Строитель (Builder) заключается в *отделении процесса создания сложного объекта от его представления*. Это позволяет использовать один и тот же алгоритм построения для создания разных вариантов объектов — в зависимости от того, какие именно части нужно собрать. Паттерн особенно полезен в .NET при работе с объектами, у которых множество свойств, в том числе необязательных.

Паттерн хорошо подходит, если:

- вы хотите избежать перегруженного конструктора со множеством параметров;
- объект требует сложной и пошаговой инициализации;
- нужно предоставить простой и читаемый API для создания сложных объектов.

Основные компоненты паттерна Строитель:

- **Строитель (Builder)** — определяет абстрактный интерфейс для создания отдельных частей объекта;
- **Конкретный строитель (ConcreteBuilder)** — реализует интерфейс строителя, отвечает за создание и сборку конкретных компонентов объекта;
- **Распорядитель (Director)** — управляет процессом построения, используя интерфейс строителя;
- **Продукт (Product)** — это итоговый объект, который создается с помощью строителя.

## Пример использования паттерна Строитель

Рассмотрим пример применения паттерна Строитель в .NET для создания объекта `UserProfile` с несколькими свойствами:

```
public class UserProfile
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public string Email { get; set; }
    public string Address { get; set; }
}
```

Сначала определим интерфейс строителя. Он содержит методы для пошаговой настройки объекта и метод `Build`, который возвращает готовый экземпляр:

```
public interface IUserProfileBuilder
{
    IUserProfileBuilder SetFirstName(string firstName);
    IUserProfileBuilder SetLastName(string lastName);
```

```
IUserProfileBuilder SetAge(int age);
IUserProfileBuilder SetEmail(string email);
IUserProfileBuilder SetAddress(string address);
UserProfile Build();
}
```

Затем реализуем конкретный строитель, который накапливает значения и создает объект:

```
public class UserProfileBuilder : IUserProfileBuilder
{
    private UserProfile _ userProfile = new UserProfile();

    public IUserProfileBuilder SetFirstName(string firstName)
    {
        _ userProfile.FirstName = firstName;
        return this;
    }

    public IUserProfileBuilder SetLastName(string lastName)
    {
        _ userProfile.LastName = lastName;
        return this;
    }

    public IUserProfileBuilder SetAge(int age)
    {
        _ userProfile.Age = age;
        return this;
    }

    ...

    public UserProfile Build()
    {
        return _ userProfile;
    }
}
```

Теперь этот строитель можно использовать для создания объекта `UserProfile`:

```
UserProfile userProfile = new UserProfileBuilder()
    .SetFirstName("John")
    .SetLastName("Doe")
    .SetAge(30)
    .SetEmail("john.doe@example.com")
    .SetAddress("123 Main St")
    .Build();
```

В этом примере класс `UserProfileBuilder` выступает в роли конкретного строителя, реализующего интерфейс `IUserProfileBuilder`. Он предлагает понятный и гибкий API для настройки объекта `UserProfile`, позволяя поэтапно задавать свойства. Метод `Build` завершает процесс и возвращает итоговый объект.

Паттерн Строитель широко используется в .NET, особенно когда требуется избежать перегруженных конструкторов и обеспечить создание объектов с предсказуемым, полноценно инициализированным состоянием. Он помогает сделать код чище, понятнее и менее подверженным ошибкам.

### Паттерн Строитель в ASP.NET Core и других библиотеках

Паттерн Строитель активно используется в ASP.NET Core и во многих сторонних библиотеках, с которыми работают .NET-разработчики. Особенно часто он встречается там, где требуется поэтапная настройка компонентов через Fluent API — например, при конфигурации сервисов, промежуточного ПО или приложения в целом.

В ASP.NET Core для запуска приложения применяются классы `WebHostBuilder` и `HostBuilder`. Они предоставляют гибкий механизм настройки окружения: параметры сервера, ведение журналов, регистрацию зависимостей и другие аспекты можно определять пошагово, в одном выразительном конструкторе. Паттерн Строитель также применяется при конфигурации сервисов в контейнере внедрения зависимостей. Интерфейс `IServiceCollection` поддерживает цепочку методов вроде `AddControllersWithViews` и `AddRazorPages`, позволяя регистрировать компоненты приложения в понятной и гибкой форме.

Entity Framework Core активно использует этот паттерн в методе `OnModelCreating` класса `DbContext`. Такой подход облегчает настройку сущностей, отношений и соединений с базой данных, делая процесс конфигурирования модели последовательным и предсказуемым.

Похожим образом работает и Serilog — популярная библиотека для журналирования. Она предоставляет Fluent API для пошаговой настройки журналирования, что делает конфигурацию ясной и расширяемой.

## Структурные паттерны проектирования

Структурные паттерны описывают способы организации классов и объектов в более крупные структуры, обеспечивая при этом расширяемость и повторное использование. Они помогают упростить проектирование, устанавливая четкие и понятные связи между компонентами системы. Главная цель этих паттернов — избежать излишней связности. Изменения в одной части системы не должны приводить к каскадным изменениям в других. Это способствует модульности и делает код гибче и удобнее в сопровождении.



Полный список структурных паттернов и их описания доступен на сайте [github.com/markjprice/tools-skills-net8/blob/main/docs/design-patterns/structural.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/design-patterns/structural.md).

В качестве примера структурного паттерна в этой главе выбран Адаптер, поскольку он активно используется при конфигурации промежуточного ПО в ASP.NET Core.

## Адаптер (он же паттерн Обертка)

Паттерн *Адаптер* (*Adapter*) используется для согласования интерфейсов между несовместимыми объектами. Он позволяет одному классу «добавляться» в структуру другого, даже если изначально они не предназначались для совместной работы. Это похоже на использование переходника между розетками разных стандартов — устройства могут быть несовместимы напрямую, но с помощью адаптера все работает.

Такой паттерн особенно полезен, когда нужно *интегрировать новый функционал в существующую систему*, не меняя исходный код. Вместо переработки интерфейсов или переписывания логики достаточно обернуть объект в адаптер, который будет «переводить» обращения от клиента к ожидаемой форме.

Паттерн Адаптер содержит три ключевых элемента:

- **Цель (Target)** — интерфейс, который ожидает использовать клиентский код;
- **Адаптируемый (Adaptee)** — класс с несовместимым интерфейсом, который нужно встроить;
- **Адаптер (Adapter)** — класс, реализующий интерфейс цели и преобразующий обращения клиента в вызовы адаптируемого объекта.

## Пример использования паттерна Адаптер

Представим ситуацию: вы разрабатываете систему, которая обрабатывает текстовые данные. В ней уже используется компонент, опирающийся на интерфейс `ITextProcessor`, обеспечивающий стандартную обработку текста. Но теперь вы подключаете новый модуль с расширенными возможностями анализа — он реализует другой интерфейс, `IAdvancedTextAnalytics`. Этот интерфейс несовместим с тем, что ожидает существующая система, но при этом предоставляет полезные функции, которые хотелось бы использовать.

Определяются оба интерфейса и соответствующие классы:

```
public interface ITextProcessor
{
    void ProcessText(string text);
}

public class TextProcessor : ITextProcessor
{
    public void ProcessText(string text)
    {
        // Обработка текста с помощью базового интерфейса
        WriteLine($"Processing text: {text}");
    }
}
```

```

public interface IAdvancedTextAnalytics
{
    void AnalyzeTextComplexity(string text);
    void FindKeyPhrases(string text);
}

public class AdvancedTextAnalytics : IAdvancedTextAnalytics
{
    public void AnalyzeTextComplexity(string text)
    {
        // Здесь реализован продвинутый анализ текста
        WriteLine($"Analyzing text complexity: {text}");
    }

    public void FindKeyPhrases(string text)
    {
        // Выделение ключевых фраз в расширенной логике
        WriteLine($"Finding key phrases in: {text}");
    }
}

```

Создается адаптер, который реализует интерфейс `ITextProcessor`, но внутри использует класс `AdvancedTextAnalytics`:

```

public class TextAnalyticsAdapter : ITextProcessor
{
    private readonly IAdvancedTextAnalytics _advancedTextAnalytics;

    public TextAnalyticsAdapter(IAdvancedTextAnalytics advancedTextAnalytics)
    {
        _advancedTextAnalytics = advancedTextAnalytics;
    }

    public void ProcessText(string text)
    {
        // Адаптер преобразует вызов в формат, поддерживаемый новым интерфейсом
        _advancedTextAnalytics.AnalyzeTextComplexity(text);
        _advancedTextAnalytics.FindKeyPhrases(text);
    }
}

```

И наконец, вы можете использовать этот адаптер в своем приложении, как показано в следующем фрагменте кода:

```

ITextProcessor processor = new TextProcessor();
processor.ProcessText("Hello, world!");

// Использование адаптера для доступа к расширенной аналитике
IAdvancedTextAnalytics analytics = new AdvancedTextAnalytics();
ITextProcessor advancedProcessor = new TextAnalyticsAdapter(analytics);
advancedProcessor.ProcessText("Exploring the Adapter pattern in .NET");

```

Класс `TextAnalyticsAdapter` позволяет использовать функциональность `AdvancedTextAnalytics` в контексте, где ожидается `ITextProcessor`. Такой подход сохраняет гибкость архитектуры, делает систему открытой к расширениям и соответствует принципам SOLID — в частности, принципам подстановки Лисков и открытости/закрытости.

### Пример паттерна Адаптер в ASP.NET Core

Паттерн Адаптер активно применяется в ASP.NET Core — особенно в промежуточном ПО, где требуется согласовать взаимодействие компонентов с разными интерфейсами. В современном промежуточном ПО ASP.NET Core используются методы `Invoke` или `InvokeAsync`, которые принимают объект `HttpContext` и выполняют нужную логику. Однако в старом коде или сторонних библиотеках могут встречаться компоненты с другими сигнатурами. Чтобы интегрировать такое промежуточное ПО, можно реализовать адаптер, который «переводит» обращение к новому стандарту. Так вы сохраните совместимость и сможете повторно использовать существующие решения без их переработки.

## Поведенческие паттерны проектирования

Поведенческие паттерны помогают управлять взаимодействием между объектами и классами в системе. Они не столько касаются структуры, сколько определяют поведение — то, как объекты обмениваются данными, сотрудничают и реагируют друг на друга.

Цель этих паттернов — *четко распределить ответственность между компонентами* и наладить эффективное взаимодействие без жесткой связанности. Они позволяют строить гибкие, расширяемые архитектуры, в которых объекты взаимодействуют предсказуемо и независимо друг от друга.



Полное описание поведенческих паттернов доступно в репозитории по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/design-patterns/behavioral.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/design-patterns/behavioral.md).

В качестве примера поведенческого паттерна в этой главе выбран Шаблонный метод, поскольку он активно используется в типах базовой библиотеки .NET, таких как `Stream` и `ControllerBase`.

### Паттерн Шаблонный метод

Паттерн Шаблонный метод (Template Method) задает общую структуру алгоритма в базовом классе, позволяя подклассам переопределять отдельные шаги — *не изменяя саму последовательность выполнения*. Это особенно удобно, когда несколько классов должны следовать одному сценарию, но с разными деталями реализации.

С помощью этого паттерна можно инкапсулировать неизменяемые части алгоритма в суперклассе, а для вариативных шагов предоставить абстрактные методы, которые реализуются в подклассах. Такой подход помогает сохранять логику алгоритма в одном месте, избегая дублирования и повышая читаемость.

Компоненты паттерна Шаблонный метод:

- **шаблон (Template)** — определяет метод, задающий общий скелет алгоритма. Он может вызывать как конкретные, так и абстрактные шаги, предоставляя точку расширения;
- **конкретные классы** — реализуют абстрактные методы, определяя, как выполняются конкретные шаги алгоритма в их контексте.

### Пример использования паттерна Шаблонный метод

Рассмотрим пример с кулинарными рецептами: у нас есть общий процесс приготовления блюда, но его отдельные шаги могут отличаться в зависимости от конкретного рецепта.

Сначала определяется абстрактный класс с шаблонным методом:

```
public abstract class CookingRecipe
{
    // Шаблонный метод
    public void CookMeal()
    {
        PrepareIngredients();
        Cook();
        Serve();
    }

    protected abstract void PrepareIngredients();
    protected abstract void Cook();

    // Общий метод, используемый всеми подклассами
    protected void Serve()
    {
        WriteLine("Serving the meal.");
    }
}
```

Затем реализуются конкретные подклассы для разных блюд:

```
public class PastaRecipe : CookingRecipe
{
    protected override void PrepareIngredients()
    {
        WriteLine("Preparing pasta and sauce.");
    }
}
```

```
protected override void Cook()
{
    WriteLine("Cooking pasta in boiling water.");
}

public class SaladRecipe : CookingRecipe
{
    protected override void PrepareIngredients()
    {
        WriteLine("Chopping vegetables.");
    }

    protected override void Cook()
    {
        WriteLine("Mixing vegetables with dressing.");
    }
}
```

И наконец, шаблонный метод используется в коде:

```
CookingRecipe pasta = new PastaRecipe();
pasta.CookMeal();
```

```
CookingRecipe salad = new SaladRecipe();
salad.CookMeal();
```

В этом примере класс `CookingRecipe` определяет шаблонный метод `CookMeal()`, который задает структуру процесса приготовления. Шаги `PrepareIngredients()` и `Cook()` объявлены как абстрактные и переопределяются в подклассах (`PastaRecipe` и `SaladRecipe`). Метод `Serve()` реализован в базовом классе, так как он одинаков для всех блюд.

Такой подход позволяет сохранить общую структуру алгоритма в одном месте, давая при этом подклассам свободу в реализации отдельных шагов. Паттерн Шаблонный метод хорошо подходит для случаев, когда важно сохранить последовательность операций, но детали этих операций могут варьироваться.

### Примеры паттерна Шаблонный метод в .NET

В экосистеме .NET паттерн Шаблонный метод встречается довольно часто. Классы `Task` и `Task<TResult>` используют шаблонный подход для определения базовой структуры асинхронных операций. Конкретная логика выполнения задачи может задаваться путем переопределения соответствующих методов.

Абстрактный класс `Stream` задает шаблонные методы для операций чтения, записи и поиска. Производные классы, такие как `FileStream`, `MemoryStream` и `NetworkStream`, реализуют эти шаги в зависимости от источника данных.

В ASP.NET Core классы `Controller` и `ControllerBase` предоставляют шаблонные методы для обработки HTTP-запросов. Поведение можно настраивать путем переопределения методов действий в производных контроллерах.

Класс `TagHelper`, используемый в Razor-представлениях, также реализует паттерн Шаблонный метод. Разработчики могут переопределять методы вроде `Process` или `ProcessAsync`, чтобы создавать собственные HTML-компоненты с нужным поведением.

## Принципы проектирования

Как и паттерны, принципы проектирования, описанные в этой главе, не являются жесткими правилами — это рекомендации, помогающие писать более качественный, понятный и устойчивый код. Они помогают принимать архитектурные решения, которые улучшают модульность, читаемость и сопровождение системы.

В этом разделе мы рассмотрим несколько общепринятых принципов, к которым часто обращаются разработчики.

- Не повторяйся (*Don't Repeat Yourself, DRY*).
- Делай проще, турица (*Keep It Simple, Stupid, KISS*).
- Вам это не понадобится (*You Ain't Gonna Need It, YAGNI*).
- Закон Деметры (*Law of Demeter, LoD*) — также известен как принцип наименьшего знания.
- Агрегирование вместо наследования.
- Принцип наименьшего удивления.

### DRY

Принцип *DRY – Don't Repeat Yourself (Не повторяйся)* — это фундаментальная концепция разработки ПО. Она подчеркивает важность избегания дублирования: если какая-то логика или структура кода повторяется, ее стоит выделить и использовать повторно. Это помогает делать системы более читаемыми, масштабируемыми и простыми в сопровождении.

Ключевые идеи принципа DRY.

- **Исключайте повторения.** Обращайте внимание на повторяющиеся блоки кода или похожие конструкции. Если обнаружите такие фрагменты — вынесите их в отдельный метод, класс или модуль.
- **Упрощайте сопровождение.** Благодаря DRY при необходимости внести изменение вы, скорее всего, сделаете это в одном месте, а не в десяти разных копиях одной и той же логики.
- **Повышайте читаемость.** Код без дублирования выглядит чище: основные идеи становятся заметнее, а уникальные детали каждого метода или класса не теряются в повторяющихся шаблонах.

Следование принципу DRY — основа чистого, эффективного и устойчивого кода. Он актуален не только в .NET, но и в любой другой среде программирования, где важны ясная архитектура и легкость поддержки.

## KISS

Принцип *KISS* – *Keep It Simple, Stupid* (*Делай проще, тупица*) – напоминает о том, что *простота – ключ к качественной архитектуре*. В разработке ПО он означает стремление к понятным и прямолинейным решениям без избыточной сложности. Простой код легче читать, тестировать и сопровождать, а значит, он менее подвержен ошибкам и дольше сохраняет свою жизнеспособность.

Ключевые идеи принципа KISS.

- **Выбирайте простое решение.** Избегайте преждевременных оптимизаций, избыточной абстракции и ненужных функций. Если задача может быть решена просто – решайте ее просто.
- **Качество через ясность.** Чем меньше слоев и уловок в реализации, тем надежнее работает код и тем проще его проверить.
- **Поддерживаемость кода.** Когда код понятен с первого взгляда, его проще поддерживать, дорабатывать и объяснять другим – особенно со временем.

Важно понимать, что KISS – это не призыв к примитивности, а напоминание о том, что лучшее решение – часто самое простое из тех, которые действительно работают.

## YAGNI

Принцип *YAGNI* – *You Ain't Gonna Need It* (*Вам это не понадобится*) – это напоминание разработчикам: *не реализовывайте функциональность заранее, до тех пор пока она действительно не нужна*. Этот подход особенно ценится в гибкой разработке (Agile) и помогает избежать лишней работы, неоправданной сложности и технического долга. В основе YAGNI лежит простая идея: потребности проекта проявляются со временем, и попытки угадать их заранее часто приводят к ненужной логике, которую в итоге никто не использует.

Ключевые идеи принципа YAGNI.

- **Работайте с настоящим.** Реализуйте только то, что требуется прямо сейчас. Не пишите код «на будущее», особенно если потребность в нем еще не подтверждена.
- **Минимизируйте потери.** Вложите усилия туда, где они приносят реальную пользу. Все остальное – трата времени и ресурсов.
- **Упрощайте проект.** Отказ от ненужной функциональности делает систему проще и понятнее. Это облегчает поддержку и развитие в долгосрочной перспективе.

Следование принципу YAGNI помогает сохранять фокус на ценности, избегать чрезмерной инженерии и строить системы, которые развиваются по мере необходимости, а не «на всякий случай». Это особенно актуально в .NET и других зрелых экосистемах, где гибкость и читаемость ценятся не меньше, чем производительность.

## Закон Деметры

Закон Деметры (*Law of Demeter, LoD*), также известный как *принцип наименьшего знания*, — это рекомендация ограничивать знание объекта о других объектах в системе. Суть можно выразить просто: «общайтесь только со своими ближайшими друзьями». На практике это означает, что объект должен взаимодействовать только с теми, с кем связан напрямую, — например, с собственными полями, параметрами метода или объектами, которые он сам создает.

### Пример использования закона Деметры

Представим, что у вас есть классы *Customer* и *BankAccount*. Вместо того чтобы представлять клиентскому коду прямой доступ к балансу счета, вы задаете специальный метод в классе *Customer*, тем самым соблюдая принцип закона Деметры:

```
public class BankAccount
{
    public decimal Balance { get; private set; }

    public BankAccount(decimal initialBalance)
    {
        Balance = initialBalance;
    }

    public decimal GetBalance()
    {
        return Balance;
    }
}

public class Customer
{
    private BankAccount account;

    public Customer(BankAccount account)
    {
        this.account = account;
    }

    public decimal CheckAccountBalance()
    {
        return account.GetBalance();
    }
}
```

В этом примере *Customer* взаимодействует напрямую с *BankAccount*, не углубляясь в его внутреннюю структуру. Связь остается простой и контролируемой.

Теперь рассмотрим противоположную ситуацию:

```
public class Wallet
{
    public BankAccount Account { get; set; }
}
```

```

public class Customer
{
    public Wallet Wallet { get; set; }

    public decimal GetAccountBalance()
    {
        // Прямой доступ к свойству Balance класса BankAccount нарушает закон Деметры
        return Wallet.Account.Balance;
    }
}

```

Здесь `Customer` обращается к балансу счета через цепочку `Wallet.Account.Balance`. Такая навигация увеличивает связанность: теперь `Customer` зависит не только от `Wallet`, но и от структуры `BankAccount`. Так код более хрупкий и менее удобный для сопровождения, поскольку любое изменение в промежуточных объектах может повлиять на `Customer`.

## Основные выводы по закону Деметры

Приведу несколько ключевых выводов, касающихся закона Деметры.

- **Снижение связанности.** Соблюдение закона Деметры помогает снижать связанность между компонентами и повышать независимость модулей.
- **Упрощение сопровождения.** Когда объекты взаимодействуют только с «близкими» зависимостями, изменения в удаленных частях системы реже приводят к каскадным изменениям.
- **Повышение модульности.** Закон Деметры способствует построению более модульной архитектуры, в которой компоненты взаимодействуют ограниченно и предсказуемо, — это упрощает понимание, отладку и развитие системы.

## Агрегирование вместо наследования

Агрегирование вместо наследования — это принцип, который рекомендует повторно использовать функциональность не через иерархию базовых классов, а путем включения зависимостей в виде отдельных компонентов. Такой подход способствует более гибкой и масштабируемой архитектуре с низкой связанностью. Композиция дает больше свободы: объекты можно комбинировать, заменять и расширять без жесткой привязки к общей иерархии. Это облегчает тестирование, сопровождение и развитие системы.

## Пример реализации принципа агрегирования

Допустим, в приложении есть несколько сервисов, которым требуется ведение журнала. Вместо того чтобы наследовать этот функционал, каждый сервис может получать компонент журналирования через агрегирование:

```

public interface ILogger
{
    void Log(string message);
}

```

```

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

public class FileLogger : ILogger
{
    public void Log(string message)
    {
        // Компонент отвечает за запись сообщений
    }
}

public class OrderService
{
    private readonly ILogger _logger;

    public OrderService(ILogger logger)
    {
        _logger = logger;
    }

    public void ProcessOrder()
    {
        _logger.Log("Processing order");
        // Сервис использует внедренный механизм журналирования
    }
}

```

В этом примере `OrderService` содержит ссылку на `ILogger`, что позволяет свободно выбирать реализацию — `ConsoleLogger`, `FileLogger` или любую другую — без изменения самого класса. Благодаря этому архитектура становится гибкой и адаптируемой.

## Пример нарушения принципа агрегирования

Сравните предыдущий пример с вариантом, в котором используется наследование:

```

public abstract class BaseService
{
    protected void Log(string message)
    {
        Console.WriteLine(message);
    }
}

public class OrderService : BaseService
{
    public void ProcessOrder()
    {
        Log("Processing order");
        // Логика обработки заказов
    }
}

```

В этом случае класс `OrderService` наследует функциональность журналирования от `BaseService`. Такой подход жестко связывает `OrderService` с конкретной реализацией. Если в будущем потребуется изменить механизм журналирования, это повлияет на все классы, унаследованные от `BaseService`, что усложняет сопровождение и снижает гибкость архитектуры.

## Основные выводы о принципе агрегирования вместо наследования

Рассмотрим несколько ключевых моментов, связанных с принципом агрегирования вместо наследования.

- **Гибкость.** Агрегирование позволяет выстраивать более гибкие конструкции, в которых поведение можно изменять и расширять, комбинируя различные реализации компонентов.
- **Удобство сопровождения.** При изменении логики компонента, например логгера, изменения локализуются и не затрагивают другие части системы — в отличие от наследования, где они могут распространяться по всей иерархии.
- **Повторное использование.** Компоненты, созданные через агрегирование, легче повторно использовать в разных контекстах, не создавая громоздких иерархий.

Выбор агрегирования вместо наследования помогает строить модули с четко определенной ответственностью и минимальной связанностью. Такой подход ближе к современным принципам проектирования и способствует созданию систем, которые проще понять, развивать и сопровождать.

## Принцип наименьшего удивления

Принцип наименьшего удивления (*Principle of Least Astonishment, PoLA*) предполагает, что ПО должно вести себя максимально предсказуемо — так, как этого интуитивно ожидает пользователь или разработчик. Он касается как интерфейсов, так и поведения кода или API и помогает снижать количество ошибок, путаницы и недопонимания.

### Пример применения принципа PoLA

Принцип наименьшего удивления можно проиллюстрировать на примере наименования методов. Рассмотрим метод с ясным назначением:

```
public class EmailService
{
    public void SendEmail(string toAddress, string subject, string body)
    {
        // Отправка письма по электронной почте
    }
}
```

Метод `SendEmail` — это хороший пример, соответствующий принципу наименьшего удивления: его название и параметры дают четкое представление о назначении. Разработчику не нужно заглядывать внутрь метода, чтобы понять, что он делает.

Теперь рассмотрим менее удачный вариант:

```
public class DataProcessor
{
    public void ProcessData(int mode)
    {
        if (mode == 1)
        {
            // Загрузка данных
        }
        else if (mode == 2)
        {
            // Сохранение данных
        }
    }
}
```

Метод `ProcessData` и использование им магического числа (`mode`) для определения поведения могут сбить с толку. Из сигнатуры метода неясно, к чему относится магическое число и что конкретно делает метод `ProcessData`. Это может привести к ошибкам или непониманию разработчиков, не знакомых с реализацией метода, что нарушает принцип наименьшего удивления.

Чтобы привести поведение класса `DataProcessor` в соответствие с принципом наименьшего удивления и сделать его интерфейс более предсказуемым, можно пойти двумя путями: либо заменить «магическое число» на перечисление `enum`, которое явно описывает возможные режимы работы, либо разделить универсальный метод `ProcessData` на два отдельных метода с понятными именами — например:

```
public enum DataMode
{
    Load,
    Save
}

public class DataProcessor
{
    public void LoadData()
    {
        // Загрузка данных
    }

    public void SaveData()
    {
        // Сохранение данных
    }
}
```

Методы `LoadData` и `SaveData` названы так, чтобы ясно отражать выполняемую ими задачу. Такое именование исключает двусмысленность и делает API понятным с первого взгляда. При необходимости использования режимов вместо «магических чисел» лучше определить перечисление `DataMode`, которое явно укажет на доступные варианты поведения.

## Выводы по принципу PoLA

Основные выводы по принципу наименьшего удивления (PoLA).

- **Понятные имена.** Используйте четкие и описательные имена для методов, классов и переменных — они должны сразу передавать смысл и ожидаемое поведение.
- **Избавляйтесь от магических чисел.** Заменяйте неочевидные значения на именованные константы или перечисления. Такой код читабельнее и понятнее без дополнительного контекста.
- **Документируйте поведение.** Если поведение метода неочевидно, снабдите его кратким пояснением. Хорошая документация снижает риск недопонимания и ошибок.

Соблюдение принципа наименьшего удивления помогает создавать библиотеки и приложения, которые проще понять, легче поддерживать и сложнее использовать неправильно. Благодаря этому опыт разработки в .NET становится более приятным и надежным.

Завершим эту главу кратким рассказом о распространенных алгоритмах и структурах данных.

## Алгоритмы и структуры данных

Платформа .NET предлагает богатую экосистему встроенных структур данных и алгоритмов, что избавляет от необходимости реализовывать большинство из них вручную. Однако понимание этих основ — это то, что отличает хороших разработчиков от по-настоящему сильных.

Ключевые категории алгоритмов и структур данных:

- алгоритмы сортировки;
- алгоритмы поиска;
- алгоритмы работы со структурами данных;
- алгоритмы хеширования;
- рекурсивные алгоритмы.

Эти темы часто поднимаются на собеседованиях и полезны при решении прикладных задач. Знание алгоритмов и структур данных показывает, что вы не только

разбираетесь в инструментах .NET, но и обладаете фундаментальным пониманием принципов информатики. Даже если вы в реальной работе чаще используете встроенные средства, важно уметь оценить алгоритмическую сложность, выбрать подходящий инструмент и, при необходимости, реализовать собственное решение.

При подготовке к собеседованиям не ограничивайтесь запоминанием алгоритмов — разберитесь в их логике и применении. Протестируйте их на C#, чтобы не только понять идею, но и уверенно ориентироваться в синтаксисе и возможностях стандартной библиотеки .NET.

## Алгоритмы сортировки

Алгоритмы сортировки — это хлеб с маслом (или код и кофе — кому как нравится) программиста, особенно когда дело доходит до собеседований. Знание принципов этих алгоритмов — обязательная часть арсенала любого разработчика. Среди наиболее важных:

- **быстрая сортировка (QuickSort)** — эффективный «друг», который быстро разбивает задачу на части и решает их по очереди;
- **сортировка слиянием (MergeSort)** — аккуратно делит данные на управляемые фрагменты, а затем объединяет их в отсортированную структуру;
- **пузырьковая сортировка (BubbleSort)** — самая простая для понимания, но не самая быстрая. Многократно проходит по списку, сравнивает соседние элементы и меняет их местами, пока все не станет по порядку.

## Алгоритмы поиска

Поиск — одна из самых базовых задач в программировании, и его алгоритмы тоже часто всплывают на собеседованиях. Вот два ключевых алгоритма поиска:

- **двоичный поиск (Binary Search)** — применяется к отсортированным массивам. Работает по принципу «раздели пополам»: на каждом шаге область поиска сокращается вдвое;
- **линейный поиск (Linear Search)** — подходит для неотсортированных структур. Последовательно перебирает элементы, пока не найдет нужный.

## Алгоритмы работы со структурами данных

Умение эффективно использовать такие структуры, как деревья и графы, помогает решать сложные задачи элегантно и с минимальными затратами. Наиболее важные алгоритмы в этой категории:

- **обход дерева** — включает центрированный (Inorder), прямой (Preorder) и обратный (Postorder) обходы. Эти методы применяются для навигации по иерархическим структурам;

- **графовые алгоритмы** — в их числе:
  - **алгоритм Дейкстры** — используется для поиска кратчайшего пути (вспомните, как работает навигатор в смартфоне);
  - **поиск в глубину (DFS) и поиск в ширину (BFS)** — применяются для поиска узлов графа.

## Алгоритмы хеширования

Хеширование — это способ преобразования данных в компактное значение фиксированного размера. Оно используется в самых разных контекстах: от поиска в коллекциях до обеспечения безопасности. Для разработчика важно понимать, как устроены хеш-таблицы, какие бывают подходы к обработке коллизий и как создать надежную хеш-функцию, которая дает равномерное распределение и минимизирует конфликты.



Как мы обсуждали в главе 8, не стоит самостоятельно разрабатывать криптографические алгоритмы. Это сложная и критичная область, где ошибка может стоить безопасности системы. На практике вам достаточно понимать суть хеширования и знать, какие алгоритмы применимы в том или ином сценарии. Реализовывать их вручную не требуется — разве что для тренировки или демонстрации на собеседовании.

## Рекурсивные алгоритмы

Рекурсия — это не отдельный алгоритм, а способ мышления, лежащий в основе множества решений. Она позволяет разбивать сложную задачу на более простые версии самой себя — и таким образом постепенно добираться до результата. Рекурсия полезна в задачах вроде «Ханойской башни», перестановок, обхода дерева, а иногда даже в задачах обработки файловых структур и выражений.

На собеседовании важно продемонстрировать понимание как возможностей, так и подводных камней рекурсии.

- Рекурсивные вызовы расходуют стек вызовов — это может привести к переполнению памяти.
- Во многих случаях итеративный подход работает быстрее и безопаснее, особенно когда требуется контроль над ресурсами.
- Рекурсивный код может выглядеть элегантно, но он сложнее для отладки и понимания, особенно в команде.

Если вы не уверены, что поддержите этот код и через полгода — возможно, стоит выбрать более простое решение.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 17.1. Дополнительные онлайн-материалы

Старую, но все еще актуальную статью *Discover the Design Patterns You're Already Using in the .NET Framework* Роба Пьерри можно найти на сайте Microsoft по адресу [learn.microsoft.com/ru-ru/archive/msdn-magazine/2005/july/discovering-the-design-patterns-you-re-already-using-in-net](http://learn.microsoft.com/ru-ru/archive/msdn-magazine/2005/july/discovering-the-design-patterns-you-re-already-using-in-net).

### Упражнение 17.2. Практические задания

Если вы хотите отточить свои навыки решения алгоритмических задач, обратите внимание на следующие платформы:

- LeetCode — [leetcode.com](https://leetcode.com);
- HackerRank — [www.hackerrank.com](https://www.hackerrank.com);
- CodeSignal — [codesignal.com](https://codesignal.com).

### Упражнение 17.3. Проверьте свои знания

Некоторые из этих вопросов потребуют от вас чуть больше, чем просто вспомнить материал, — нужно будет самому немного покопаться и разобраться. Я специально не даю все ответы в книге, потому что важно уметь выходить за рамки и смотреть шире.

И вот что действительно стоит запомнить: учитесь спрашивать, искать помошь и разбираться сами. Это как с рыбой — полезнее понять, как ее ловить, чем каждый раз ждать, что кто-то принесет готовую.

1. Как расшифровывается аббревиатура SOLID и какие принципы она включает?
2. Какие три группы паттернов проектирования считаются основными?
3. В чем суть паттерна Стратегия? Приведите пример из базовой библиотеки классов .NET.
4. Паттерн Одиночка в .NET часто реализуют через static-класс со свойством `Instance` и приватным конструктором. Какой способ предпочтительнее?
5. В чем заключается принцип проектирования, известный как закон Деметры?

## Упражнение 17.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-17---design-patterns-and-principles](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-17---design-patterns-and-principles).

## Резюме

В этой главе мы разобрали:

- принципы SOLID;
- паттерны проектирования;
- принципы проектирования.

А также познакомились с основами алгоритмов и структур данных, которые помогают глубже понимать внутреннюю механику приложений.

Освоение паттернов и принципов проектирования — одна из отличительных черт опытного разработчика. Эти знания показывают не только владение конкретным языком или инструментом, но и глубокое понимание построения систем, что усиливает ваши архитектурные навыки.

В следующей главе мы начнем знакомство с основами архитектуры решений и ПО.

# 18 Основы архитектуры ПО и решений

Эта глава познакомит вас с основами архитектуры ПО и решений. Вы узнаете о ключевых концепциях и подходах, которые лежат в основе построения успешных программных решений.

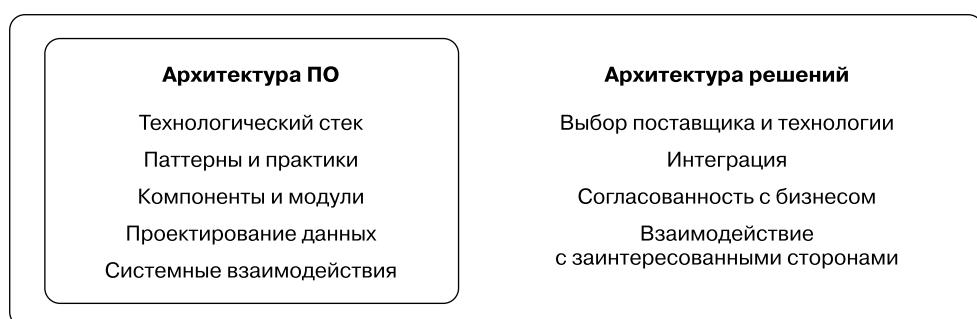
Цель главы — не углубленное изучение всех архитектур, а первичное знакомство с основными понятиями и стилями. Это поможет вам участвовать в командных обсуждениях, понимать решения архитекторов и менеджеров и лучше ориентироваться в выборе подхода к проекту.

По мере изучения вы начнете видеть общие черты разных архитектурных стилей, их особенности, а также преимущества и ограничения, которые могут повлиять на разработку реальных систем.

## Основные понятия архитектуры ПО и решений

Термины «архитектура ПО» и «архитектура решений» нередко вызывают путаницу, особенно потому, что их области ответственности частично пересекаются — обе играют важную роль в проектировании программных решений и планировании ИТ-инициатив.

На рис. 18.1 показано, чем эти два архитектурных направления различаются:



**Рис. 18.1.** ПО и архитектура

И архитектура ПО, и архитектура решений необходимы для успешной реализации .NET-проектов. Но при этом они фокусируются на разных аспектах проектирования: одна — на технической структуре приложения, другая — на интеграции решения в более широкий бизнес-контекст.

Давайте рассмотрим каждое из этих направлений подробнее, а затем сравним их в совокупности.

## Архитектура ПО

Архитектура ПО в первую очередь связана со структурой и проектированием программных решений. Ее задача — сформулировать решение, которое соответствует техническим и эксплуатационным требованиям, при этом оптимизируя такие характеристики, как производительность, безопасность и управляемость.

В фокусе архитектуры ПО — технологическая сторона системы, включая следующее.

- **Технологический стек.** Выбор подходящих языков, фреймворков и библиотек. Если вы читаете эту книгу, то, скорее всего, .NET — ключевая технология в вашем стеке.
- **Паттерны и практики.** Использование паттернов проектирования, стандартов кодирования и проверенных подходов — все это помогает обеспечить масштабируемость, надежность и удобство сопровождения системы. Многие такие практики мы рассмотрели в главе 17.
- **Компоненты и модули.** Определение структурных частей приложения и принципов их взаимодействия. Один из главных выборов здесь — между монолитной архитектурой, модульным монолитом и микросервисами.
- **Проектирование данных.** Структура баз данных, организация схем и способы передачи данных между компонентами.
- **Системные взаимодействия.** Взаимодействие приложения с внешними системами, API и сторонними сервисами.

Архитектура ПО — глубоко техническая область, требующая понимания как программирования, так и проектирования систем. Качественно продуманная архитектура создает основу, на которую может опираться команда разработки: она задает вектор, формирует структуру и помогает сохранять гибкость в процессе развития проекта.

## Архитектура решений

Архитектура решений — это более широкое и бизнес-ориентированное направление, охватывающее не только программное, но и аппаратное обеспечение, процессы, людей и организационные ограничения. Ее цель — создать устойчивое

и согласованное решение, которое эффективно отвечает конкретным бизнес-потребностям.

Архитектор решений должен уметь интегрировать различные технологии и согласовывать технические решения с целями бизнеса. Эта роль включает в себя следующее.

- **Интеграцию.** Обеспечение совместной работы программных и аппаратных компонентов на разных платформах и в разных системах.
- **Согласование с бизнесом.** Увязка архитектурных решений с бизнес-стратегией, процессами и ожиданиями.
- **Определение границ проекта.** Уточнение объема, требований, сроков и стоимости проекта.
- **Взаимодействие с заинтересованными сторонами.** Коммуникация как с техническими, так и с нетехническими участниками — перевод бизнес-задач на язык решений.
- **Выбор поставщиков и технологий.** Оценка вариантов по стоимости, надежности, совместимости и потенциалу.

Архитектура решений охватывает всю экосистему — не только код, но и окружение, в котором этот код работает. Она требует сочетания технической экспертизы и понимания бизнеса, а ее фокус — в создании ценности через устойчивые, сбалансированные решения.

Обычно архитекторы решений — это специалисты с серьезным опытом, часто более десяти лет в разработке и архитектуре. Если вы только начинаете путь в ИТ, стремление к этой роли может быть вашей долгосрочной целью или ориентиром в карьере.

А пока вы развиваетесь, можно начать с малого: попросите руководителя включать вас в архитектурные обсуждения или встречи по системным решениям за пределами вашего текущего участка. Это поможет вам расти — не только как разработчику, но и как будущему архитектору.

## Концепции архитектуры ПО

Концепции архитектуры ПО помогают разбить сложность систем на управляемые части, каждая из которых охватывает определенные аспекты проектирования, реализации и сопровождения.

В табл. 18.1 перечислены ключевые архитектурные концепции, с которыми полезно быть знакомым каждому разработчику и архитектору.

Знание и практическое применение этих концепций помогает архитекторам и разработчикам создавать надежные, масштабируемые и хорошо управляемые системы, способные эффективно решать поставленные задачи.

**Таблица 18.1.** Краткое описание архитектурных концепций

Концепция	Описание
<b>Паттерны проектирования</b>	Стандартные решения типовых задач проектирования. Представляют собой лучшие практики, используемые опытными разработчиками объектно-ориентированных систем. Это многократно используемые конструкции и способы взаимодействия объектов. Подробнее о них — в главе 17
<b>Архитектурные стили</b>	Подходы к построению архитектуры, определяемые своими особенностями. Примеры: микросервисы, монолит, многоуровневая ( <i>n-tier</i> ), событийно-ориентированная и сервис-ориентированная архитектура (SOA). У каждого стиля — свои принципы и рекомендации
<b>Атрибуты качества</b>	Нефункциональные характеристики, которые делают систему надежной и желаемой: масштабируемость, надежность, доступность, безопасность, производительность и др.
<b>Масштабируемость и производительность</b>	Подходы к построению систем, способных выдерживать рост нагрузки. Включают балансировку, кэширование, проектирование распределенных решений, оптимизацию БД
<b>DevOps</b>	Совокупность практик, объединяющих разработку (Dev) и эксплуатацию (Ops) для сокращения жизненного цикла поставки и повышения стабильности релизов. Сюда входят CI/CD, автоматизация, мониторинг
<b>Архитектура безопасности</b>	Проектирование архитектуры с учетом угроз, уязвимостей и требований безопасности. Включает проверку подлинности, авторизацию, шифрование и другие защитные механизмы (см. также главу 12)
<b>Облачные и бессерверные архитектуры</b>	Модели развертывания приложений в облачной среде. Облачные подходы обеспечивают гибкость и масштабируемость, а бессерверные — автоматическое управление ресурсами и высокой отказоустойчивостью

## Предметно-ориентированное проектирование (DDD)

Предметно-ориентированное проектирование (Domain-Driven Design, DDD) — это подход к разработке ПО, при котором особое внимание уделяется сотрудничеству между техническими специалистами и экспертами предметной области для создания моделей, отражающих реальные бизнес-процессы. DDD фокусируется на построении модели предметной области (домена), которая помогает решать сложные бизнес-задачи. Такая модель становится центральной частью архитектуры и направляет разработку системы.

Важно понимать, что DDD — это не архитектура в строгом смысле слова. Это методология, влияющая на архитектурные решения. DDD помогает выстраивать

реализацию вокруг развивающейся бизнес-модели, постепенно уточняя ее через итеративное сотрудничество с экспертами. Подход DDD способствует созданию решений, которые лучше соответствуют требованиям бизнеса, и часто приводит к архитектурам типа микросервисов, где каждый сервис отвечает за свою ограниченную область знаний в рамках бизнес-домена.

## Методологии жизненного цикла разработки ПО (SDLC)

Методологии жизненного цикла разработки ПО (Software Development Lifecycle, SDLC) описывают фазы создания ПО и порядок их выполнения. Каждая методология предлагает свой подход к прохождению жизненного цикла проекта, чтобы повысить шансы на успешную реализацию. Выбор модели SDLC напрямую влияет на архитектурные решения: разные модели предъявляют разные требования к структуре системы, порядку ее развития и способу взаимодействия команд. Каждая методология SDLC имеет свои принципы, практики, преимущества и особенности, направленные на решение различных задач в процессе разработки. От правильного выбора методологии зависит, насколько эффективно команда будет справляться с требованиями, изменениями и рисками.

Наиболее известные методологии разработки ПО кратко рассмотрены в табл. 18.2.

**Таблица 18.2.** Краткое описание методологий разработки ПО

Методология	Описание
Agile	Agile – это скорее философия, чем строгая методология. Она основана на ценностях и принципах манифеста Agile Manifesto и подчеркивает гибкость, сотрудничество с заказчиками и готовность к изменениям. Agile поддерживает итеративную разработку, при которой требования и решения развиваются совместно. Самые популярные фреймворки – Scrum и Канбан
Scrum	Подмножество Agile, дополняющее его конкретными процессами и ролями: Scrum-мастер, владелец продукта, команда разработчиков. Scrum организует работу в спринты продолжительностью от двух до четырех недель и включает регулярные встречи, обзоры и ретроспективы для постоянного улучшения
Канбан	Еще один фреймворк в рамках Agile, ориентированный на визуализацию процесса разработки с помощью канбан-доски. Помогает управлять потоком задач, выявлять узкие места и балансировать между спросом и доступными ресурсами. Все рабочие элементы отображаются визуально, что позволяет видеть текущее состояние работы

*Продолжение ↗*

**Таблица 18.2 (окончание)**

<b>Методология</b>	<b>Описание</b>
<b>Lean</b>	Основан на принципах бережливого производства, адаптированных для разработки ПО. Фокусируется на устранении потерь, усилении обучения, отсрочке принятия решений до последнего ответственного момента и ускорении поставки ценности клиенту
<b>Каскадная модель</b>	Классическая методология, предполагающая линейное выполнение этапов проекта. Каждый этап должен быть полностью завершен перед началом следующего. Модель подходит для проектов с четко определенными требованиями, но менее эффективна в динамичных условиях
<b>Экстремальное программирование (XP)</b>	Методология, направленная на повышение качества кода и скорости адаптации к изменениям требований заказчика. Поощряет частые релизы, короткие циклы разработки и регулярную обратную связь с заказчиком
<b>Rapid Application Development (RAD)</b>	Ориентирована на быструю разработку и поставку рабочих систем. Акцентируется на активном вовлечении пользователей, быстрым создании прототипов и минимальном объеме предварительного планирования

Каждая из рассмотренных методологий затрагивает разные аспекты процесса разработки — от управления проектами и взаимодействия в команде до проектирования и развертывания решений. Выбор подходящей методологии зависит от потребностей проекта, размера команды, сроков и сложности задачи. На практике лучший результат часто дает гибридный подход: сочетание элементов нескольких методологий, адаптированных к уникальным особенностям проекта и рабочему процессу команды.

## Архитектурные стили ПО

В мире разработки существует несколько архитектурных стилей, которые стали своеобразными отраслевыми стандартами благодаря надежности, масштабируемости и эффективности при решении практических задач.

Важно понимать, что архитектурные стили — это не жесткие рамки. Их можно сочетать, комбинировать и адаптировать под конкретные требования проекта. Точно так же, как в реальной архитектуре могут переплетаться готика и модерн (вспомните собор Саграда Фамилия в Барселоне), в программной архитектуре часто комбинируются подходы. Например, модульный монолит объединяет черты монолитной и модульной архитектуры, чтобы взять лучшее от обеих.

В табл. 18.3 представлены краткие описания наиболее распространенных архитектурных стилей.



Когда я жил в Ванкувере и изучал сценарное мастерство в 2005 году, мне нравилось бывать в Ванкуверской публичной библиотеке. В ее дизайне сочетаются постмодернистские и римские архитектурные стили. Здание напоминает Колизей, но использует современные материалы, создавая динамичное пространство, где историческое наследие и современность сливаются воедино. Это место даже стало фоном для сцен в ремейке «Звездного крейсера Галактика», где требовалось передать футуристическую атмосферу с отсылкой к древнеримскому политеизму.

**Таблица 18.3.** Краткое описание архитектурных стилей ПО

Архитектура	Описание
<b>Монолитная</b>	Это одноуровневое приложение, в котором пользовательский интерфейс и доступ к данным объединены в общий проект на одной платформе. Простое в разработке, развертывании и сопровождении, оно хорошо подходит для небольших систем. Однако по мере роста проекта усложняются понимание и модификация кода. Масштабирование отдельных функций требует масштабирования всего приложения, а сборка и развертывание начинают замедляться, снижая производительность
<b>Модульная монолитная</b>	Эта архитектура развивает классический монолит, сохраняя его преимущества и одновременно устранивая часть недостатков. Приложение организовано в отдельные модули или компоненты, каждый из которых фокусируется на конкретной бизнес-области и изолирован от других. Хотя система по-прежнему развертывается как единое целое, модули проектируются так, чтобы можно было масштабировать их более независимо, используя, например, многопоточность. Процесс развертывания остается таким же простым, как в традиционном монолите, но благодаря модульности улучшается время сборки и открываются возможности для более целенаправленной оптимизации
<b>Микросервисная</b>	Представьте себе сложную машину, состоящую из множества независимых компонентов, каждый из которых отвечает за свою функцию и способен работать самостоятельно. Именно так устроена микросервисная архитектура: приложение структурируется как набор небольших сервисов с четко определенными границами ответственности. Эти сервисы легко сопровождать, тестировать и развертывать независимо друг от друга. Архитектура особенно хорошо подходит для крупных, сложных приложений, требующих высокой масштабируемости и гибкости. Один из наиболее известных примеров ее успешного применения — компания Netflix, которая использовала микросервисы для масштабирования своих глобальных сервисов

*Продолжение ↗*

**Таблица 18.3 (продолжение)**

<b>Архитектура</b>	<b>Описание</b>
<b>Сервис-ориентированная архитектура (SOA)</b>	SOA напоминает план города, где разные службы (например, электроснабжение, водоснабжение, утилизация отходов) предоставляются как отдельные, но совместимые услуги. Архитектура ориентирована на повторное использование сервисов по всей организации, обеспечивает гибкость интеграции различных систем и способствует функциональной совместимости. Примером успешного внедрения SOA является Amazon: около 25 лет назад Джек Безос обязал все подразделения компании перестроить внутренние сервисы на принципы SOA. Хотя этот переход сначала сопровождался трудностями, он в итоге подготовил основу для создания AWS и перехода к более современным архитектурам, таким как микросервисы
<b>Многоуровневая (N-уровневая)</b>	Эта архитектура похожа на многоэтажное здание, где каждый этаж выполняет свою функцию — например, представление данных, бизнес-логику или доступ к данным. Многоуровневая архитектура широко использовалась в 1990-х годах, еще до появления интернета, и до сих пор остается популярной благодаря своей простоте и понятной структуре. Приложения разделяются на четко определенные слои, взаимодействующие через стабильные интерфейсы, что упрощает управление, масштабирование и развитие системы. Такая архитектура особенно хорошо подходит для корпоративных приложений, веб-приложений и мобильных бэкендов
<b>Событийно-ориентированная (EDA)</b>	Архитектура EDA строится вокруг событий: система производит, обнаруживает и реагирует на события в реальном времени. Это похоже на сеть умного дома, где каждый датчик инициирует события, а исполнительные устройства реагируют на них. Событийно-ориентированная архитектура отлично подходит для асинхронных систем с переменным или непредсказуемым трафиком и для приложений, где требуются высокая масштабируемость и быстрая реакция
<b>Бессерверная</b>	В бессерверной архитектуре разработчики сосредоточиваются исключительно на коде бизнес-логики, а управление серверами и инфраструктурой берет на себя облачный провайдер (например, AWS Lambda). Такой подход снижает операционные издержки, повышает масштабируемость и ускоряет разработку. Бессерверные архитектуры особенно эффективны для специфических рабочих нагрузок и динамически изменяющихся приложений
<b>Чистая</b>	Чистая архитектура направлена на создание систем, которые легко поддерживать, тестировать и масштабировать. Она требует четкого разделения ответственности между слоями приложения, с особым вниманием к изоляции бизнес-логики

<b>Архитектура</b>	<b>Описание</b>
	от внешних зависимостей. Хотя для небольших проектов такая структура может показаться избыточной, ее преимущества становятся все более очевидными по мере роста масштабов и сложности системы. Чистая архитектура высоко ценится в профессиональном сообществе разработчиков
<b>Гексагональная</b>	Гексагональная архитектура (также известная как архитектура портов и адаптеров) схожа с чистой и направлена на создание компонентов с минимальной связанностью, которые взаимодействуют с внешним миром через четко определенные интерфейсы. Приложение строится вокруг внутреннего ядра — бизнес-логики, которая не зависит от инфраструктуры. Все взаимодействия (пользовательский интерфейс, базы данных и т. п.) происходят через адаптеры, подключаемые к портам. Такой подход позволяет удобно тестировать бизнес-логику в изоляции, заменять внешние компоненты без влияния на остальную систему и повышает устойчивость архитектуры
<b>Луковая</b>	Луковая архитектура во многом схожа с чистой и гексагональной. Она структурирует приложение в виде концентрических слоев, где в центре находятся модели домена и доменные сервисы. Наружные слои реализуют интерфейсы, определенные внутри. Важная особенность — зависимости направлены строго внутрь, что защищает предметную область от влияния внешних механизмов
<b>Архитектура вертикальных слайсов</b>	В этой архитектуре система проектируется как набор вертикальных функциональных срезов, охватывающих все уровни приложения — от пользовательского интерфейса до базы данных. Каждый слайс реализует отдельную функцию и разрабатывается как автономный блок. Такой подход отличается от классического послойного проектирования, при котором разработка ведется по уровням. Архитектура вертикальных слайсов повышает гибкость, упрощает параллельную работу над различными частями системы и позволяет быстрее поставлять бизнес-ценность. Каждый слайс может использовать разные технологии, если они вписываются в общую структуру приложения

Каждая из архитектурных моделей имеет свои преимущества и компромиссы. Выбор подхода зависит от множества факторов: требований к масштабируемости и гибкости, удобства сопровождения, специфики бизнес-процессов и опыта команды разработчиков. Одной из наиболее популярных архитектур — и той, о которой стоит узнать подробнее, даже если вы пока не планируете ее применять, — можно назвать чистую архитектуру Роберта Мартина. С ней тесно связаны и другие похожие стили, например, гексагональная и луковая архитектура. Подробнее о чистой архитектуре мы поговорим позже в этой главе.



На первый взгляд многие архитектурные стили могут показаться очень похожими — и действительно, отличия между ними иногда тонкие. Однако сторонники разных подходов могут проявлять настоящую страсть к отстаиванию деталей и нюансов.

## Разделение ответственности команд и запросов (CQRS)

*CQRS (Command and Query Responsibility Segregation)* — это паттерн проектирования и архитектурный стиль, при котором модели для чтения и обновления данных разделяются. Он основан на фундаментальном принципе информатики — разделении команд и запросов (CQS, Command-Query Separation), согласно которому каждый метод должен быть либо командой (выполняющей действие), либо запросом (возвращающим результат), но не одновременно тем и другим. CQRS расширяет этот принцип до уровня архитектуры, разделяя не только поведение, но и инфраструктуру, отвечающую за чтение и запись данных.

Преимущества CQRS.

- **Упрощенный дизайн.** Разделение логики чтения и записи позволяет проектировать более целенаправленные и оптимизированные модели. Модель чтения можно настраивать под производительность запросов, а модель записи — под обработку команд.
- **Масштабируемость.** Системы чтения и записи можно масштабировать независимо. Например, если приложение испытывает высокую нагрузку на чтение, можно масштабировать только модель чтения.
- **Гибкость.** CQRS позволяет использовать разные представления данных. Для чтения можно применять денормализованные структуры, а для записи — нормализованные.
- **Безопасность.** Модели команд и запросов могут требовать разных уровней авторизации, что повышает общий уровень безопасности системы.

Хотя CQRS и способен предоставить серьезные преимущества, это не панацея для всех проектов. Его внедрение может добавить сложности, особенно в случаях, когда граница между командами и запросами размыта либо когда затраты на поддержку отдельных моделей превышают выгоду от их разделения. Одна из распространенных трудностей — обеспечение согласованности между моделями чтения и записи. Поскольку эти модели существуют отдельно, данные в представлении для чтения могут быть не сразу актуальными после выполнения команды.

Когда CQRS особенно уместен:

- в системах со сложной предметной областью, где сценарии чтения и обновления данных отличаются не только по структуре, но и по частоте или логике;
- в приложениях с высокой нагрузкой на чтение, где отдельная оптимизация модели чтения может дать заметный прирост производительности;
- в архитектурах, где важно четкое разделение ответственности, упрощающее сопровождение, тестирование и развитие кода.

CQRS — мощный паттерн, особенно при работе с крупными и сложными системами, где важны масштабируемость, гибкость и удобство сопровождения. Однако, как и любое серьезное архитектурное решение, CQRS влечет за собой дополнительную сложность и накладные расходы, связанные с реализацией и поддержкой. Его применение будет оправдано только тогда, когда преимущества действительно перевешивают затраты. Выбор в пользу CQRS должен основываться на реальных потребностях и ограничениях конкретного проекта, а не на моде или желании использовать продвинутую архитектуру.

## Концепции архитектуры решений

В табл. 18.4 приведены некоторые ключевые понятия, являющиеся неотъемлемой частью архитектуры решения.

**Таблица 18.4.** Краткое описание концепций архитектуры решения

Концепция	Описание
<b>Анализ требований</b>	Включает сбор и интерпретацию как функциональных, так и нефункциональных требований проекта. Понимание ожиданий заинтересованных сторон критически важно для определения объема решения и обеспечения соответствия бизнес-целям
<b>Выбор технологии</b>	Одна из ключевых задач архитектора решений — определение технологий, которые будут использоваться в проекте: языков программирования, фреймворков, баз данных, облачных платформ и сопутствующих инструментов. Выбор основывается на таких факторах, как производительность, масштабируемость, стоимость, совместимость и соответствие технологической экосистеме организации
<b>Интеграционный дизайн</b>	Большинство решений должны взаимодействовать с уже существующими системами. Интеграционный дизайн определяет, как различные системы обмениваются данными, какие используются форматы и протоколы и как обеспечивается надежная совместимость между ними
<b>Дизайн безопасности</b>	Безопасность имеет первостепенное значение в архитектуре решений. Разработка стратегии защиты данных и систем включает реализацию механизмов проверки подлинности, авторизации, шифрования, а также обеспечение соответствия требованиям стандартов и нормативов безопасности. Включает реализацию механизмов проверки подлинности, авторизации, шифрования, а также обеспечение соответствия требованиям стандартов и нормативов безопасности
<b>Архитектура инфраструктуры</b>	Хотя архитектура решений часто сосредоточена на ПО, основная роль инфраструктуры — локальной или облачной — остается критически важной. Архитектура инфраструктуры включает проектирование оборудования, сетей и сервисов, необходимых для надежной поддержки развертывания, эксплуатации и масштабирования решений

*Продолжение* 

**Таблица 18.4 (окончание)**

Концепция	Описание
<b>Архитектура данных</b>	Описывает, как в системе организуются, хранятся и обрабатываются данные. Включает проектирование баз данных, моделирование данных, проектирование потоков данных, а также обеспечение масштабируемости, устойчивости и целостности данных. Архитектура данных играет ключевую роль в эффективном и рациональном использовании информации в организации
<b>Масштабируемость и планирование производительности</b>	Предусматривает разработку решений с расчетом на будущий рост нагрузки. Включает планирование горизонтального и вертикального масштабирования, оптимизацию производительности и обеспечение устойчивости работы системы при увеличении объемов данных и числа пользователей
<b>Аварийное восстановление и высокая доступность</b>	Предусматривает создание решений, способных продолжать работу даже при сбоях. Включает стратегии резервного копирования, репликации, отказоустойчивости и процессы быстрого восстановления для минимизации времени простоя и потерь данных
<b>Дизайн пользовательского опыта (UX)</b>	Хотя традиционно UX связывают с дизайном продукта, он также играет важную роль в архитектуре решений, особенно для систем с прямым взаимодействием с пользователем. Продуманное, интуитивное и эффективное использование системы напрямую влияет на удовлетворенность пользователей и их продуктивность
<b>Соблюдение требований и управление</b>	Решения должны соответствовать действующему законодательству, отраслевым стандартам и внутренним политикам организации. Соображения соответствия влияют на обработку данных, управление пользователями и обеспечивают соблюдение стандартов безопасности и контроля
<b>Управление затратами</b>	Стоимость — один из ключевых факторов в архитектуре решений, особенно с учетом активного использования облачных ресурсов. Архитектору необходимо оценивать финансовые последствия выбора технологий и стремиться к разработке экономически эффективных решений без ущерба для производительности, надежности и масштабируемости системы
<b>Интеграция DevOps и CI/CD</b>	Предусматривает внедрение практик DevOps и процессов непрерывной интеграции и доставки (CI/CD) в архитектуру решения. Это помогает автоматизировать сборку, тестирование и развертывание приложений, увеличивая скорость поставки и надежность систем

Архитектура решений — это многогранная дисциплина, требующая целостного понимания как технических аспектов, так и бизнес-контекста. Учитывая перечисленные концепции, архитекторы могут проектировать системы, которые не только технологически обоснованы, но и органично сочетаются с целями и задачами организации. При этом архитекторам решений необходимо учитывать не только технические, но и финансовые, юридические и организационные аспекты проекта.

Технологические задачи, как правило, оказываются наименее сложной частью работы архитектора.

## В заключение

Во многих проектах, особенно в небольших организациях, роли архитектуры ПО и архитектуры решений могут пересекаться или даже выполняться одним и тем же человеком. Тем не менее важно понимать разницу между ними.

- *Архитектура ПО* отвечает за принятие ключевых решений по структуре и дизайну самого программного решения.
- *Архитектура решения* делает шаг назад и фокусируется на интеграции этого ПО в более широкую бизнес-экосистему и технологический ландшафт, чтобы решить конкретные задачи или удовлетворить определенные требования.

Архитекторы ПО строят внутреннюю структуру системы, обеспечивая ее качество и развитие, тогда как архитекторы решений ориентированы на соответствие технологического решения бизнес-стратегии, операционным требованиям и долгосрочным целям организации. Обе роли одинаково важны для создания эффективных, устойчивых и жизнеспособных технологических решений.

О каждой из архитектур можно написать отдельную книгу. В рамках этой главы мы познакомимся с наиболее важными понятиями, чтобы заложить прочную базу знаний и подготовить вас к собеседованиям. Изучение архитектур похоже на освоение иностранных языков: как только вы глубоко разобрались в одном, вам становится легче понимать и другие, ведь у всех есть общая «грамматика» проблем, которые они пытаются решить, просто расставляя разные акценты. Как уже отмечалось ранее, основой, на которой базируются многие архитектурные подходы, является чистая архитектура. Далее мы рассмотрим ее более подробно.

## Чистая архитектура Роберта Мартина

*Чистая архитектура* — это философия проектирования ПО, акцентирующая внимание на четком разделении задач между различными компонентами системы. Ее основная цель — создание приложений, не зависящих от конкретных фреймворков, пользовательских интерфейсов, баз данных и других внешних технологий.

В основе чистой архитектуры лежит стремление разработать системы, которые легко тестировать, масштабировать и сопровождать на протяжении всего их жизненного цикла, а также чтобы изменения во внешней среде не разрушали внутреннюю структуру приложения.

Концепцию чистой архитектуры предложил Роберт С. Мартин (известный в узких кругах как «дядя Боб»). Он сформулировал структурированный подход к по-

строению архитектуры ПО, направленный на использование практик, которые повышают удобство сопровождения кода.



**В копилку знаний:** с биографией Роберта Мартина и его вкладом в развитие отрасли можно ознакомиться на сайте [ru.wikipedia.org/wiki/Мартин,\\_Роберт\\_\(инженер\)](http://ru.wikipedia.org/wiki/Мартин,_Роберт_(инженер)). Также вы можете посетить его личный сайт [cleanCoder.com](http://cleancoder.com). Однако стоит учитывать, что собственный сайт, в отличие от Википедии, отражает точку зрения самого автора.

## Концепции чистой архитектуры

Чистая архитектура основана на принципах, направленных на создание гибких, тестируемых и долгоживущих программных решений.

- **Независимость от фреймворков.** Архитектура не привязана к конкретным библиотекам или фреймворкам. Они рассматриваются как инструменты, а не как основа системы. Вы можете легко менять или исключать их без ущерба для архитектуры.
- **Тестируемость.** Бизнес-логика может быть протестирована в изоляции, без участия пользовательского интерфейса, базы данных, веб-серверов и других внешних компонентов.
- **Независимость пользовательского интерфейса.** Интерфейс может быть легко заменен без изменения остальной системы. Например, веб-интерфейс можно заменить на консольный — и бизнес-логика останется нетронутой.
- **Независимость от базы данных.** Архитектура не зависит от конкретной СУБД. Можно использовать Oracle, SQL Server, файловую систему или что-то иное — бизнес-правила не зависят от хранилища.
- **Независимость от внешних компонентов.** Бизнес-логика изолирована от всего внешнего: пользовательского интерфейса, веб-слоя, баз данных и других инфраструктурных элементов.

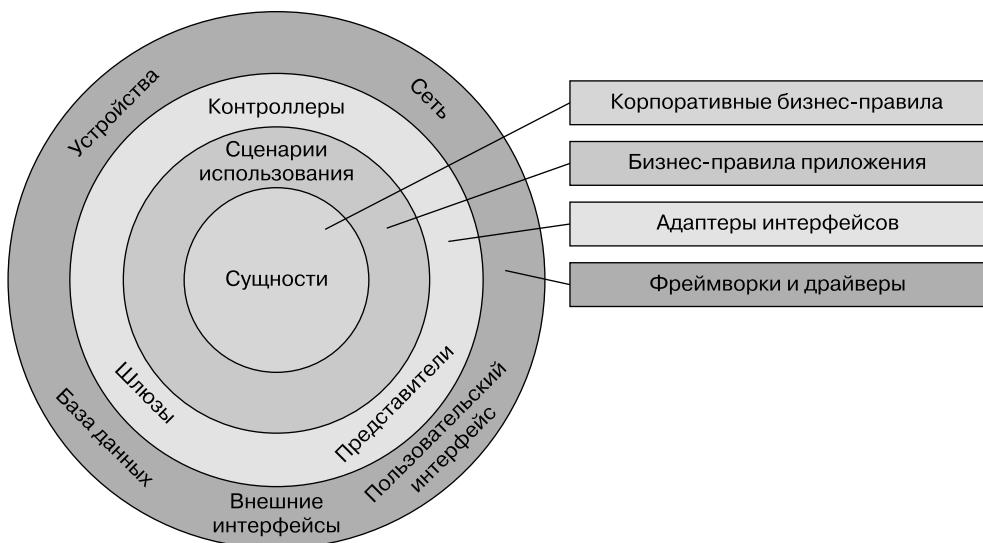
Чистая архитектура строится вокруг нескольких концентрических слоев, каждый из которых имеет свою четкую роль и уровень ответственности (рис. 18.2).

Слои организованы так, что зависимости всегда направлены вовнутрь — к сущностям и бизнес-правилам, а не наружу — к фреймворкам или интерфейсам.

- **Корпоративные бизнес-правила/сущности.** Внутренний слой, содержащий высокоуровневые бизнес-правила предприятия. Сущности описывают общие концепции и правила, которые остаются стабильными даже при изменениях внешних требований или технологий.
- **Бизнес-правила приложения/сценарии использования.** Этот слой отвечает за реализацию конкретных сценариев использования системы. Здесь инкапсулируется логика, специфичная для приложения. В .NET такие сценарии обычно реализуются в виде классов с четко определенными интерфейсами

в библиотеке классов, что способствует модульности и делает код пригодным для различных архитектур — от модульного монолита до микросервисов.

- **Адаптеры интерфейсов.** Этот слой преобразует данные между внутренними моделями приложения и внешними источниками или приемниками данных. Он включает контроллеры, шлюзы и представителей. В .NET-проектах сюда могут входить компоненты объектно-реляционного отображения, такие как EF Core, или низкоуровневые API, например ADO.NET SqlClient с возможным использованием ORM-помощников вроде Dapper.
- **Фреймворки и драйверы.** Внешний слой, где располагаются фреймворки и инструменты: базы данных (например, SQL Server), веб-фреймворки (например, ASP.NET Core), файловые системы, сетевые стеки и прочие элементы инфраструктуры.



**Рис. 18.2.** Круговые слои чистой архитектуры

Реализация чистой архитектуры предполагает следование этим принципам на протяжении всего процесса разработки ПО.

## Определение сущностей

Сущности в чистой архитектуре представляют собой бизнес-объекты приложения. Они инкапсулируют наиболее общие и фундаментальные бизнес-правила, которые остаются стабильными независимо от изменений во внешней среде. Даже если база данных, пользовательский интерфейс или внешние интеграции будут изменены, сами сущности, как правило, затрагиваются минимально — при условии, что бизнес-правила системы остаются прежними.

В экосистеме .NET сущности обычно реализуются как простые объекты C# (Plain Old C# Object, POCO). Они не должны зависеть от сторонних библиотек

или фреймворков и должны сосредоточиваться исключительно на бизнес-логике и свойствах, критичных для предметной области.

К примеру, сущность `User` может содержать методы для проверки возраста пользователя или правильности формата его электронной почты. Однако она *не должна* заботиться о том, как эти данные отображаются в пользовательском интерфейсе или как сохраняются в базе данных.

## Определение сценариев использования и бизнес-правил

Сценарии использования управляют потоком данных к сущностям и от них, а также реализуют дополнительные бизнес-правила, специфичные для операций внутри приложения. В чистой архитектуре их иногда называют интеракторами. Они служат связующим звеном между слоем пользовательского интерфейса и ядром бизнес-логики.

В экосистеме .NET реализация сценариев использования обычно включает создание сервисов или обработчиков команд и запросов, которые инкапсулируют бизнес-правила, специфичные для приложения.

Такие компоненты:

- принимают данные с уровня контроллеров;
- выполняют необходимые операции над сущностями;
- возвращают результаты на уровень представления.

Каждый сценарий использования должен быть узкоспециализированным, управляя одной конкретной бизнес-операцией или транзакцией.

## Реализация интерфейсов

Интерфейсы в чистой архитектуре определяют границы взаимодействия между различными слоями приложения, например между сценариями использования и сущностями или между сценариями использования и инфраструктурными сервисами. Они поддерживают принцип разделения ответственности, который лежит в основе чистой архитектуры.

В экосистеме .NET интерфейсы часто используются для:

- абстракции доступа к хранилищу данных;
- интеграции с внешними API и сторонними сервисами;
- организации потоков данных между сценариями использования и сущностями.

К примеру, интерфейс `IUserRepository` может объявлять методы для получения и сохранения пользовательских данных. Конкретная реализация интерфейса (`UserRepository`) может использовать любой механизм доступа к данным — например, Entity Framework Core, Dapper или ADO.NET `SqlClient` — в зависимости от потребностей проекта.

## Реализация представителей

Представители — часть границы вывода в чистой архитектуре. Они получают данные от сценариев использования и подготавливают их для передачи на уровень фреймворков и драйверов. Это разделение позволяет изолировать бизнес-логику и сценарии использования от требований конкретных технологий пользовательского интерфейса или инфраструктуры.

В экосистеме .NET представители обычно реализуются как компоненты, которые преобразуют модели данных из сценариев использования в модели представления или объекты передачи данных (Data Transfer Object, DTO), а также формируют структуры, удобные для отправки на фронтенд. Такое разделение гарантирует, что изменения в структуре пользовательского интерфейса или логике отображения не затронут основную бизнес-логику приложения.

## Реализация контроллеров

В веб-приложениях на платформе .NET, использующих такие фреймворки, как ASP.NET Core, контроллеры играют роль входной границы в чистой архитектуре. Они принимают входящие запросы, делегируют обработку бизнес-логики соответствующим сценариям использования и возвращают клиенту сформированный ответ.

В .NET контроллеры должны оставаться компактными и сосредоточенными на маршрутизации запросов и проверке входных данных. Они не должны содержать бизнес-логики — ее место в слоях сценариев использования или сущностей.

## Передовой опыт в области чистой архитектуры .NET

Эффективные методы реализации чистой архитектуры в .NET следующие.

- **Инверсия зависимостей.** Принцип инверсии зависимостей занимает центральное место в чистой архитектуре: модули высокого уровня не должны зависеть от модулей низкого уровня — и те и другие должны зависеть от абстракций, например интерфейсов.
- **Игнорирование персистентности.** Сущности и бизнес-логика должны быть независимыми от механизмов хранения данных. При проектировании сущностей не учитывается, где и как будут сохраняться или извлекаться данные, — это забота инфраструктурного слоя.
- **Тестируемость.** Чистая архитектура упрощает модульное тестирование. Изолированная бизнес-логика позволяет писать тесты без необходимости настраивать базы данных, веб-серверы или другие внешние зависимости.

Чистая архитектура в .NET-проектах требует осознанного и аккуратного подхода к проектированию. Основная идея — четко разделять ответственность между компонентами и изолировать детали реализации. Вместо того чтобы воспринимать программу как набор последовательно выполняемых операций, она строится

вокруг отдельных, независимых моделей и действий, которые общаются между собой через заранее определенные интерфейсы. Такой подход упрощает тестирование, сопровождение и масштабирование системы.

## Построение архитектурных диаграмм с помощью Mermaid

Как вы уже знаете из главы 6, Mermaid — это текстовый инструмент для создания диаграмм, синтаксис которого напоминает Markdown. Благодаря этому его удобно использовать при документировании и проектировании, особенно в среде разработчиков. Mermaid хорошо интегрируется с различными платформами и инструментами для документирования, поэтому его часто выбирают для визуализации архитектуры и процессов.



Руководство пользователя Mermaid опубликовано на сайте [mermaid.js.org/intro/getting-started.html](http://mermaid.js.org/intro/getting-started.html).

## Визуализация архитектуры и процессов с помощью Mermaid

Инструмент Mermaid особенно удобен при работе с архитектурой ПО, поскольку диаграммы создаются в виде обычного текста. Благодаря этому их легко хранить в системе контроля версий вместе с исходным кодом и документацией. Такой подход упрощает отслеживание изменений, совместную работу и поддержку актуальности данных. Инструмент поддерживает разные типы диаграмм, часто используемые в архитектуре: блок-схемы, диаграммы последовательностей, классов, состояний, а также диаграммы связей между сущностями. Это дает возможность наглядно описывать как структуру системы, так и ее поведение.

Диаграммы Mermaid можно генерировать и обновлять программно, что особенно полезно при создании динамической визуализации текущего состояния системы или ее данных. Их даже можно автоматически обновлять в конвейерах CI/CD, чтобы документация всегда отражала актуальное состояние проекта.

К типичным примерам использования Mermaid относятся:

- диаграммы системной архитектуры — отражают общую структуру программного решения, включая компоненты, сервисы и связи между ними;
- диаграммы процессов и рабочих потоков — помогают визуализировать бизнес-процессы, последовательность действий и взаимодействие пользователей с системой;
- модели данных и диаграммы «сущность — связь» (*ER-диаграммы*) — используются для описания схем БД, взаимосвязей между сущностями и движения данных;

- диаграммы последовательностей — показывают пошаговое взаимодействие между компонентами системы во времени и помогают разобраться в сценариях использования;
- диаграммы развертывания — наглядно показывают, как программные компоненты размещаются в разных средах и инфраструктурах.

## Типы диаграмм Mermaid

Инструмент Mermaid поддерживает разные типы диаграмм, которые особенно полезны при проектировании архитектуры и разработке ПО. Ниже приведены наиболее востребованные из них.

- **Блок-схемы** — универсальный способ визуализации потока логики, данных или процессов. Такие диаграммы удобно использовать для описания алгоритмов, технологических процессов и общей структуры системы — они делают сложные идеи более наглядными.
- **Диаграммы последовательностей** — отображают, как компоненты системы взаимодействуют во времени. Особенно полезны при описании вызовов методов между объектами в объектно-ориентированных системах.
- **Диаграммы классов** — помогают визуализировать структуру приложения: классы, их свойства, методы и связи между объектами. Это полезно при проектировании архитектуры и понимании модели предметной области.
- **Диаграммы состояний** — показывают возможные состояния объекта и переходы между ними. Такой тип диаграммы особенно удобен для описания поведения приложений с условными или циклическими потоками.
- **Диаграммы «сущность – связь» (Entity Relationship Diagram, ERD)** — применяются для моделирования баз данных. С их помощью легко представить сущности, их атрибуты и связи, что особенно важно при проектировании структуры хранения данных.
- **Диаграммы Ганта** — хотя их обычно не связывают напрямую с архитектурой ПО, они отлично подходят для планирования сроков и этапов разработки. С их помощью удобно отображать фазы проекта, зависимости и ключевые точки, что помогает в управлении сроками и составлении графиков.
- **Диаграммы компонентов** — показывают, как организованы и связаны между собой ключевые части системы. Такие диаграммы дают общее представление о структуре системы и особенно полезны при документировании архитектурных решений.
- **Диаграммы пути пользователя** — фокусируются на взаимодействии пользователя с системой. Это своего рода повествовательные схемы, помогающие понять, как человек достигает своей цели в приложении. Такой подход полезен при проектировании, ориентированном на пользователя, — он помогает согласовать архитектуру с реальными сценариями использования.

Синтаксис Mermaid достаточно простой, чтобы вставлять диаграммы прямо в текстовую документацию — без использования сторонних графических редакторов. Это экономит время и помогает поддерживать документацию в актуальном состоянии: диаграммы обновляются вместе с текстом и остаются частью единого источника информации.

Поскольку блок-схемы — один из самых универсальных и гибких типов диаграмм в Mermaid, начнем с них.

## Блок-схемы Mermaid

Блок-схемы подходят для самых разных сценариев, связанных с архитектурой и разработкой ПО. С их помощью можно:

- описывать этапы процессов или рабочих потоков;
- наглядно показывать логические шаги в алгоритмах;
- выделять точки принятия решений и результаты этих решений;
- отображать движение данных или управление в системе.

Такие диаграммы легко воспринимаются — даже неспециалистами в ИТ. Их можно использовать как для общего обзора, так и для детального пошагового описания, поэтому блок-схемы подходят для разных уровней абстракции и разных целевых аудиторий.

### Синтаксис блок-схемы

Основные элементы синтаксиса для построения блок-схем с помощью Mermaid приведены в табл. 18.5.

**Таблица 18.5.** Синтаксис блок-схемы Mermaid

Синтаксис	Пример	Описание
flowchart <направление>	TD, TB, LR, RL, BT	Задает направление потока: сверху вниз (TD и TB), слева направо (LR), справа налево (RL), снизу вверх (BT). По умолчанию используется TD
node	A, B, C	Определяет идентификаторы для узлов
node[текст]	A[Пуск], C[Доступ к информационной панели]	Добавляет текстовую подпись в прямоугольнике. Используется для обозначения шагов или действий в процессе

Синтаксис	Пример	Описание
node{текст}	B{Вошел ли пользователь в систему?}, D{Верно ли имя пользователя и пароль?}	Отображает подпись внутри ромба — используется для обозначения точек принятия решений, в которых поток может расходиться в зависимости от условия
-->	A --> B	Простая стрелка между узлами. Показывает направление потока от одного шага или решения к другому
--> метка	B --> Да  D	Стрелка между узлами с подписью. Используется для пояснения перехода, например при условии «Да» или «Нет»
classDef <имя_стиля> <настройки>	classDef startend fill:#f9f,stroke:#333, stroke-width:4px;	Определяет именованный стиль (класс) для оформления узлов. Настройки могут включать цвет заливки (fill), цвет обводки (stroke) и ее толщину (stroke-width)
class <спи-сок_узлов> <имя_стиля>;	class A,L startend;	Применяет ранее определенный стиль к одному или нескольким узлам (перечисляются через запятую)

Помимо стандартных прямоугольников и ромбов, Mermaid поддерживает и другие формы узлов. Вот как они задаются:

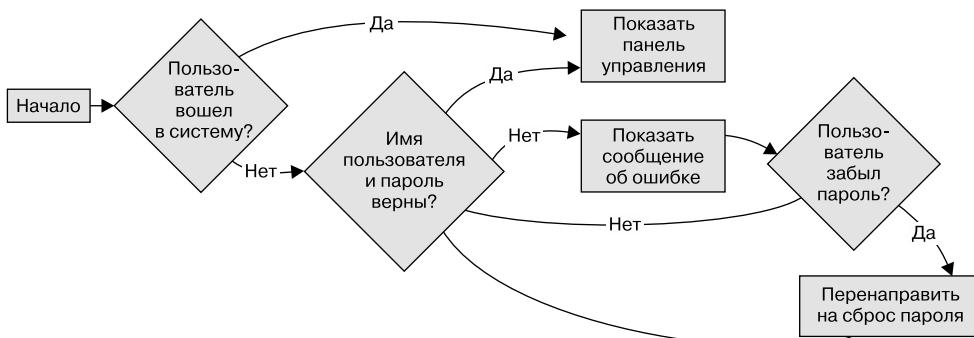
- (...) — скругленный прямоугольник;
- ((...)) — круг;
- (((...)) — двойной круг;
- ([ ... ]) — капсула (похож на скругленный прямоугольник, но уже);
- [[ ... ]] — подпрограмма;
- [(...)] — цилиндр, обычно используется для обозначения базы данных;
- {{...}} — шестиугольник;
- [/.../] или [\...\\\] — параллелограмм;
- [/...\\\] или [\\\.../] — трапеция;
- >... — прямоугольник с вырезанным треугольником с левой стороны.

Рассмотрим несколько примеров с использованием синтаксиса Mermaid — они помогут понять, как можно визуализировать разные сценарии с помощью блок-схем. Чтобы просмотреть такие диаграммы, обычно достаточно вставить код Mermaid в Markdown-файл и открыть его в среде, поддерживающей визуализацию Mermaid, например на GitHub или в редакторе Markdown с соответствующим плагином.

## Пример блок-схемы — процесс входа пользователя в систему

Эта блок-схема иллюстрирует упрощенный процесс входа пользователя в систему. В ней показаны основные точки принятия решений — успешная проверка подлинности и обработка ситуации, когда пользователь забыл пароль (см. рис. 18.3 и следующий фрагмент разметки):

```
flowchart LR
    A[Начало] --> B{Пользователь вошел в систему?}
    B -- Да --> C[Показать панель управления]
    B -- Нет --> D{Имя пользователя и пароль верны?}
    D -- Да --> C
    D -- Нет --> E{Показать сообщение об ошибке}
    E --> F{Пользователь забыл пароль?}
    F -- Да --> G[Перенаправить на сброс пароля]
    F -- Нет --> D
    G --> D
```



**Рис. 18.3.** Блок-схема базового процесса входа пользователя в систему

## Пример блок-схемы: процесс разработки ПО

Следующая диаграмма показывает типовой, упрощенный процесс разработки ПО — от сбора требований до развертывания.

В конце диаграммы определяются классы стилей, чтобы визуально различать типы узлов. Например, класс `startend` применяется к узлам `A[Start]` и `L[End]`, чтобы сделать их более заметными. Классы `process` и `decision` помогают отличать действия от точек принятия решений (см. рис. 18.4 и соответствующую разметку ниже).

```
flowchart
    A[Начало: ошибка выявлена] --> B[Зарегистрировать ошибку в трекере]
    B --> C{Ошибка воспроизводится?}
    C -- Да --> D[Исправить ошибку]
    C -- Нет --> E[Запросить дополнительную информацию]
    E --> F{Дополнительная информация получена?}
    F -- Да --> C
    F -- Нет --> G[Закрыть как "Ошибка не воспроизводится"]
    D --> H[Протестировать исправление]
```

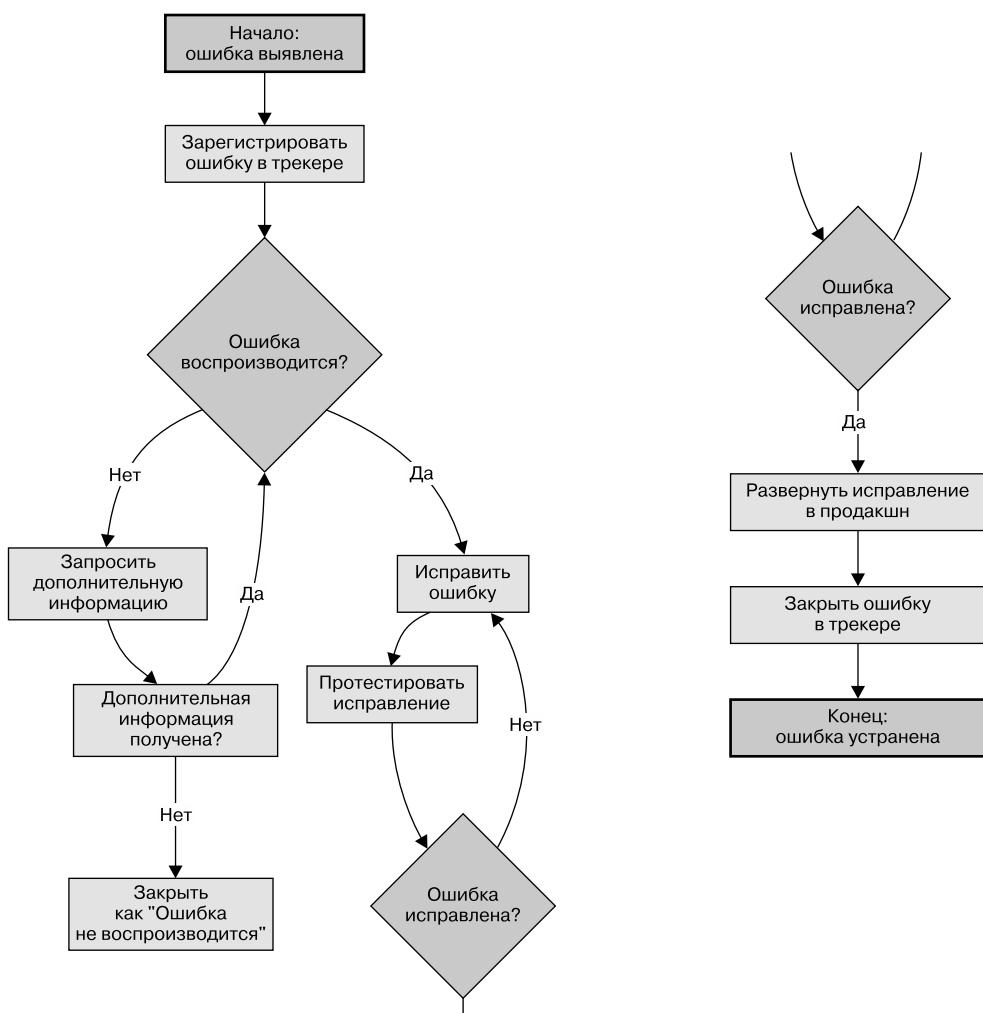
```

H --> I{Ошибка исправлена?}
I -->|Да| J[Развернуть исправление в продакшен]
I -->|Нет| D
J --> K[Закрыть ошибку в трекере]
K --> L[Конец: ошибка устранена]

classDef startend fill:#f9f,stroke:#333,stroke-width:4px;
classDef process fill:#bbf,stroke:#333,stroke-width:2px;
classDef decision fill:#fbf,stroke:#333,stroke-width:2px;

class A,L startend;
class B,D,E,F,G,H,J,K process;
class C,I decision;

```



**Рис. 18.4.** Блок-схема процесса отслеживания и исправления ошибок

Хотя блок-схемы остаются одним из самых гибких и часто используемых типов диаграмм, Mermaid поддерживает и другие форматы, которые также широко применяются в архитектуре и разработке. В частности, диаграммы последовательностей помогают наглядно показать взаимодействие между компонентами системы во времени. Они особенно удобны для иллюстрации сценариев использования и бизнес-процессов.

Теперь подробнее рассмотрим диаграммы последовательностей.

## Диаграммы последовательностей Mermaid

Диаграммы последовательностей позволяют понять, как система ведет себя во времени: они показывают порядок выполнения процессов, а также поток сообщений и событий между участниками.

Такая визуализация помогает выявить потенциальные проблемы, убедиться, что все необходимые взаимодействия учтены, и наладить коммуникацию внутри команды — благодаря ясному и наглядному представлению происходящего.

Диаграммы последовательностей особенно полезны при описании точного порядка операций. Они повышают точность проектирования, а также упрощают планирование и устранение ошибок в процессе разработки.

### Синтаксис диаграммы последовательностей

Основные элементы синтаксиса для создания диаграммы последовательности приведены в табл. 18.6.

**Таблица 18.6.** Синтаксис диаграммы последовательностей

Синтаксис	Пример	Описание
sequenceDiagram	sequenceDiagram	Объявляет диаграмму последовательности
participant <имя> as <подпись>	participant U as User participant F as Front-End participant B as Back-End	Определяет участников диаграммы — они отображаются в верхней части и взаимодействуют между собой
узел1->>узел2: <текст>	U->>F: Clicks Register F->>B: POST /register	Задает сообщение между участниками. Показывает, кто инициирует взаимодействие и кому оно направлено
alt <текст>	alt successful registration	Начало блока альтернативного сценария
else <текст>	else registration error	Альтернативная ветвь в блоке alt
end	end	Завершает блок alt или другой управляющий блок
loop <текст>	loop Data Validation	Начало цикла — для повторяющихся действий

## Пример диаграммы последовательности — процесс регистрации пользователя

Следующая диаграмма показывает, как проходит регистрация нового пользователя на сайте. В ней отражены шаги взаимодействия между пользователем, внешним интерфейсом и серверной частью системы (см. рис. 18.5 и соответствующую разметку).

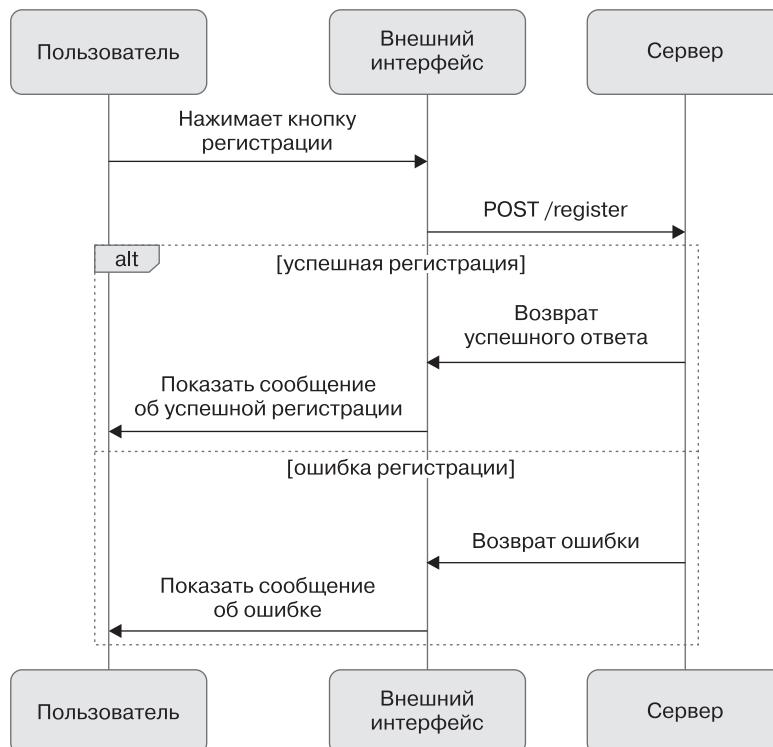
```

sequenceDiagram
    participant U as Пользователь
    participant F as Внешний интерфейс
    participant B as Сервер

    U->>F: Нажимает кнопку регистрации
    F->>B: POST /register

    alt успешная регистрация
        B->>F: Возврат успешного ответа
        F->>U: Показать сообщение об успешной регистрации
    else ошибка регистрации
        B->>F: Возврат ошибки
        F->>U: Показать сообщение об ошибке
    End

```



**Рис. 18.5.** Диаграмма последовательности действий, показывающая процесс регистрации пользователя

На приведенной выше диаграмме:

- участники обозначены явно — это разные сущности, участвующие в процессе: пользователь, внешняя часть (фронтенд) и серверная часть (бэкенд);
- символ `->` указывает на сообщение или действие, направленное от одного участника к другому;
- блок `alt` используется для отображения альтернативных сценариев — например, успешной регистрации или ошибки. Блок завершается с помощью ключевого слова `end`.

Инструмент Mermaid поддерживает несколько вариантов стрелок, отражающих разные типы взаимодействия между участниками:

- `->` — обычное сообщение, сплошная линия;
- `-->` — сообщение с пунктирной линией;
- `->>+` — начало активного взаимодействия (участник становится активным);
- `->>-` — завершение активного взаимодействия.



На предыдущей диаграмме показаны не все типы стрелок.

Возможны также стрелки, направленные от участника к самому себе, — такой прием используется для отображения внутренних операций или повторяющихся действий.

### Пример диаграммы последовательности: запрос к базе данных

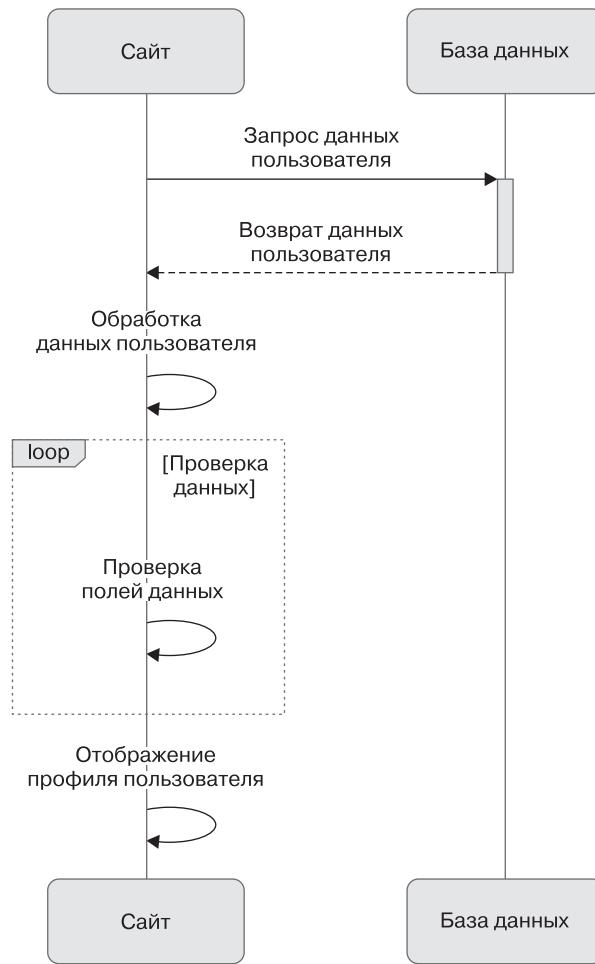
Следующий пример демонстрирует простой сценарий, в котором сайт отправляет запрос к базе данных для получения информации о пользователе и обрабатывает ответ (см. рис. 18.6 и соответствующую разметку).

```
sequenceDiagram
    participant W as Сайт
    participant D as База данных
    W->>+D: Запрос данных пользователя
    D-->>-W: Возврат данных пользователя
    W->>W: Обработка данных пользователя
    loop Проверка данных
        W->>W: Проверка полей данных
    end
    W->>W: Отображение профиля пользователя
```

Ключевые особенности диаграммы.

- Символы `->>+` и `-->>-` отражают начало и завершение взаимодействия с базой данных: первое сообщение инициирует процесс, второе — возвращает результат. Такая запись подчеркивает границы активности между участниками.

- Блок `loop` обозначает повторяющееся действие — в данном случае это проверка полей данных, выполняемая на стороне сайта как часть внутренней логики.



**Рис. 18.6.** Диаграмма последовательности, показывающая, как сайт запрашивает базу данных

Эта диаграмма — наглядный пример, как Mermaid можно использовать для описания последовательности операций и взаимодействий в системе. Благодаря простому синтаксису диаграммы последовательности делают сложные процессы более понятными и удобными для документирования архитектуры и поведения ПО.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 18.1. Дополнительные онлайн-материалы

В этой главе я рассмотрел только два наиболее часто используемых типа диаграмм Mermaid — блок-схемы и диаграммы последовательностей. Если вам интересно увидеть примеры других типов диаграмм в будущих изданиях книги, пожалуйста, дайте знать, какие из них вы хотели бы видеть в печатной версии.

А пока вы можете самостоятельно познакомиться с другими возможностями Mermaid — ниже я привел несколько полезных ссылок.

- **Диаграммы классов** — помогают отразить структуру системы: классы, атрибуты, методы и связи между объектами. Широко используются в объектно-ориентированном проектировании: [mermaid.js.org/syntax/classDiagram.html](http://mermaid.js.org/syntax/classDiagram.html).
- **Диаграммы состояний** — описывают различные состояния объектов и переходы между ними. Особенно полезны при моделировании жизненного цикла компонентов системы: [mermaid.js.org/syntax/stateDiagram.html](http://mermaid.js.org/syntax/stateDiagram.html).
- **Диаграммы «сущность — связь» (ERD)** — представляют отношения между сущностями в БД. Отлично подходят для моделирования и проектирования баз данных: [mermaid.js.org/syntax/entityRelationshipDiagram.html](http://mermaid.js.org/syntax/entityRelationshipDiagram.html).
- **Диаграммы Ганта** — применяются в управлении проектами. Показывают задачи, сроки, зависимости и стадии выполнения: [mermaid.js.org/syntax/gantt.html](http://mermaid.js.org/syntax/gantt.html).

### Упражнение 18.2. Практические задания

В работе над архитектурой решений на базе .NET часто бывает полезно обсудить идеи — и не всегда с человеком. Попробуйте обратиться к своей любимой языковой модели (LLM), чтобы сформировать или уточнить архитектурный подход. Ниже я привел пример формулировки запроса, с которого можно начать диалог:

*«Прошу вас выступить в роли эксперта по архитектуре ПО на базе .NET и смежных технологий. Мы реализуем проект для платформы электронной коммерции. Какие архитектурные решения стоит рассмотреть? Какие вопросы стоит задать себе при выборе подходящей архитектуры? Если возможно, добавьте примеры кода и диаграммы Mermaid. Предположим, что сторонние библиотеки еще не рассматривались — если вы порекомендуете такие, выделите их преимущества в отдельном разделе».*

## Упражнение 18.3. Проверьте свои знания

Ответьте на следующие вопросы. Если что-то непонятно, попробуйте поискать ответ в интернете. При необходимости загляните в приложение — там вы найдете все ответы.

1. В чем разница между архитектурой ПО и архитектурой решений?
2. Кратко опишите три стиля архитектуры ПО.
3. Что такое предметно-ориентированное проектирование (Domain-Driven Design, DDD)?
4. Что означает сокращение CQRS и в чем суть этого подхода?
5. Кто такой Роберт Мартин и почему его считают важной фигурой в мире .NET-разработки?

## Упражнение 18.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-18---software-and-solution-architecture-foundations](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-18---software-and-solution-architecture-foundations).

## Резюме

В этой главе мы разобрали:

- ключевые понятия архитектуры ПО и архитектуры решений;
- основные идеи чистой архитектуры Роберта Мартина;
- способы создания архитектурных диаграмм с помощью блок-схем и диаграмм последовательности в Mermaid.

В следующей главе речь пойдет о развитии карьеры: вы узнаете, как повысить свои шансы на успех и как убедительно представить себя на собеседовании.

# 19

## Ваша карьера, командная работа и собеседования

Эта заключительная глава посвящена строительству карьеры разработчика, работе в команде и прохождению собеседований на замещение вакансий .NET-разработчика и смежные роли.

Поскольку вы читаете эту книгу, значит, вас интересует работа с платформой .NET – возможно, вы планируете начать карьеру в этой области или хотите перейти на новую позицию. Все, о чем шла речь в предыдущих главах, должно помочь вам подготовиться технически. Теперь пришло время обсудить, как превратить эти знания в работу.



Ответы на примеры вопросов для собеседований доступны по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/interview-qa/readme.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/interview-qa/readme.md).

### Работа в команде разработчиков

Среди читателей моих книг по .NET есть и те, кто занимается программированием не ради профессии, а для собственного удовольствия. Они изучают платформу, чтобы решать личные задачи или создавать программы для себя.

Но большинство – это специалисты, которые уже работают с .NET или стремятся начать карьеру в этой области. Хотя некоторые из них работают самостоятельно, чаще всего .NET-разработчики входят в команду, где взаимодействуют с другими разработчиками и коллегами из смежных сфер.

### Чем занимается .NET-разработчик

Повседневные задачи .NET-разработчика могут сильно различаться в зависимости от проекта и компании. Тем не менее есть общий набор областей, в которых важно хорошо разбираться. Приведу основные из них.

- **Разработка и программирование.** Прежде всего это написание, отладка и сопровождение кода на C# и .NET. Работа может включать создание сайтов,

приложений, сервисов и API, устранение ошибок, оптимизацию производительности и добавление новых функций. Эти темы подробно разбираются во всех главах книги.

- **Сотрудничество.** Вам предстоит взаимодействовать с другими разработчиками, дизайнерами, менеджерами продуктов и заинтересованными сторонами, чтобы создать решение, соответствующее требованиям бизнеса.
- **Тестирование.** Вы будете писать модульные и интеграционные тесты, а в некоторых случаях — и автоматизированные UI-тесты, чтобы обеспечить стабильность и надежность кода.
- **Проверка кода.** Члены команды часто участвуют во взаимных проверках кода, чтобы повышать его качество и делиться знаниями.
- **Документирование.** Придется писать документацию как для других разработчиков, так и для пользователей — чтобы ваш код было проще понять и поддерживать.
- **Обучение и развитие.** Разработчик должен быть в курсе последних изменений в экосистеме .NET в частности и отрасли в целом. Это помогает применять современные подходы и инструменты.
- **Взаимодействие с командой.** Без встреч не обойтись: планирование спринтов, ретроспективы и другие мероприятия в рамках agile-процессов станут частью вашей работы.

Рассмотрим ключевые технические навыки .NET-разработчика.

- **C# и .NET.** Основной язык и платформа, на которых строится вся разработка.
- **ASP.NET Core.** Используется для создания современных веб-приложений и API.
- **Entity Framework Core.** Технология доступа к данным, позволяющая работать с базами данных на уровне объектов.
- **Фронтенд-технологии.** Базовые знания HTML, CSS и JavaScript — особенно для тех, кто работает над веб-интерфейсами.
- **Базы данных.** Владение SQL и понимание работы с реляционными СУБД, например SQL Server.
- **Системы контроля версий.** Умение работать с Git, а также знание платформ вроде GitHub или Azure DevOps.
- **Облачные сервисы.** Знакомство с Azure или другими провайдерами облачной инфраструктуры.

Назовем ключевые гибкие навыки .NET-разработчика.

- **Решение проблем.** Умение мыслить логически, анализировать и находить решения в сложных ситуациях.
- **Коммуникация.** Четкое и понятное общение внутри команды и с внешними участниками проекта.

- **Командная работа.** Умение эффективно взаимодействовать с другими разработчиками, дизайнерами, аналитиками и менеджерами.
- **Адаптивность.** Готовность быстро осваивать новые инструменты, технологии и подходы.
- **Внимание к деталям.** Тщательная работа с кодом, стремление к его качеству и минимизации ошибок.

Инженеров-программистов называют по-разному в зависимости от специализации, компаний и контекста. Вот наиболее распространенные варианты.

- **Разработчик ПО.** Пишет и поддерживает код для различных приложений и систем, создавая решения под конкретные задачи пользователей.
- **Бэкенд-разработчик.** Специализируется на серверной части приложений: работе с базами данных, серверной логикой и *интерфейсами программирования приложений (Application Programming Interfaces, API)*.
- **Фулстек-разработчик.** Занимается как клиентской, так и серверной частью — от пользовательского интерфейса до работы с сервером и БД.
- **Разработчик приложений.** Создает, тестирует и поддерживает приложения для конкретных платформ — например, Windows, macOS, iOS/iPadOS или Android.
- **Системный разработчик.** Работает над системным ПО: операционными системами, сетевыми сервисами, СУБД и другими компонентами низкого уровня.
- **Веб-разработчик.** Разрабатывает и сопровождает сайты и веб-приложения, в том числе интерфейс, навигацию и иногда серверную часть.
- **DevOps-инженер.** Сводит воедино процессы разработки и сопровождения: автоматизирует развертывание, повышает надежность и помогает ускорить выпуск обновлений.

## Карьера .NET-разработчика

Карьерный рост .NET-разработчика обычно проходит через несколько этапов.

1. **Начальный уровень – младший разработчик (джуниор).** Вы сосредоточены на изучении технологий, написании кода под руководством более опытных коллег и участии в простых задачах. Стажировки особенно полезны на этом этапе, чтобы получить практический опыт.
2. **Средний уровень – разработчик ПО (мидл).** Вы берете на себя большую ответственность, участвуете в проектировании архитектуры и помогаете младшим коллегам. Часто на этом этапе выбирают специализацию, например DevOps, базы данных или фронтенд-разработка.
3. **Продвинутый уровень – старший разработчик (сеньор).** Вы участвуете в ключевых технических решениях, ведете проекты, наставляете команду.

Возможно, возьмете на себя роль технического руководителя (техлида), координирующего работу других разработчиков и следящего за соблюдением технических стандартов.

Многие выбирают профессию разработчика потому, что она сочетает техническую глубину и творческий подход к решению задач. В работе .NET-разработчика важно не только хорошо программировать и понимать архитектуру, но и уметь общаться, работать с командой и выстраивать взаимодействие с руководством — все это тоже часть профессии.

Карьерный путь гибкий. Вы можете углубиться в техническую специализацию, взять на себя управленческую роль или перейти в смежную область. Вне зависимости от того, делаете ли вы первые шаги или стремитесь продвинуться дальше, профессия разработчика — это динамичное и развивающееся направление с широкими возможностями для роста.

И наконец, чтобы успешно работать в команде, важно понимать, с кем вы взаимодействуете ежедневно и как выстраивать сотрудничество с другими специалистами.

## Роли в команде разработчиков, с которыми вам предстоит взаимодействовать

В повседневной работе .NET-разработчик взаимодействует с разными специалистами, каждый из которых играет свою роль в достижении целей проекта. Хорошее понимание этих ролей помогает наладить рабочие отношения, повысить качество коммуникации и в итоге улучшить общий результат. Чем лучше разработчик учитывает задачи, потребности и вклад коллег, тем слаженнее идет работа команды.

Давайте разберемся, с кем .NET-разработчику чаще всего приходится работать, какие задачи решают эти специалисты и как происходит взаимодействие.

### Менеджер проекта

*Менеджер проекта (Project manager, PM)* отвечает за соблюдение сроков, управление бюджетом и ресурсами, а также за соответствие проекта ожиданиям заказчиков и заинтересованных сторон.

.NET-разработчику часто приходится взаимодействовать с менеджером проекта — уточнять требования, сообщать о ходе работы и обсуждать возможные риски или препятствия, которые могут повлиять на сроки и объем работ.

Чтобы сотрудничество было эффективным, важно точно оценивать сроки выполнения задач и быть готовым предложить нестандартные решения, если проект сталкивается с проблемами.

## Бизнес-аналитик

*Бизнес-аналитик (Business Analyst, BA)* отвечает за сбор требований от заинтересованных сторон, их уточнение и преобразование в технические спецификации. Он также проверяет, соответствуют ли реализованные функции ожиданиям бизнеса.

.NET-разработчику важно понимать бизнес-логику, изложенную аналитиком, и следить за тем, чтобы технические решения точно отражали потребности заказчика. Иногда разработчик может помочь аналитикам оценить реалистичность и техническую осуществимость предлагаемых решений.

Если вы сможете объяснить технически сложные вещи простым и понятным языком, бизнес-аналитик это оценит — ведь именно ему предстоит доносить суть до руководства, которое может быть еще дальше от технологий.

## Аналитик по обеспечению качества (QA) или тестировщик

В обязанности *специалистов по обеспечению качества (Quality Assurance, QA)* входят разработка планов и сценариев тестирования на основе требований проекта, проведение проверок на соответствие стандартам качества, а также выявление и документирование ошибок для последующего исправления командой разработчиков.

.NET-разработчику предстоит тесно взаимодействовать с QA-специалистами: анализировать найденные дефекты, расставлять приоритеты для исправления ошибок, а в некоторых случаях — помогать с автоматизированными тестами или предоставлять вспомогательные модульные тесты.

Чем больше тестов вы пишете самостоятельно, тем лучше начинаете понимать потребности тестировщика. Это помогает писать более надежный и качественный код, избегая типичных причин сбоев. А если выстроить хорошее рабочее взаимодействие, цикл разработки, тестирования и исправления будет проходить быстрее и эффективнее — и это заметят руководители.

## UX-дизайнер пользовательского опыта

*Дизайнер пользовательского опыта (User Experience, UX)* отвечает за проектирование интерфейсов, которые соответствуют потребностям конечных пользователей. В его задачи входят создание логики взаимодействия, разработка макетов, прототипов, электронных схем и спецификаций интерфейса.

Для .NET-разработчика сотрудничество с UX-дизайнером особенно важно, чтобы обеспечить плавную интеграцию между фронтендом и серверной частью. Обратная связь помогает уточнять детали и улучшать восприятие интерфейса — и с визуальной стороны, и с точки зрения поведения.

Честно говоря, большинство разработчиков, которых я встречал, не слишком хорошо разбираются в UX- и UI-дизайне. Но если вы будете работать бок о бок

с дизайнером, сможете прокачать эти навыки и начать лучше чувствовать, каким должен быть хороший пользовательский интерфейс. На одной из самых известных конференций — D5 в мае 2007 года — Стива Джобса и Билла Гейтса спросили, в чем заключаются их сильные стороны и как они повлияли на индустрию. Вот что тогда сказал Гейтс:

*«Я бы многое отдал, чтобы обладать вкусом Стива. [...] У него отличный вкус и на людей, и на продукты. Решения, которые лично мне казались чисто инженерными, Стив решал с точки зрения “люди — продукт”, и для меня сложно объять это. Стив принимает решения, полагаясь на свое знание желаний людей. Для меня это очень сложно. Он все делает не так, как все, и это просто удивительно».*

Отличный вкус — то, чего в инженерии часто не хватает. Если вы развиваете такую сторону своей работы, это обязательно будет замечено и оценено.

## Администратор базы данных

Администратор базы данных (*Database Administrator, DBA*) отвечает за проектирование, оптимизацию и поддержку структуры БД, а также за производительность, целостность и безопасность данных.

.NET-разработчику важно взаимодействовать с DBA-специалистами, чтобы убедиться, что SQL-запросы работают эффективно, модели данных соответствуют требованиям приложения, а изменения в базе не приводят к сбоям или снижению производительности.

Знания DBA в своей области глубоки, и стоит обращаться к ним за советом по вопросам проектирования схемы, настройки индексов, масштабируемости и контроля целостности. Такое сотрудничество особенно важно на ранних этапах проекта — еще при создании схемы БД.

Совместно с DBA можно разработать стратегию индексирования, рассчитать возможный рост объемов данных и заранее предусмотреть, как база будет обслуживаться в будущем. Открытое и уважительное сотрудничество помогает избежать многих технических проблем и закладывает прочный фундамент для надежной работы приложения.

## DevOps-инженер

DevOps-инженер отвечает за настройку и сопровождение процессов непрерывной интеграции и доставки (CI/CD), автоматизацию развертывания и тестирования, а также за надежность и масштабируемость инфраструктуры.

.NET-разработчик тесно взаимодействует с DevOps-специалистами при настройке среды, оптимизации процессов разработки, развертывании кода и устранении связанных с этим проблем. Кроме того, разработчику может потребоваться помочь

в настройке конфигураций, зависящих от конкретной среды, — например, для тестирования, промежуточной или продакшен.

Если вы будете использовать такие инструменты, как Docker или .NET Aspire, для локальной разработки и контейнеризации сервисов, это значительно упростит взаимодействие с DevOps-инженерами и сделает вашу работу более предсказуемой и стабильной в разных средах.

## Фронтенд-разработчик

*Фронтенд-разработчик* отвечает за реализацию визуальной и интерактивной части веб-приложения, а также за обеспечение адаптивности интерфейса и его корректной работы в разных браузерах.

Хотя .NET-разработчик иногда занимается клиентской частью, особенно в небольших командах, в большинстве случаев взаимодействие происходит через интеграцию: фронтенд-разработчики подключают API и бэкенд-сервисы, реализованные на .NET, обеспечивая стабильный обмен данными и полноценную работу интерфейса.

Многие .NET-разработчики избегают фронтенд-задач по тем же причинам, по которым им не всегда комфортно работать с UX-дизайном. Но чтобы эффективно сотрудничать, важно понимать, что помогает или, наоборот, мешает фронтенд-специалисту при интеграции с вашим бэкеном. Благодаря этому команда работает слаженнее и результат достигается быстрее.

## Технический руководитель/архитектор

*Технический руководитель/архитектор* отвечает за проектирование архитектуры, выбор технологий, соблюдение стандартов разработки и общее техническое руководство проектом. Он координирует работу команды, задает направление и принимает ключевые инженерные решения.

Для .NET-разработчика регулярное взаимодействие с архитектором или техническим руководителем (техлидом) особенно важно: от этого зависит, насколько точно ваша реализация соответствует архитектурным принципам проекта. Кроме того, техлид или архитектор может помочь с решением сложных технических задач, где требуется опыт и широкое видение.

## Процесс введения в должность

Введение в должность (онбординг) .NET-разработчика может выглядеть по-разному в разных компаниях, но, как правило, включает несколько общих этапов. Задача этого процесса — помочь новому сотруднику быстрее влиться в команду, освоиться в проекте и адаптироваться к культуре компании. Типичная последовательность шагов приведена в табл. 19.1.

**Таблица 19.1.** Этапы введения в должность .NET-разработчика

<b>Этап</b>	<b>Описание</b>
<b>Предварительная подготовка до выхода на работу</b>	Компания заранее готовит все необходимое: настраиваются ИТ-системы, создаются учетные записи, предоставляется доступ к нужному ПО и сервисам. Отдел кадров может заранее отправить документы по оформлению и льготам. Благодаря этому новый сотрудник сможет полноценно включиться в работу с первого дня
<b>Первый рабочий день</b>	Вводный инструктаж по правилам компании, льготам и процедурам. Затем — знакомство с командой, руководством и ключевыми коллегами. Часто проводится экскурсия по офису
<b>Настройка среды и первичное обучение</b>	Настройка среды разработки (например, Visual Studio), доступов к базам данных и облачным сервисам. Знакомство с технологическим стеком, репозиториями и текущими проектами. Запуск существующих решений для изучения кода. Обязательный тренинг по безопасности и нормативам — особенно в компаниях с чувствительными данными
<b>Погружение в проектную работу</b>	После настройки среды разработчик подключается к реальным задачам — сначала небольшим, чтобы освоиться с проектом и кодовой базой. Часто назначается наставник, который помогает в первые месяцы, отвечает на вопросы и поддерживает в адаптации
<b>Поддержка и развитие</b>	Периодические встречи с менеджером и наставником для оценки прогресса, постановки целей и обсуждения сложностей. Возможны тренинги по продвинутым технологиям и гибким навыкам (например, коммуникации и управлению проектами). В конце испытательного срока — формальный обзор результатов и обсуждение дальнейших шагов
<b>Интеграция в культуру компании</b>	Командные и корпоративные мероприятия помогают наладить связи внутри коллектива и за его пределами. Участие в общих собраниях дает представление о целях и направлениях развития компании



**Дельный совет:** чтобы быстрее войти в работу, важно иметь доступ к документации и внутренним материалам. Если вам их не предоставили — не стесняйтесь вежливо попросить. А если документация в принципе отсутствует, стоит спокойно объяснить руководителю, что это может замедлить ваш прогресс и затруднить вливание в работу.

## Как поднять тему профессионального развития

Если вы хотите обсудить обучение, подходите к разговору стратегически. Подготовьте четкий и аргументированный запрос: укажите, какие знания или навыки хотите получить, зачем они вам нужны и как это поможет вашей работе или проекту. Чем конкретнее и обоснованнее будет просьба, тем выше шанс, что вас поддержат.

### Определите конкретные потребности в обучении

Прежде чем обращаться к руководителю с запросом на обучение, важно четко сформулировать, что именно вам нужно и зачем. Здесь можно выделить несколько шагов.

- **Выявите пробелы в навыках.** Определите, какие умения требуют развития, чтобы успешно справляться с текущими или будущими задачами. Это может быть актуализация знаний по новой версии .NET, освоение облачных технологий или внедрение современных практик разработки.
- **Изучите возможные форматы обучения.** Подберите подходящие курсы, книги, конференции или сертификаты, которые помогут закрыть эти пробелы. Оцените стоимость, длительность, формат (онлайн или очный), а также надежность и репутацию провайдера.
- **Уточните бюджет и предложите варианты.** Во многих компаниях есть годовой бюджет на обучение сотрудников. Вместо того чтобы соглашаться на предложенное решение по умолчанию, уточните у руководителя, каким объемом средств вы располагаете, и предложите альтернативы — возможно, более выгодные по стоимости и пользе.

Для многих .NET-разработчиков эффективным подходом оказывается не очное обучение, а комбинированный формат: комплект актуальных книг, подписка на модель LLM (например, Gemini Pro), доступ к видеокурсам и полезным рассылкам по .NET. Такой набор часто оказывается более гибким и выгодным, чем короткий очный семинар.

### Свяжите обучение с целями компании

Чтобы ваш запрос на обучение был убедительным, важно увязать его с задачами компании. Это покажет, что вы мыслите стратегически и хотите расти вместе с организацией.

- **Повышение эффективности работы.** Объясните, как обучение поможет вам лучше справляться с текущими задачами. Например, курс по новой версии ASP.NET Core может ускорить разработку и повысить безопасность кода.
- **Польза для команды.** Покажите, как новые навыки принесут выгоду всему отделу/команде: сократят потребность во внешней помощи, позволят внедрить внутренние решения или ускорят завершение проекта.

- **Поддержка стратегических целей компании.** Свяжите обучение с долгосрочными приоритетами бизнеса — ростом, инновациями, выходом на новые рынки. Обращайте внимание на общекорпоративные собрания: о чём говорит руководитель? Какие направления сейчас в приоритете? Постарайтесь соотнести это со своими предложениями. Помните: часто решение зависит не только от вашего непосредственного руководителя. Сделайте все, чтобы у него были аргументы для защиты вашего запроса перед его руководством. Подготовьте отчет кратко, четко и по делу — и тогда у вашего предложения будет больше шансов.

## Подготовьте анализ затрат и пользы

Чтобы обосновать необходимость обучения, подготовьте краткий анализ, в котором укажите не только затраты, но и потенциальные выгоды.

- **Прямые расходы.** Стоимость курса, командировочные расходы (если они есть), стоимость материалов.
- **Непрямые расходы.** Время, потраченное на обучение, и возможное временное снижение продуктивности.
- **Ожидаемые выгоды.** Оцените, как обучение поможет ускорить завершение проекта, повысить качество продукта, сократить количество ошибок или улучшить удовлетворенность пользователей.

## Сформулируйте гибкий учебный план

Предложите такой формат обучения, который позволит минимизировать влияние на команду и вашу текущую работу. Это покажет, что вы подходите к обучению ответственно и с пониманием бизнес-процессов.

- **Планирование.** Запланируйте обучение на менее загруженный период или разбейте его на части, чтобы не выпадать из рабочих задач надолго.
- **Управление нагрузкой.** Продумайте, как справитесь с текущими обязанностями, — возможно, временно перераспределите задачи или обучите коллег, чтобы обеспечить преемственность.
- **Передача знаний.** По завершении обучения проведите короткий семинар или внутреннюю презентацию для команды. Это усилит ценность обучения для всей группы, а вам даст возможность отточить навыки коммуникации и публичных выступлений — они важны для карьерного роста.

## Назначьте встречу и изложите предложение

Организуйте официальную или неформальную встречу, чтобы обсудить свое предложение. Во время разговора постарайтесь:

- **излагать четко и по существу:** объясните, какое обучение вы предлагаете, зачем оно нужно и как оно принесет пользу не только вам, но и всей команде;

- **быть готовым к вопросам:** руководитель может уточнить, насколько обучение актуально, как оно связано с текущими задачами и как вы планируете справляться с работой во время его прохождения;
- **проявить заинтересованность и зрелость:** покажите, что стремитесь расти как специалист и хотите делать вклад в развитие команды. При этом подчеркните: приоритет — интересы компании.

После встречи отправьте краткое письмо с благодарностью, напоминанием о ключевых аргументах и готовностью доработать предложение, чтобы оно лучше вписывалось в планы и возможности команды.

Когда вы подходите к обучению осознанно и связываете его с целями организации, вы не только повышаете шансы на одобрение, но и демонстрируете свою зрелость и стратегическое мышление. Такой подход показывает, что ваше развитие — это не индивидуальная инициатива, а вклад в общее дело.

## Парное программирование

*Парное программирование* — это метод командной работы, при котором два разработчика пишут код вместе за одним компьютером. Один из них выступает в роли «водителя» и набирает код, а второй — «штурмана» — следит за логикой, замечает возможные ошибки и предлагает улучшения. Партнеры регулярно меняются ролями, что способствует обмену знаниями и помогает находить более эффективные решения. Этот подход широко используется в гибких методологиях разработки, таких как *Agile* и *экстремальное программирование (XP)*.

Типичный рабочий процесс при парном программировании включает:

- **регулярную смену ролей:** водитель и штурман поочередно меняются местами, чтобы оба участвовали и в проработке архитектуры, и в деталях реализации. Это поддерживает баланс между стратегическим и техническим мышлением;
- **непрерывную проверку кода:** код постоянно просматривается обеими сторонами, сразу замечаются ошибки, улучшается стиль и снижаются затраты на последующую отладку;
- **совместный поиск решений:** работа в паре позволяет мгновенно обсуждать идеи и принимать решения, что ускоряет процесс и стимулирует творческий подход;
- **обмен знаниями:** парное программирование особенно эффективно в передаче опыта: между новичками и более опытными коллегами или между разработчиками с разной специализацией.



Многие современные редакторы кода поддерживают удаленное парное программирование. Например, расширение Live Share для Visual Studio позволяет двум разработчикам одновременно работать над одним проектом, видеть изменения в реальном времени и даже совместно отлаживать код. Такая интеграция особенно удобно работает в связке с Microsoft Teams или Discord. Подробнее — на сайте [visualstudio.microsoft.com/services/live-share/](https://visualstudio.microsoft.com/services/live-share/).

## Может ли LLM стать напарником в парном программировании?

Хотя языковая модель, такая как ChatGPT, может значительно помочь в процессе разработки, она пока не способна полностью заменить живого напарника при парном программировании. Тем не менее ее можно использовать как дополнительный инструмент — особенно в роли условного «виртуального штурмана».

Вот как LLM может быть полезна в такой роли.

- **Быстрый поиск информации.** Модель может оперативно ответить на вопросы по синтаксису, концепциям и лучшим практикам, не отвлекая вас от процесса кодирования.
- **Подсказки по коду.** LLM способна предложить фрагмент кода, улучшить уже написанный или обсудить альтернативные варианты реализации. Это похоже на помочь штурмана, который предлагает идеи.
- **Поддержка при отладке.** Модель может объяснить смысл сообщения об ошибке, подсказать, где искать причину, и предложить пути решения.

Ограничения парного программирования с LLM.

- **Галлюцинации.** Модель генерирует ответы, опираясь на статистические закономерности, а не на достоверные факты. В результате она может предложить несуществующий вызов API, ошибочную реализацию алгоритма, сомнительную «оптимизацию» или даже сгенерировать код с потенциальной уязвимостью.
- **Неполное понимание контекста.** LLM может не учитывать широкий контекст проекта и целей команды, что снижает релевантность ее предложений. Если честно, этим грешат и люди. Поэтому важно самому держать в голове всю картину, иначе вы вряд ли превзойдете LLM.
- **Ограниченнная способность к суждениям.** Модель не способна делать инженерные оценки в полной мере — такие, которые основаны на понимании ограничений, рисков и целей конкретного проекта. В этом как раз и проявляется разница между джуниором и сеньором.
- **Отсутствие эмоционального интеллекта.** Модель не чувствует вашего настроения, не подбодрит и не поможет сохранить рабочий настрой. Эмпатия важна в командной работе — и особенно в парном программировании. (Кстати, в ИТ-сфере у людей с этим часто беда. Так что если вы прокачаете эмпатию — это будет ваше конкурентное преимущество.)
- **Слабая адаптивность.** Хотя LLM обучены на огромных массивах данных, они не подстраиваются под ваш проект в реальном времени и не учитывают ваш конкретный прошлый опыт. Но... дайте им пару лет. Думаю, когда я буду готовить второе издание этой книги в конце 2025 года, многое из этой главы придется переписать!

Хотя ChatGPT и другие языковые модели действительно могут помочь в разработке — быстро находить информацию, генерировать код и подсказывать решения, — они пока не способны заменить ту живую, гибкую и контекстно чувствительную

динамику, которая возникает между двумя людьми при парном программировании. Настоящая сила парного подхода — не только в скорости написания кода, а в совместном мышлении, обмене знаниями, обсуждении решений и постоянном обучении. Ни одна LLM на текущем этапе не способна полноценно участвовать в этом процессе.

Да, ИИ может расширять возможности команды, особенно в рутинных задачах. Но именно человеческие качества — проницательность, критическое мышление, эмпатия, умение чувствовать цель и контекст — остаются незаменимыми, особенно в сложной и командной работе.

И если вы хотите идти в ногу со временем — ориентируйтесь не на то, где находится индустрия сегодня, а на то, куда она движется. Как когда-то посоветовал Уолтер Гретцки своему сыну, «*нужно катиться туда, где будет шайба, а не туда, где она была*». Если в 2024 году вы все еще изучаете C#, то, возможно, уже через несколько лет вас будет сопровождать не человек, а ИИ. И чтобы быть по-настоящему ценным в этой новой реальности, стоит уже сейчас развивать те навыки, которые машина не заменит: видеть систему в целом, понимать людей, принимать инженерные решения — и брать на себя ответственность за них.

## Подготовка к собеседованию

Соискание должности .NET-разработчика включает несколько этапов, и чем лучше вы к ним подготовитесь, тем выше шансы произвести хорошее впечатление. Эта часть главы поможет вам понять, как уверенно представить свои навыки, подчеркнуть свой опыт и выделиться среди других кандидатов.

### До отклика на вакансию

Прежде чем откликнуться на вакансию или связаться с работодателем, важно провести подготовку.

#### Освежите свои навыки и знания

Прежде чем откликаться на вакансию, важно убедиться, что у вас есть уверенная техническая база в ключевых технологиях .NET и смежных областях. Вот на какие навыки стоит обратить внимание.

- **Основные технологии.** Уверенное владение C#, ASP.NET Core, .NET и Entity Framework. Если позиция предполагает полную разработку, не забудьте про основы фронтенда: JavaScript, HTML и CSS.
- **Базы данных.** Знание SQL Server — обязательно, а также приветствуется опыт работы с другими СУБД: MySQL, Oracle или NoSQL-системами вроде MongoDB.
- **Контроль версий.** Умение уверенно работать с Git — это по умолчанию для любой современной команды.

- **Инструменты разработки.** Знакомство с Visual Studio и Visual Studio Code – желательно на уровне повседневной работы.
- **Облачные технологии.** Опыт работы с Azure, AWS или Google Cloud – сильное преимущество. При этом сейчас все чаще ценится и обратный опыт: миграция из облака на самостоятельный хостинг. Если вы с этим сталкивались – обязательно укажите.

Хороший набор книг (например, моя трилогия по .NET) поможет вам освоить большинство этих технологий и уверенно чувствовать себя в реальной разработке.

## Получите сертификаты

Хотя наличие сертификатов не всегда обязательно, особенно если у вас уже есть реальный опыт, они могут стать сильным дополнением к резюме – особенно на старте карьеры. Сертификаты демонстрируют вашу мотивацию, серьезное отношение к обучению и уверенное владение технологиями .NET и продуктами Microsoft.

Например, компания Microsoft совместно с платформой freeCodeCamp предлагает новую базовую сертификацию по C#. Она сопровождается бесплатным онлайн-курсом продолжительностью 35 часов. Подробнее – на сайте [www.freecodecamp.org/learn/foundational-c-sharp-with-microsoft/](http://www.freecodecamp.org/learn/foundational-c-sharp-with-microsoft/).

## Ищите доступные вакансии

Поиск вакансий .NET-разработчику стоит начинать с нескольких основных источников.

- **Сайты с вакансиями.** Такие платформы, как hh.ru (HeadHunter), SuperJob и Работа.ru, регулярно публикуют вакансии в сфере .NET-разработки.
- **Сайты компаний.** Если есть компании, которые вам особенно интересны, – загляните в раздел вакансий на их официальных сайтах и подавайте заявку напрямую.
- **Нетворкинг.** Личные контакты по-прежнему работают. Многие вакансии заполняются по рекомендации. Посещайте мероприятия, конференции и встречи .NET-сообщества – это отличный способ пообщаться с потенциальными работодателями.

Кстати, по данным нейросети ChatGPT-4о, на апрель 2025 года на платформе LinkedIn размещено более 35 000 вакансий для .NET-разработчиков по всему миру. Однако это число не отражает вашу индивидуальную ситуацию. Вас интересуют только те предложения, которые доступны в вашей стране, городе или в формате удаленной работы и соответствуют вашему опыту. Поэтому важно сосредоточиться на вакансиях, которые реально доступны и подходят именно вам.

Даже если бы открытых вакансий на Java было в два раза больше, чем на C#, вы бы действительно стали переходить на Java? Вряд ли. Вам нужна не тысяча вариантов – а *одна* подходящая позиция, где требуются ваши навыки, интересы и предлагаются достойные условия.

Тем не менее большое число открытых вакансий говорит о высоком спросе на .NET-разработчиков по всему миру. Роли варьируются от начального уровня до старших должностей и фулстек-позиций, что подтверждает — .NET остается востребованной платформой на всех этапах карьеры. Вакансии доступны как в крупных городах, так и на удаленке, что дает гибкость в выборе формата работы.

Зарплаты .NET-разработчиков существенно различаются в зависимости от региона, уровня опыта и специфики компании.

Ниже — подборка популярных сайтов для поиска вакансий .NET-разработчиков.

- Hh.ru — hh.ru.
- Indeed — www.indeed.com.
- LinkedIn Jobs — www.linkedin.com/jobs.
- Remote OK (удаленная работа) — remoteok.com.
- Stack Overflow Jobs — stackoverflow.com/jobs.
- ZipRecruiter — www.ziprecruiter.com.
- Азия — JobStreet — www.jobstreet.com.
- Индия — Naukri — www.naukri.com.
- Китай — Zhaopin — www.zhaopin.com.

## Составление резюме, онлайн-профилей и заявлений о приеме на работу

Если вы только начали работу в текущей компании, но уже рассматриваете переход, вполне возможно, что ваше резюме и профили в интернете давно не обновлялись. Самое время привести их в порядок.

- **Резюме.** Обновите резюме с акцентом на опыт в .NET, выполненные проекты и используемые технологии. Используйте четкие формулировки, глаголы действия и, по возможности, количественные показатели — это добавляет убедительности. В следующем разделе мы подробнее поговорим о составлении эффективного резюме.
- **Профиль на LinkedIn.** Если вы рассматриваете международные вакансии или хотите повысить профессиональную видимость, обновите свой профиль и следите за компаниями в сфере .NET. Подписывайтесь на компании, которые вам интересны, следите за .NET-новостями и участуйте в тематических обсуждениях<sup>1</sup>.
- **GitHub и портфолио.** Размещайте свои проекты или участуйте в проектах с открытым исходным кодом. Хорошо оформленный репозиторий с примерами

---

<sup>1</sup> Для российского рынка актуальны такие площадки, как hh.ru, Хабр Карьера и GitHub, — именно на них чаще всего ищут и оценивают кандидатов. — *Примеч. пер.*

кода — это весомое дополнение к резюме. Если возможно, создайте отдельный сайт-портфолио с описанием проектов.

- **Сопроводительное письмо.** Составьте краткое и персонализированное письмо, в котором объясните, почему вы подходите на конкретную позицию. Подчеркните знания .NET и проекты, связанные с вакансиеей.
- **Индивидуальные отклики.** Не отправляйте одно и то же резюме на все вакансии. Подстраивайте каждое под конкретную должность, делая акцент на совпадении вашего опыта с требованиями работодателя.

Ваше резюме, профили и заявления формируют первое впечатление о вас как о специалисте. Далее разберем, как составить резюме, которое действительно работает.

## Резюме и CV: в чем разница?

Составляя резюме .NET-разработчика, важно выделить не только технические навыки, но и реальный вклад в проекты, способность работать в команде и адаптироваться к новым задачам. Покажите, как вы решаете практические проблемы с помощью экосистемы .NET.

Полезный шаблон с примерами и рекомендациями по структуре резюме доступен на GitHub по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/resume.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/resume.md).

Этот шаблон поможет подчеркнуть не только ваши технические навыки, но и вклад в проекты, а также роль в команде. Продемонстрируйте в каждом разделе свой опыт и сильные стороны — так ваше резюме действительно привлечет внимание работодателей, ищущих .NET-разработчиков.

Термины «резюме» и *Curriculum Vitae (CV)* часто воспринимаются как синонимы, но на практике это разные документы — особенно в международной среде:

- *резюме* — это краткий (обычно 1–2 страницы) документ, подстраиваемый под конкретную вакансию. В нем отражены ключевые навыки, релевантный опыт и достижения;
- *Curriculum Vitae (CV)* — подробная хронология образования, профессионального опыта, публикаций, сертификатов и всего, что имеет отношение к карьере. Объем CV может достигать десяти и более страниц, особенно у опытных специалистов. Такие документы чаще всего используют в Европе, Азии, Африке, на Ближнем Востоке, а также в академической и научной среде США и Канады. CV не подгоняется под каждую вакансию, а служит как полная профессиональная биография, которую вы просто дополняете по мере развития карьеры.

Если вы претендуете на позицию .NET-разработчика, лучше начать с резюме — оно быстрее читается и проще адаптируется под конкретную вакансию. При этом вы всегда можете упомянуть в сопроводительном письме, что у вас есть и полное CV, если работодатель захочет увидеть более развернутую информацию.

## Улучшите резюме с помощью нейросети

Модель вроде ChatGPT может быть полезным помощником при доработке резюме — особенно если вы готовитесь откликнуться на конкретную вакансию. Вот как можно получить максимальную пользу.

- Загрузите в нейросеть описание вакансии.** Приведите основные требования и ключевые фразы из описания работы. Это поможет LLM понять, какие навыки и опыт наиболее важны и на чем стоит сделать акцент в вашем резюме.
- Загрузите текст текущего резюме.** Можно не прикреплять файл — достаточно скопировать разделы вроде краткого описания, опыта работы, навыков и образования. Это даст модели полную картину вашего профиля.
- Опишите, что вас смущает.** Если вы чувствуете, что какое-то место «не звучит», неубедительно написано или устарело, скажите об этом. Чем точнее вы укажете, что хотите изменить, — тем лучше результат.

Основываясь на предоставленной информации, LLM может предложить следующие улучшения.

- **Переформулировка цели или краткого описания.** Этот раздел можно подстроить под конкретную позицию, чтобы сразу задать нужный тон и показать соответствие роли.
- **Актуализация опыта.** Модель может помочь выделить в вашем опыте именно те навыки и достижения, которые важны для выбранной должности.
- **Уточнение раздела с навыками.** Стоит убедиться, что ключевые технологии, инструменты и методы представлены четко и находятся на видном месте.
- **Форматирование и структура.** Резюме должно выглядеть профессионально и читаться легко — от выравнивания до единообразия стиля.



**Дельный совет:** как и всегда при использовании LLM, воспринимайте результаты как черновик или источник идей. Обязательно перечитывайте и оценивайте предложения критически, особенно в формулировках, касающихся фактов и ваших достижений!

## Комплексная подготовка к собеседованию

Подходите к подготовке всесторонне.

- **Техническая подготовка.** Освежите знания по C#, ASP.NET, Entity Framework и другим технологиям .NET. Будьте готовы обсудить проекты, в которых участвовали, обосновать принятые решения и решить программные задачи.
- **Поведенческие вопросы.** Продумайте примеры из практики, которые демонстрируют вашу способность работать в команде, укладываться в сроки, справляться с трудностями и проявлять инициативу.
- **Тренировка.** Проведите пробное собеседование с коллегой или наставником. Так вы повысите свою уверенность и сможете четче формулировать ответы.

Готовясь к собеседованию на должность .NET-разработчика, важно подготовиться комплексно — технически, логически и психологически. В разных компаниях процесс может отличаться, но чаще всего включает несколько ключевых этапов.

1. **Первичный отбор.** Обычно это телефонный или видеозвонок с представителем отдела кадров или рекрутером. Цель — понять, насколько вы соответствуете требованиям вакансии, каков ваш карьерный путь и уровень коммуникативных навыков. Ожидайте вопросов о содержании резюме, интересе к компании и ваших профессиональных целях.
2. **Техническое задание.** Следующий этап — проверка технических знаний. Это может быть онлайн-тест на платформах вроде HackerRank или Codility либо встреча в офисе с техническим специалистом. Для .NET-позиций задания чаще всего охватывают задачи по языку C# и следующие темы.
  - **Базовые и продвинутые концепции программирования.** Принципы ООП, структуры данных, алгоритмы, системное мышление и архитектурные основы.
  - **Знание платформы .NET.** Понимание CLR, языка C#, ASP.NET Core, Entity Framework Core и LINQ. Обязательно освежите все темы, рассмотренные в моей книге *C# 12 and .NET 8 — Modern Cross-Platform Development Fundamentals*.
  - **Отладка, тестирование и оптимизация.** Умение диагностировать ошибки, писать читаемый и производительный код. Эта книга крайне важна для изучения таких тем.
3. **Техническое собеседование в офисе.** Обычно этот этап включает серию собеседований с участием разработчиков, технических руководителей и менеджеров. Вас могут ждать технические собеседования, обсуждение архитектурных решений, а иногда — сессии парного программирования. На позиции .NET-разработчика стоит ожидать следующего.
  - **Углубленные технические вопросы.** Готовьтесь к вопросам по .NET, C#, а также смежным областям — например, SQL Server или фронтенд-технологиям (если рассматривается позиция фулстек-разработчика).
  - **Проектирование систем.** Особенно важно для старших позиций. Рекрутеры ожидают, что вы сможете предложить архитектурное решение, учитывая масштабируемость, надежность и производительность.
  - **Оценка гибких навыков.** Ваш подход к решению задач, коммуникация, умение работать в команде и под давлением — все это имеет значение.
  - **Обсуждение опыта.** Часто начинают с открытого вопроса вроде «Расскажите о проекте, которым вы особенно гордитесь». Это ваш шанс задать тон беседе — выделите сильный опыт, на который рекрутер сможет опереться при уточняющих вопросах.
4. **Проверка на соответствие корпоративной культуре и поведенческое собеседование.** Этот этап помогает понять, насколько вы подходите команде. Вас могут спросить, как вы реагируете на критику, как справляетесь с конфликтами, что делаете при изменении приоритетов, каким образом адаптируетесь к новой информации или нестабильной обстановке.

## Не пользуйтесь нейросетью во время собеседования

Использовать LLM до собеседования вполне допустимо: с их помощью можно отточить резюме, освежить знания, подготовить ответы и даже отрепетировать собеседование. Но в процессе собеседования полагаться на LLM нельзя — и неэтично, и рискованно.

С развитием онлайн-собеседований облазн использовать подсказки от ИИ в режиме реального времени стал довольно распространенным. Некоторые кандидаты открывают Zoom в одном окне, ChatGPT — в другом, получив вопрос, делают паузу: «*Дайте подумать...*» — и быстро вводят запрос в чат. Через пару секунд они зачитывают полученный ответ.

Не стоит так поступать. Это заметно. И это ставит под сомнение не только вашу честность, но и профессиональную пригодность.

Как-то я работал инструктором в компании, которая проводила интенсивные курсы подготовки к сертификационным экзаменам Microsoft для разработчиков. На одном из занятий присутствовали 20 человек, и в последнем ряду, в углу, куда не так просто было подойти и заглянуть в экран, сидел участник, который в течение дня задавал странные, специфические вопросы, не относящиеся к теме. Во время перерыва один из студентов, сидевших рядом, подошел ко мне и сообщил, что тот участник читает нелегально выложенные в интернете списки реальных экзаменационных вопросов и ответов для сертификационных тестов Microsoft.

В результате мне пришлось удалить наглеца с курса и отказать в допуске к финальному экзамену. Подобное поведение — нарушение не только профессиональной этики, но и базовых принципов честности. И, что характерно, такие попытки обмана редко бывают успешными — по-настоящему сильные специалисты так не поступают.



**Внимание!** Модели вроде ChatGPT действительно полезны — но только в сочетании с вашим собственным мышлением и опытом. Если вы превращаетесь в «передаточное звено» между рекрутером и ИИ, возникает вопрос: а в чем тогда ваша ценность? Если все, что вы можете, — это переформулировать ответы модели, вас заменит не человек, а алгоритм. Работайте с ИИ как с инструментом: дополняйте, развивайте, проверяйте себя. Используйте его, чтобы учиться — а не чтобы отключаться.

## Собеседования в крупных компаниях

Проходит собеседование в технологических гигантах — таких как Microsoft, Amazon, Google, Apple, Яндекс — или в крупных игровых студиях — это особый опыт. У каждой из этих компаний своя культура, критерии оценки и акценты в подборе кандидатов. В табл. 19.2 приведены советы и наблюдения, которые помогут вам лучше подготовиться.

**Таблица 19.2.** Особенности собеседований в крупных компаниях

Компания	На что обратить внимание
Microsoft	Microsoft ценит командную работу и способность эффективно взаимодействовать с другими. Будьте готовы рассказать о проектах, в которых участвовали, и о своем вкладе в общий успех. Знание экосистемы Microsoft и ее продуктов — таких как Azure, Office 365 и Windows — может стать весомым плюсом, особенно если вы умеете показать, как применяли эти технологии на практике. Многие кандидаты отмечают, что на собеседованиях в Microsoft особенно важен сторителлинг. Один из них поделился, что структурировал рассказ о своем опыте в формате «проблема — решение — результат», — и это помогло удержать внимание рекрутеров и продемонстрировать глубину участия в проекте
Amazon	Компания Amazon известна своими 16 принципами лидерства. Будьте готовы привести примеры из своего опыта, которые соответствуют этим принципам. Многие вопросы будут касаться ориентированности на клиента, поэтому вспомните ситуации, когда вы делали все возможное, чтобы удовлетворить его потребности. В процессе собеседования может участвовать Bar Raiser — человек, отвечающий за высокие стандарты найма. Чтобы произвести впечатление, покажите, как вы решаете сложные задачи, и подчеркните готовность брать на себя ответственность
Google	В Google уделяют особое внимание вашим навыкам программирования и решению задач с использованием алгоритмов и структур данных. Практика на таких платформах, как LeetCode, может оказаться очень полезной. Ожидается глубокие технические знания в той области, в которой вы претендуете на должность — будь то проектирование систем, искусственный интеллект или веб-разработка. Подчеркните умение брать на себя ответственность за проекты и вести их до результата — в Google особенно ценят инициативность и предпринимательский подход даже в рамках инженерных ролей
Apple	В компании Apple ценят инновационное мышление и внимание к деталям. Расскажите о случаях, когда вам удавалось предложить нестандартные решения или создать продукт с продуманным пользовательским опытом. Компания особенно гордится интуитивностью своих решений, поэтому покажите, что вы понимаете, как создаются такие продукты, — и разделяете это стремление к качеству. Если позиция связана с облаком или пользовательскими данными, будет плюсом продемонстрировать понимание принципов конфиденциальности и защиты данных. Упоминание о вашем интересе к экосистеме Apple и осведомленность о сильных сторонах ее продуктов помогут лучше выстроить контакт с рекрутерами

Продолжение ↗

**Таблица 19.2 (окончание)**

<b>Компания</b>	<b>На что обратить внимание</b>
<b>Яндекс</b>	В Яндексе ценятся инженерная зрелость, ответственность и способность быстро адаптироваться к изменениям. Ожидается уверенное знания алгоритмов, структур данных и понимание архитектуры ПО. Вопросы могут касаться не только кода, но и реального влияния вашей работы: улучшение пользовательского опыта, ускорение процессов или оптимизация решений. Приветствуется активное участие в обсуждении
<b>Лаборатория Касперского</b>	В Лаборатории Касперского особенно важны фундаментальные знания в области информационной безопасности и устойчивые технические навыки. Будьте готовы к строгим вопросам по C++, архитектуре систем и практикам безопасной разработки. Также ценятся внимательность к деталям, способность к критическому мышлению и умение работать с чувствительной информацией
<b>Игровые студии</b>	Компании, разрабатывающие видеоигры, такие как Blizzard, Electronic Arts и Riot Games, особенно ценят страсть к играм и понимание своей аудитории. Обязательно покажите, что вы знакомы с продуктами студии и искренне интересуетесь их развитием. Идеи по улучшению геймплея или пользовательского опыта будут восприняты с интересом. Техническая подготовка также критична: в зависимости от студии потребуется уверенное владение C++, Unity или другими игровыми движками. Будьте готовы к вопросам по специфике разработки — например, алгоритмы поиска пути, обработка столкновений, игровая физика и оптимизация рендеринга. Кроме технических навыков, важна способность работать в команде. Культурное соответствие и командный дух — ключевые качества, на которые обращают внимание в игровых студиях

Несколько универсальных советов, которые пригодятся независимо от компании.

- **Практикуйтесь в программировании «на доске».** Многие компании по-прежнему используют формат, в котором нужно писать код от руки — на доске или бумаге. Репетиции в таком формате помогут чувствовать себя увереннее.
- **Порепетируйте собеседование.** Попросите коллегу или наставника провести пробное собеседование, особенно если у него есть опыт работы в этих компаниях. Так вы лучше поймете стиль и глубину вопросов.
- **Оставайтесь спокойны и доброжелательны.** Собеседования могут быть интенсивными. Спокойствие, открытость и дружелюбие помогут вам сосредоточиться, а рекрутерам — почувствовать, что с вами приятно работать.

Каждое собеседование — это не только проверка, но и возможность. Даже если результат окажется неудачным, вы приобретете опыт, лучше поймете, чего ждут компаний, и сделаете следующий шаг увереннее.

## Подача заявок на руководящие и старшие должности

Когда разработчики с опытом претендуют на позиции уровня техлида, старшего инженера или архитектора в ведущих ИТ-компаниях, процесс собеседования становится глубже и сложнее. Ожидается не только техническая экспертиза, но и зрелость в принятии решений, умение вести команду и влиять на развитие продукта.

Вот несколько ключевых тем, на которые обращают внимание при найме на такие роли.

### ● **Лидерство и влияние**

- **Управление командой.** Ожидайте вопросов о вашем опыте руководства, наставничества и координации межфункциональных проектов. Вам важно показать, что вы умеете не просто решать задачи самостоятельно, а добиваться результата через других — вдохновляя, направляя и поддерживая команду.
- **Разрешение конфликтов.** Рекрутеры часто интересуются ситуациями, где вам приходилось справляться с разногласиями или принимать непростые решения. Их цель — понять, как вы балансируете между технической точностью, дедлайнами и отношениями в команде.
- **Инициативность и изменения.** Кандидаты на старшие роли должны уметь быть агентами перемен — предлагать и внедрять улучшения, будь то технологические решения, процессы или командные практики. Будьте готовы рассказать, как вы инициировали перемены и довели их до результата.

### ● **Углубленные знания архитектуры**

- **Проектирование систем.** Для руководящих ролей умение проектировать надежные и масштабируемые системы — ключевой навык. Будьте готовы к заданиям, в которых нужно разработать архитектуру «с нуля» или проанализировать сильные и слабые стороны существующего решения.
- **Масштабируемость, производительность и безопасность.** На старших позициях важно не только писать рабочий код, но и понимать, как он поведет себя при высокой нагрузке. Вас могут спросить, как вы подходите к оптимизации, обеспечению отказоустойчивости и защите данных.
- **Выбор технологий.** Старшим разработчикам часто доверяют выбор технологий, соответствующих целям проекта. Вас могут спросить, по каким критериям вы оцениваете новые инструменты или фреймворки, как принимаете решения о внедрении и как организуете их интеграцию в существующую архитектуру.

### ● **Стратегическое мышление**

- **Видение и технологическая стратегия.** Ожидайте вопросов о вашем подходе к долгосрочному развитию продукта или архитектуры. Рекрутеры могут

спросить, как вы выстраивали стратегию, предвидели технологические или рыночные тренды, согласовывали технические решения с бизнес-целями и управляли жизненным циклом продукта.

- **Управление ресурсами и приоритизация.** Важно показать, как вы принимаете решения в условиях ограниченных ресурсов: распределяете задачи, формируете приоритеты, работаете с бюджетом и выстраиваете баланс между быстрыми результатами и долгосрочной устойчивостью решений.

#### ● Техническое наставничество

- **Обмен знаниями.** От старших разработчиков ожидается, что они будут не только решать сложные задачи, но и делиться опытом с другими. Вас могут спросить, как вы помогали коллегам развиваться, обучали новым подходам или помогали быстрее вливаться в работу. Примеры менорства и поддержки команды — важная часть вашей профессиональной репутации.
- **Продвижение инженерной культуры.** Важно показать, как вы внедряете лучшие практики: в коде, тестировании, обеспечении безопасности, работе с документацией. Вас могут попросить привести случаи, когда вы улучшали процессы, настраивали CI/CD или внедряли подходы, повышающие устойчивость и читаемость кода в команде.

#### ● Влияние и результаты

- **Измеримые достижения.** Старшие специалисты должны уметь говорить не только о проделанной работе, но и о ее эффекте. Подготовьте примеры, где ваш вклад привел к росту производительности, снижению издержек, ускорению релизов или увеличению прибыли.
- **Управление проектами.** Если вы управляли проектами, особенно в рамках Agile-процессов, расскажите, как строили план, следили за сроками, работали с рисками и вели команду к результату. Рекрутеры хотят услышать, как ваш подход помог команде уложиться в дедлайн, сохранить качество и добиться поставленных целей.

#### ● Культурный вклад

- **Соответствие ценностям и стилю компании.** На руководящих позицияхкультура — это не фон, а важный элемент работы. Вас могут оценивать не только по опыту, но и по тому, насколько ваш подход к работе, принятию решений и взаимодействию с командой совпадает с культурой компании. Будьте готовы показать, как вы вписывались в команду и влияли на атмосферу.
- **Инновационность.** Компании ищут не просто опытных, а инициативных специалистов, способных предлагать новые подходы, улучшать процессы и вдохновлять на перемены. Подумайте, какие идеи вы предлагали и как они были реализованы. Даже небольшие, но работающие улучшения — весомый аргумент.

Советы по подготовке к собеседованиям на руководящие должности.

- **Подготовьте истории с акцентом на результат.** Соберите несколько ярких примеров из своего опыта, где вы продемонстрировали лидерство, принятие решений и влияние на итог. В каждой истории должны быть четко видны контекст, ваши действия, вызовы и достигнутые результаты.
- **Оставайтесь в курсе технологий.** Даже если у вас богатый опыт, важно показывать, что вы продолжаете развиваться. Знание новых подходов, фреймворков и тенденций говорит о вашей вовлеченности и способности адаптироваться.
- **Тренируйтесь обсуждать лидерство.** Проведите пробные собеседования с упором на вопросы о стратегии, управлении людьми и принятии решений. Это поможет выработать четкие и уверенные ответы.

Собеседование на руководящую должность — это возможность продемонстрировать не только техническую экспертизу, но и лидерское мышление. Важно показать, что вы умеете не просто выполнять задачи, а формировать направление, развивать команду и двигать компанию вперед — с пониманием стратегии и перспектив роста.

## Путь к должности тестировщика или QA-специалиста

Соискание должности тестировщика или специалиста по обеспечению качества (QA) требует особого подхода, отличающегося от подготовки к роли разработчика. Здесь на первый план выходят внимательность к деталям, аналитическое мышление и понимание методологий тестирования. Технические навыки по-прежнему важны, но акценты смешаются. Вот что важно учесть, если вы готовитесь к роли тестировщика.

- **Знание методологий тестирования**
  - **Разберитесь в подходах.** Изучите основные методологии: классические (например, каскадная модель), гибкие (Agile), а также современные практики, такие как разработка через тестирование (Test-Driven Development, TDD) и разработка через поведение (Behavior-Driven Development, BDD).
  - **Рассмотрите возможность сертификации.** Сертификаты, такие как ISTQB (International Software Testing Qualifications Board) и CSTE (Certified Software Tester), помогут подтвердить вашу квалификацию и продемонстрировать серьезный интерес к профессии.
- **Навыки внимательности и точности**
  - **Обнаружение ошибок.** Тестировщику важно уметь замечать даже мельчайшие отклонения от ожидаемого поведения. Тренируйте внимательность, просматривая интерфейсы, сценарии использования и код — ищите непоследовательности, визуальные или логические ошибки, которые могли бы ускользнуть от разработчика.

- **Документирование.** Навыки создания ясной и точной документации — ключевая часть работы. Умение составлять понятные отчеты об ошибках, тест-кейсы и планы тестирования поможет вашей команде быстрее устранять проблемы и делать продукт стабильнее.
- **Техническая подготовка**
  - **Овладение инструментами.** Освойте популярные инструменты тестирования, такие как Playwright, Postman, Jira и др. Умение писать автоматизированные тесты с помощью сценарных языков, например Python или JavaScript, будет большим преимуществом, особенно если в компании практикуется автоматизированное тестирование.
  - **Понимание технологического стека.** Даже если вы не пишете основной код, важно понимать, как работает система. Например, при тестировании веб-приложений полезно знать основы HTML, CSS и JavaScript — это поможет эффективнее находить и описывать ошибки.
- **Гибкие навыки**
  - **Коммуникация.** Четкое и понятное общение — одна из важнейших составляющих работы тестировщика. Вам нужно будет грамотно описывать ошибки, уметь объяснить разработчику, как воспроизвести проблему, и убедиться, что вас правильно поняли.
  - **Решение задач.** На собеседовании могут предложить сценарий, например: «Как бы вы протестировали такую-то функцию?» или «Что вы сделали бы, если баг возникает не каждый раз?» Покажите, что умеете мыслить логично, находить причины проблем и предлагать практические способы их выявления.
- **Подготовка к вопросам на собеседовании**
  - **Практические задания.** Будьте готовы к демонстрации своих навыков: вас могут попросить найти ошибки в коде, составить тест-кейсы по заданному сценарию или проанализировать часть интерфейса. Ценится не только правильный ответ, но и ваш ход рассуждений.
  - **Сценарные вопросы.** Рекрутер может задать вопрос вроде «Как бы вы протестирували онлайн-форму регистрации?» — и ожидает от вас последовательного, логичного подхода. Подготовьтесь описывать свои действия поэтапно: от анализа требований до тестирования краевых случаев.
- **Продемонстрируйте свой опыт в тестировании**
  - **Портфолио выявленных ошибок.** Если у вас есть такая возможность, собираите примеры наиболее значимых дефектов, которые вы обнаружили в прошлых проектах. Подчеркните, как ваши находки позитивно повлияли на стабильность и качество продукта.
  - **Оценка влияния вашей работы.** Говорите о том, как ваше тестирование реально помогло команде: уменьшило количество багов в продакшене, ускорило выпуск обновлений, повысило удовлетворенность пользователей или позволило принять более обоснованные технические решения.

- **Демонстрируйте мышление, ориентированное на качество**
  - **Фокус на качестве продукта.** Подчеркните, что ваша работа направлена не только на поиск багов, но и на повышение общей надежности и удобства использования продукта. Расскажите, как вы отстаивали интересы пользователей и помогали улучшать качество проекта.
  - **Проактивность.** Продемонстрируйте способность предвидеть возможные точки отказа еще до появления проблем. Упомяните, как заранее предложенное вами тестирование помогло снизить риски и избежать ошибок в продакшене.
- **Сфокусируйтесь на понимании продукта и пользователя**
  - **Тестирование с ориентацией на пользователя.** Покажите, что вы учитываете реальные сценарии использования и стремитесь обеспечить удобство и надежность продукта для конечного пользователя. Расскажите, как вы адаптируете свои тестовые сценарии под реальные потребности.
  - **Знание продуктов компании.** Перед собеседованием изучите продуктовую линейку потенциального работодателя. Это поможет вам предложить разумные идеи по улучшению качества и целенаправленные подходы к тестированию.

Тестировщики играют ключевую роль в разработке ПО, отвечая за стабильность, качество и производительность конечного продукта. Сфокусировавшись на специализированных навыках, инструментальных знаниях и точности мышления, вы сможете не только успешно подготовиться к собеседованию, но и убедительно продемонстрировать свою ценность как специалиста, который помогает создавать действительно надежные решения.

## Выбор вопросов для собеседования

Вопросы, которые вам зададут на собеседовании, обычно подбираются так, чтобы оценить как технические знания, так и ваше соответствие культуре компании. Вот на что чаще всего ориентируются работодатели.

- **Соответствие должности.** Вопросы подбираются с учетом специфики вакансии и требуемых навыков. Если вы претендуете на должность в .NET-разработке, ждите акцента на экосистему .NET, программирование на C# и работу с популярными библиотеками и инструментами.
- **Навыки решения задач.** Работодатели часто включают задания, которые проверяют вашу способность решать задачи в стрессовой ситуации. Это помогает не только увидеть, как вы находите решения, но и понять, как вы рассуждаете, формулируете мысли и объясняете свои действия.
- **Потенциал роста.** Некоторые вопросы направлены на то, чтобы оценить вашу способность к обучению и развитию. Это особенно важно в условиях стремительно меняющихся технологий, где адаптивность ценится не меньше, чем текущие знания.

- **Культурное соответствие.** Работодатели также стремятся понять, насколько вы вписываетесь в команду и разделяете ценности компании. В таких вопросах внимание акцентируется не только на вашем опыте, но и на личных качествах. Рекрутеры часто обращают внимание на то, «горит ли у вас взгляд» и легко ли с вами общаться. Один из простых, но показательных вопросов: «Расскажите о том, чем вы особенно гордитесь».

Вот несколько советов по подготовке к техническому собеседованию.

- **Повторите основы.** Убедитесь, что у вас крепкие знания по C#, .NET и распространенным фреймворкам.
- **Практикуйтесь в решении задач.** Используйте платформы вроде LeetCode или CodeSignal, чтобы отточить навыки программирования.
- **Изучите технологии компании.** Разберитесь в технологическом стеке и инструментах, которые применяются в компании, в которую вы подаете заявку.
- **Проводите пробные собеседования.** Обратитесь к коллегам, наставникам или воспользуйтесь онлайн-ресурсами, чтобы провести тренировочное собеседование и получить конструктивную обратную связь.

Понимание структуры и целей технического собеседования поможет вам лучше подготовиться, чувствовать себя увереннее и показать себя как сильного и разностороннего кандидата на должность .NET-разработчика.

## Поведенческие вопросы

Вопросы поведенческого типа — это обычная часть собеседований, особенно если должность предполагает тесное взаимодействие с командой, принятие решений или разрешение конфликтов. Такие вопросы помогают работодателю оценить, как вы будете вести себя в аналогичных ситуациях в будущем, опираясь на ваш прошлый опыт.

Качество ваших ответов играет ключевую роль — от этого во многом зависит, сочтут ли вас подходящим кандидатом.

Ниже приведены три популярных поведенческих вопроса. Для каждого из них показаны один неудачный и два удачных варианта ответа, чтобы вы лучше поняли, как можно усилить впечатление.

### 1. Расскажите о случае, когда вы столкнулись с серьезной проблемой на работе. Как вы с ней справились?

Этот вопрос помогает оценить, как вы решаете сложные задачи и насколько хорошо справляетесь с давлением. Рекрутеров интересует, как вы подходите к непредвиденным трудностям, насколько вы упорны, изобретательны и умеете работать в команде.

**Неудачный ответ:** «Я не припомню серьезных проблем. Обычно просто делаю свою работу, и все идет по плану».

**Почему это плохо:** такой ответ может создать впечатление, что у вас мало опыта в нестандартных ситуациях или что вы не склонны брать на себя ответственность в сложные моменты.

**Удачный ответ:** «В одном из проектов у нас резко сократилось время на реализацию ключевой функции, при этом команда была перегружена. Я предложил план перераспределения задач, взял на себя часть чужой нагрузки и договорился о ежедневных коротких встречах, чтобы держать всех в курсе. Благодаря слаженной работе мы сдали проект на два дня раньше срока, и заказчик был доволен».

**Почему это хорошо:** показывает инициативность, умение принимать решения в условиях ограниченного времени и работать с командой под давлением.

## 2. Опишите ситуацию, когда вам приходилось тесно работать с человеком, характер которого сильно отличался от вашего

Этот вопрос помогает работодателю оценить вашу способность эффективно взаимодействовать с разными людьми. Рекрутеров интересует, как вы адаптируетесь к стилям общения и работы, отличным от ваших, как решаете конфликты и выстраиваете продуктивное сотрудничество. Важно продемонстрировать эмпатию, гибкость и зрелый подход к командной работе.

**Неудачный ответ:** «У меня был коллега, с которым было трудно работать из-за его отношения. Я просто старался избегать общения с ним и выполнял свою часть работы самостоятельно».

**Почему это плохо:** такой ответ показывает, что вы не стремитесь решать проблемы во взаимодействии и не умеете адаптироваться в сложных межличностных ситуациях.

**Удачный ответ 1:** «Однажды мне пришлось работать с коллегой, который был очень эмоциональным и склонным к спонтанным решениям, тогда как я предпочитаю планирование и структурированный подход. Сначала это вызывало напряжение, но со временем мы нашли общий ритм: я предложил проводить короткие регулярные встречи, чтобы планировать ключевые этапы, а он привнес гибкость в наши процессы. Мы многому научились друг у друга, а результат оказался успешным — проект завершили вовремя и с хорошим качеством».

**Почему это хорошо:** показывает, что вы умеете адаптироваться, ищете компромиссы и можете превратить различия в командное преимущество.

**Удачный ответ 2:** «Был случай, когда мой коллега предпочитал письменное общение, особенно по электронной почте, в то время как я привык обсуждать задачи устно. Я предложил гибкий подход — еженедельные личные встречи плюс коммуникация по почте в остальное время. Такой формат устроил нас обоих, улучшил координацию и укрепил рабочие отношения».

**Почему это хорошо:** демонстрирует коммуникативные навыки, инициативу, уважение к чужому стилю работы и способность находить рабочие компромиссы.

### 3. Можете привести пример цели, которую вы не достигли, и как вы справились с этой ситуацией?

Цель этого вопроса — оценить вашу устойчивость, способность к самоанализу и отношение к ошибкам. Рекрутеры хотят понять, умеете ли вы честно признавать промахи, извлекать из них уроки и восстанавливаться. Хороший ответ демонстрирует ответственность, зрелость и готовность к росту.

**Неудачный ответ:** «Однажды я с треском провалил план продаж. Думаю, я просто смирился с этим — иногда выигрываешь, иногда проигрываешь».

**Почему это плохо:** такой ответ показывает отсутствие ответственности и стремления к улучшению. Нет анализа ошибок, действий по исправлению или выводов на будущее.

**Удачный ответ 1:** «В прошлом году я поставил перед собой цель — увеличить квартальные продажи на 20 %, но недобрал около 5 %. Проанализировав ситуацию, понял, что не учел изменившиеся потребительские предпочтения. Это подтолкнуло меня пройти курс по цифровому маркетингу и пересмотреть стратегию. В следующем квартале мне удалось превзойти исходную цель — рост составил 30 %».

**Почему это хорошо:** показывает честность, способность анализировать ошибки, учиться и адаптироваться. Такой подход говорит о зрелости и ориентированности на результат.

**Удачный ответ 2:** «На прошлой работе мне поручили руководить запуском нового проекта, но он был отложен из-за внешних факторов. Я использовал это время, чтобы провести дополнительные собеседования с пользователями и внести улучшения в стратегию. В результате запуск прошел гладко, а продукт получил высокую оценку за удобство и качество».

**Почему это хорошо:** такой ответ подчеркивает стратегическое мышление, умениеправляться с неопределенностью и превращать неудачи в точки роста.



Поиск работы — это всегда испытание. Иногда он может казаться изматывающим и деморализующим. Но важно помнить: многие компании действительно заинтересованы в том, чтобы найти сильного кандидата. Они хотят, чтобы вы прошли собеседование и заняли вакансию, — просто потому, что это позволит им вернуться к своей основной работе. Рекрутеры не ищут поводов отказать — они ищут поводы сказать «да».

## Метод STAR

При ответе на поведенческие вопросы особенно важно не только перечислить факты, но и показать ход мыслей, конкретные действия и извлеченные уроки.

Чтобы структурировать ответы и сделать их максимально понятными и убедительными, удобно использовать метод *STAR* — Ситуация (Situation), Задача (Task), Действие (Action), Результат (Result). Этот подход помогает избежать размытых

и неструктурированных ответов, позволяя вам четко донести свою мысль и подчеркнуть ключевые компетенции.

Метод STAR помогает структурировать ответы на поведенческие вопросы собеседования, разбивая рассказ на четыре части: *ситуация, задача, действие и результат*. Такой подход позволяет вам четко и лаконично описать контекст, пояснить, какую цель вы преследовали, какие шаги предприняли и к какому результату это привело. Благодаря этому рекрутеру проще понять, как вы справляетесь с задачами и какие качества проявляете в рабочих ситуациях. В каждом ответе важно:

- кратко описать ситуацию;
- обозначить задачу, которую необходимо было решить;
- подробно рассказать о действиях, которые вы предприняли;
- объяснить результат этих действий.

Такой подход помогает структурировать ответ и показать рекрутеру, как вы размышляете, действуете и добиваетесь результатов в реальных рабочих ситуациях. Ниже приведено описание каждого компонента метода.

## 1. Ситуация

Начните с краткого описания контекста. Обозначьте, где и при каких обстоятельствах происходили события, и покажите, почему ситуация была значимой. Главное — быть конкретным, но не вдаваться в ненужные подробности. Рекрутеру должно быть понятно, в какой обстановке возникла задача или проблема, о которой пойдет речь.

*Пример:* «На предыдущей должности руководителя проекта мы столкнулись с необходимостью выпустить новый продукт в ограниченные сроки из-за неожиданного роста спроса на рынке».

## 2. Задание

После того как вы описали ситуацию, уточните, какая именно задача перед вами стояла. Расскажите, в чем заключалась ваша личная ответственность, какие цели нужно было достичь и кто был вовлечен в процесс. Этот раздел должен подчеркнуть вашу роль и то, что именно вы должны были сделать.

*Пример:* «Я отвечал за координацию работы между командами разработки ПО, маркетинга и продаж, чтобы все аспекты запуска были синхронизированы и проект уложился в отведенные два месяца».

## 3. Действие

Здесь вы подробно рассказываете, какие конкретные шаги предприняли для решения задачи. Главное — сосредоточиться на своих действиях, а не на командных усилиях в целом. Это возможность продемонстрировать свои навыки, инициативность

и подход к решению проблем. Упомяните, если использовали нестандартные или особенно эффективные подходы.

*Пример:* «Я организовал еженедельные встречи между отделами, чтобы наладить коммуникацию и своевременно устранять узкие места. Кроме того, я внедрил информационную панель в реальном времени для отслеживания прогресса и создал группу быстрого реагирования для решения срочных вопросов».

#### 4. Результат

Завершите рассказ описанием последствий ваших действий. Покажите, каких результатов удалось достичь благодаря вашим усилиям, — особенно ценно, если можно привести количественные показатели. Упомяните, какие уроки вы извлекли и было ли получено признание со стороны руководства или коллег. Этот раздел подтверждает значимость вашего вклада и демонстрирует вашу пользу для компании.

*Пример:* «Продукт был выпущен в срок и превзошел прогнозы продаж на 15 %. Этот успех стал причиной моего назначения на руководящую должность, а информационная панель, которую я предложил, была внедрена как стандартный инструмент в будущих проектах компании».

### Эффективное применение метода STAR

Чтобы метод STAR действительно работал на вас, важно соблюдать несколько простых принципов.

- **Говорите по существу.** Подробности важны, но постарайтесь быть конкретным и не перегружать рассказ лишними деталями.
- **Будьте лаконичны.** Держите ответ в рамках — чтобы рекрутер не терял интерес. Иногда уместно задать уточняющий вопрос вроде: «Хотите, расскажу подробнее?»
- **Подчеркивайте личный вклад.** Даже если вы работали в команде, акцентируйте внимание на том, какую роль сыграли именно вы.
- **Выбирайте истории с учетом ситуации.** Подбирайте примеры, которые наиболее точно соответствуют вакансии и ценностям компании. Подчеркивайте навыки, которые особенно важны в этой роли.

Четко структурированные ответы по методу STAR помогут вам ясно и убедительно донести свои сильные стороны, показать профессионализм и оставить у рекрутера положительное впечатление.

### Советы во время собеседования

Помните: и у вас, и у рекрутера ограничено время. Цель собеседования — дать рекрутеру возможность убедиться, что вы разбираетесь в теме. Если вас спрашивают о чем-то, в чем вы сильны, легко поддаться искушению рассказывать слишком много. Вам комфортно, вы увлечены — и вы говорите, говорите,

те. Но важно уметь «читать» собеседника. Дайте ему пространство для перехода к другим вопросам. Например, после пары минут рассказа можно сказать: «Если хотите, могу рассказать подробнее». Это даст рекрутёру возможность попросить продолжить, перейти к более узкой теме в рамках вопроса или вовсе сменить тему. Такой подход показывает, что вы уважаете формат встречи, внимательны ко времени и умеете вести диалог.

Некоторые из наиболее распространенных вопросов, которые вам могут задать на собеседовании, приведены в табл. 19.3.

**Таблица 19.3.** Примеры распространенных вопросов общего характера на собеседовании

Вопрос	Интерпретация ответа
Расскажите о проекте, которым вы особенно гордитесь. Каков был ваш вклад?	Помогает понять, какие задачи вы считаете значимыми и какую роль сыграли в успехе команды или проекта
Почему вы решили подать заявку на эту должность?	Позволяет выяснить, насколько ваши ожидания совпадают с возможностями компании и мотивами перехода
Как вы справляетесь со стрессом и давлением, особенно в условиях дедлайна или аврала?	Помогает оценить ваши способы реагирования на стресс и адаптацию к интенсивной работе
Расскажите о своих сильных и слабых качествах	Помогает понять, насколько вы осознаете свои сильные стороны и занимаетесь ли развитием тех навыков, которые требуют внимания
Как вы представляете свою карьеру через пять лет?	Показывает, к чему вы стремитесь в долгосрочной перспективе и как вы вписываетесь в планы компании
Вспомните ситуацию, когда с кем-то из коллег было сложно найти общий язык. Как вы решили эту проблему?	Помогает оценить ваши коммуникативные навыки, умение находить подход к людям и адаптироваться в сложных ситуациях
Как вы расставляете приоритеты, когда работаете сразу над несколькими задачами?	Рекрутёр узнает, насколько эффективно вы управляете временем, расставляете приоритеты и справляетесь с высокой нагрузкой
Чем вам нравится заниматься вне работы?	Помогает понять, есть ли у вас общие интересы с командой и насколько легко вы встроитесь в коллектив
В какой рабочей обстановке вы чувствуете себя наиболее продуктивно?	Ответ покажет, насколько вам подойдет культура и стиль работы в компании
Хотите что-нибудь узнать о нас или о будущей роли?	Ваши вопросы говорят о том, насколько вы вовлечены, как глубоко интересуетесь работой и насколько серьезно подходите к выбору компании

Эти вопросы помогают получить развернутые ответы и составить объективное представление о навыках кандидата, его характере и потенциальной пригодности для работы в компании.



**Дельный совет:** подготовьте для рекрутера несколько продуманных вопросов — не ограничивайтесь только темами зарплаты и соцпакета!

Разумные и уместные вопросы в конце собеседования играют важную роль. Они показывают вашу заинтересованность в вакансии и помогают лучше понять, насколько эта должность отвечает вашим ожиданиям и карьерным целям. В табл. 19.4 приведено несколько примеров таких вопросов.

**Таблица 19.4.** Распространенные вопросы общего характера для рекрутера

Вопрос	Основание
Как выглядит обычный рабочий день на этой должности?	Поможет вам понять повседневные задачи и объем работы, который вас ожидает
Какие проекты нужно будет решать в первую очередь?	Даст представление о том, как будут проходить ваши первые недели или месяцы в компании и какие задачи наиболее важны
Какие качества особенно важны в этой работе?	Узнав, что ценит компания в сотрудниках, вы сможете понять, насколько ваши навыки и личные качества подходят под эту вакансию
Как в компании помогают сотрудникам развиваться и учиться?	Показывает, что вы заинтересованы в развитии и хотите строить долгосрочную карьеру
По каким критериям оценивают результаты работы на этой позиции?	Поможет понять, какие требования ставит компания и какие результаты от вас ждут
Какие возможности для карьерного роста есть в компании?	Позволяет узнать о перспективах роста и возможностях продвижения в компании
Можете рассказать о команде, с которой предстоит работать?	Вы узнаете о размерах команды, ее структуре и атмосфере — это важно, чтобы понять, насколько комфортно вам будет
Как в компании организована работа в команде?	Даст понять, насколько стиль совместной работы в организации соответствует вашим предпочтениям
Какие ценности особенно важны в компании и как это заметно в работе?	Поможет оценить, совпадают ли ценности компании с вашими личными и профессиональными взглядами

<b>Вопрос</b>	<b>Основание</b>
Как у вас принято давать обратную связь?	Регулярная обратная связь очень важна для профессионального роста, и знание ее механизма покажет, подходит ли вам такая рабочая среда
Какие задачи сейчас самые сложные для компании?	Позволит понять, с какими сложностями вы можете столкнуться и как компания планирует их решать
Какой вы видите компанию через пять лет?	Демонстрирует ваш интерес к будущему компании и к той роли, которую вы можете сыграть в ее развитии
Чем ваша компания отличается от конкурентов?	Поможет понять рыночное положение компании и ее конкурентные преимущества, а также стабильность и способность к инновациям
Нужно ли вам что-то еще от меня (например, рекомендации или портфолио)?	Показывает вашу готовность двигаться вперед и убедиться, что у рекрутера есть все необходимое для принятия решения
Какие следующие этапы собеседования мне нужно пройти?	Помогает вам разобраться в дальнейших шагах и сроках, чтобы подготовиться к следующему этапу



Будьте готовы, что рекрутер может переадресовать вопрос обратно! Например, если вы спросите: «Какие качества особенно важны для успеха в этой роли?», он может ответить вопросом: «А как бы вы ответили на этот вопрос?» Подготовьтесь заранее ответить и на такой вопрос.

Хорошие вопросы не только показывают ваш энтузиазм по отношению к компании и роли, но и помогают получить важную информацию. Благодаря этому вы сможете принять осознанное решение, если получите предложение о работе.

## Как отвечать на сложные вопросы по программированию

На собеседованиях вы можете столкнуться с задачами по программированию, которые кажутся вам слишком сложными или невыполнимыми. Это обычная ситуация, особенно когда вы волнуетесь. Вот несколько полезных советов, как эффективно действовать в таких случаях.

- Сохраняйте спокойствие.** Постарайтесь не паниковать. Спокойствие помогает ясно мыслить и действовать эффективнее. Сделайте глубокий вдох и сосредоточьтесь на задаче.
- Рассказывайте вслух, как вы рассуждаете.** Даже если вы не уверены в решении, опишите свои мысли и то, как подошли бы к решению задачи. Рекрутерам важно понять, как именно вы мыслите, а не просто увидеть правильный ответ.

3. **Разбейте задачу на более мелкие части.** Если задача кажется слишком объемной, попробуйте разделить ее на простые шаги. Сосредоточьтесь на решении одной части задачи за раз — это поможет двигаться вперед постепенно.
4. **Задавайте уточняющие вопросы.** Не стесняйтесь уточнять условия или детали задачи. Так вы показываете внимание и способность тщательно разбираться, прежде чем предлагать решение.
5. **Расскажите о похожих задачах, которые вы уже решали.** Если вам попадалась похожая задача, упомяните ее и объясните, как вы могли бы применить прошлый опыт к текущей ситуации. Это показывает вашу способность использовать свои знания на практике.
6. **Упомяните, как вы используете ресурсы в работе.** Вполне нормально сказать, что в реальной ситуации вы обратились бы к документации, Stack Overflow или другим источникам. Такая честность делает ваш подход более реалистичным и показывает вашу готовность учиться.
7. **Попробуйте решить задачу, даже если сомневаетесь.** Лучше сделать попытку, написать псевдокод или хотя бы объяснить логику своих действий, чем совсем отказаться от решения. Покажите, как вы продолжили бы работу, если бы у вас было больше времени.
8. **Попросите обратную связь.** Если вы затрудняетесь ответить, не бойтесь попросить рекрутера высказать свое мнение или дать подсказку. Это превратит собеседование в возможность чему-то научиться и покажет, что вы готовы прислушиваться к советам.
9. **Проанализируйте, что было трудно, после собеседования.** После собеседования подумайте, что именно вызвало затруднения и почему. Регулярная практика и имитация условий собеседования помогут вам чувствовать себя увереннее.
10. **Вернитесь к задаче в благодарственном письме.** Если после собеседования вы нашли более удачное решение или что-то переосмыслили, можете упомянуть об этом в коротком благодарственном письме. Так вы подчеркнете свою готовность учиться и расти.

Сложные вопросы на технических собеседованиях встречаются часто — это не проблема, а возможность показать, как вы справляетесь с трудностями, решаете задачи и учитесь на опыте. Помните: рекрутерам важен ваш подход и настрой, а не только правильные ответы.



Во времена учебы в колледже — много лет назад — я устроился на подработку в компанию Talking Pages. Это была телефонная справочная служба: клиенты звонили нам, чтобы узнать контакты сантехников или других местных организаций. Одним из этапов собеседования был тест на знание британских географических названий. Мне проигрывали аудиозапись названия населенного пункта, а я должен был его правильно записать. Британия известна тем, что названия многих городов и деревень сильно отличаются от того, как они произносятся. Например, название Loughborough произносится как «Лафф-бо-ро», с ударением на первом слоге.

Короче, этот тест я провалил.

После собеседования я расстроился и какое-то время просто бесцельно бродил по центру города. Затем я подумал, что попрошу дать мне второй шанс. Я позвонил в агентство по трудоустройству и спросил, можно ли прийти на следующий день и попробовать снова. Купил дорожный атлас, выучил названия населенных пунктов, вернулся, пересдал тест и получил работу.

## Как вести себя, если рекрутер не подготовился

Иногда бывает так, что вы приходите на собеседование хорошо подготовленным, а рекрутер оказывается совсем не готов. Это не самый приятный и профессиональный подход, но такое случается чаще, чем кажется. К сожалению, некоторые компании просто не учат своих сотрудников грамотно проводить собеседования.

Конечно, общаться с неподготовленным рекрутером бывает сложно и некомфортно. Но то, как вы поведете себя в такой ситуации, может повлиять на исход собеседования и оставить о вас позитивное впечатление. Вот несколько подходов, которые помогут справиться с подобной ситуацией.

- **Сохраняйте спокойствие и позитив.** Независимо от поведения рекрутера держитесь вежливо и профессионально. Это покажет, что вы можете адекватно реагировать даже в не самых простых обстоятельствах.
- **Аккуратно направляйте разговор сами.** Если рекрутер выглядит растерянным или ведет беседу бессистемно, постарайтесь деликатно направить беседу в нужное русло. Например, скажите: «Я читал о вашем недавнем проекте в области X и, думаю, мой опыт работы с Y мог бы здесь пригодиться».
- **Мягко предложите помощь.** Если рекрутер явно потерял нить беседы или не знает, о чем спросить, вы можете вежливо предложить: «Может, мне стоит рассказать побольше о своем опыте в [определенной области]?» Это подтолкнет разговор в правильном направлении и не вызовет неудобства.
- **Задавайте уточняющие вопросы.** Если вопросы звучат непонятно или сформулированы неудачно, не стесняйтесь попросить уточнить. Например: «Чтобы лучше ответить, могли бы вы пояснить, что именно вы имеете в виду под...?»
- **Обратите внимание на культуру компании.** Подумайте, может ли поведение рекрутера отражать общую рабочую культуру и атмосферу в организации. Иногда неподготовленный собеседник может сигнализировать о том, что в компании есть организационные сложности или проблемы с уважением к сотрудникам.
- **Выясните, что будет дальше.** В конце беседы задайте вопросы о следующих шагах собеседования, чтобы понять, как будут развиваться события. Например: «Как будет выглядеть следующий этап собеседования?» или «С кем я могу связаться по дальнейшим вопросам?»
- **Сохраняйте профессионализм до конца.** После собеседования отправьте короткое благодарственное письмо. Так вы оставите положительное впечатление о себе, независимо от результата собеседования.

- **Решите, подходит ли вам этот работодатель.** После собеседования подумайте, была ли неподготовленность рекрутера случайностью или это показатель более серьезных проблем в компании. Если работа все равно кажется привлекательной, двигайтесь дальше, но если негативные моменты перевешивают, возможно, лучше поискать другие варианты.

Проходить собеседование с неподготовленным или неопытным рекрутером сложно, но это хорошая возможность продемонстрировать ваше терпение, гибкость и способность управлять нестандартными ситуациями. А эти качества важны для любой работы.

### Чего ни в коем случае не стоит делать

Во время поиска работы есть несколько распространенных ошибок, которых лучше избегать, чтобы сохранить хорошую репутацию и показать себя с профессиональной стороны. Избегая этих ошибок, вы повысите свои шансы получить работу, соответствующую вашим навыкам и карьерным планам.

Вот несколько вещей, которые лучше никогда не делать.

- **Не лгите в резюме и на собеседовании.** Не стоит преувеличивать свой опыт, образование или навыки — это легко проверить, и последствия будут неприятными. Говорите правду о том, что вы действительно умеете.
- **Не подавайте одновременно заявки на несколько разных вакансий в одной компании.** Это может выглядеть так, будто вы не определились, чем именно хотите заниматься. Выберите одну вакансию, наиболее подходящую вашим навыкам и опыту.
- **Не отправляйте одинаковое резюме всем подряд.** Всегда адаптируйте свое резюме и сопроводительное письмо под каждую конкретную вакансию. Покажите, что вы заинтересованы именно в этой должности.
- **Не игнорируйте инструкции, указанные в описании вакансии.** Если в объявлении просят отправить документы в конкретном формате или указать определенную информацию — обязательно сделайте именно так. Иначе ваше резюме могут сразу отклонить.
- **Не говорите плохо о своих бывших работодателях и коллегах.** Это выглядит непрофессионально и может создать впечатление, что вы плохо справляетесь с конфликтами и не умеете работать в команде. Лучше расскажите, какой полезный опыт вы получили на предыдущих местах работы.
- **Не используйте неформальный или временный адрес электронной почты.** Ваш адрес должен выглядеть профессионально, лучше всего — содержать ваше имя. Смешные или временные адреса могут испортить первое впечатление.
- **Не отправляйте резюме, не проверив его.** Ошибки, опечатки или плохо отформатированный текст покажут вас небрежным человеком. Всегда вни-

материльно перечитайте документы перед отправкой или попросите кого-то проверить их за вас.

- **Не приходите на собеседование неподготовленным.** Если вы заранее не изучите компанию и не подумаете о возможных вопросах, собеседование может пройти неудачно. Подготовьте ответы на частые вопросы и обязательно продумайте, что вы хотите спросить у рекрутера.
- **Никогда не опаздывайте на собеседование.** Пунктуальность крайне важна: опоздание создает плохое первое впечатление и показывает неумение планировать свое время. Лучше приходите на несколько минут раньше.
- **Не перезванивайте работодателю после собеседования без повода.** Гораздо лучше отправить короткое благодарственное письмо. Если решили отказаться от предложения, сообщите об этом вежливо и своевременно.
- **Не сжигайте мосты.** Даже если вас не приняли на работу, сохраняйте вежливость и хорошие отношения с компанией. Вы не знаете, когда в ней появится другая подходящая вакансия, и хорошие отношения могут пригодиться.

## Примеры вопросов, задаваемых на собеседованиях

Теперь рассмотрим 60 распространенных вопросов, которые часто задают на собеседованиях на замещение вакансии .NET-разработчика.

В первых трех примерах будут представлены сам вопрос, удачный ответ, а также неудачный ответ с объяснением. По моему опыту, изучая неудачные ответы, вы можете узнать не меньше, чем читая только удачные! Остальные 57 вопросов приведены без ответов.



Ответы на все 60 вопросов опубликованы на сайте GitHub по адресу [github.com/markjprice/tools-skills-net8/blob/main/docs/interview-qa/readme.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/interview-qa/readme.md).

Я советую повесить на стену доску и попросить друга зачитывать вам вопросы вслух. Отвечайте на них устно, записывайте себя на видео и просматривайте записи позже. Так вы привыкнете писать код на доске в условиях легкого стресса — как на реальном собеседовании.

Когда вы начнете разбирать эти вопросы, вы быстро заметите общие проблемы, особенно в неудачных ответах.

- **Слишком упрощенные объяснения.** Иногда плохой ответ сводит все к примитивному описанию технологии, концепции или процесса. Это может означать, что кандидат запомнил пару ключевых фраз, но не вник в суть. Даже если вы использовали технологию только в учебных целях, лучше честно об этом

сказать. И уж точно стоит хотя бы один раз написать код и посмотреть, как он работает.

- **Игнорирование лучших практик.** Плохие ответы часто упускают важные подходы и стандарты, которые считаются нормой в профессиональной среде.
- **Непонимание сути вопроса.** Бывает, что человек отвечает совсем не в ту сторону или не улавливает, о чем его спрашивают. В такой ситуации лучше прямо сказать, что вы с этим пока не сталкивались, чем притворяться. Работодатели обычно гораздо больше ценят тех, кто умеет задавать уточняющие вопросы и не боится признаться, если что-то непонятно.

Полный список тем, которые могут обсуждаться на собеседовании с .NET-разработчиком, поможет вам хорошо подготовиться и быть уверенным, что вы охватили все ключевые аспекты работы с платформой .NET. Ниже приведен подробный перечень таких тем — каждая из них может стать основой для вопроса на техническом собеседовании.

## 1. Инструменты .NET CLI

*«Можете объяснить, какую роль играют инструменты .NET CLI в процессе разработки? Приведите примеры того, как вы используете их для управления .NET-проектом на разных этапах его жизненного цикла».*

### Удачный ответ

Инструменты .NET CLI играют важную роль на всех этапах жизненного цикла .NET-приложения. Это кроссплатформенный набор команд, с помощью которого можно создавать, собирать, запускать и публиковать приложения прямо из терминала. Вот как они используются на разных этапах работы с проектом.

- **Создание проектов.** CLI поддерживает шаблоны для разных типов .NET-приложений. Например, чтобы создать веб-приложение ASP.NET Core, используется команда:

```
dotnet new webapp -n MyWebApp
```

- **Сборка.** Чтобы скомпилировать проект и проверить наличие ошибок без запуска приложения:

```
dotnet build
```

- **Запуск.** Для локального запуска во время разработки:

```
dotnet run
```

- **Добавление зависимостей.** CLI позволяет напрямую подключать пакеты NuGet. Так, чтобы добавить Entity Framework Core, выполните команду:

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

- **Миграции и обновление базы данных.** В проектах с Entity Framework можно использовать команды для управления миграциями схемы базы данных:

```
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

- **Тестирование.** С помощью команды `dotnet test` можно запускать модульные тесты прямо из терминала. Она определяет тестовые проекты, собирает их и выполняет все тесты:

```
dotnet test
```

- **Публикация.** Для подготовки приложения к развертыванию используется команда `dotnet publish`. Она собирает проект, подключает все зависимости и сохраняет готовую сборку в каталоге для публикации:

```
dotnet publish -c Release -o ./publish
```

CLI-инструменты повышают продуктивность, позволяя разработчику выполнять все в терминале — это упрощает работу как в редакторах кода, так и в конвейерах CI/CD.

## Пример неудачного ответа

Для управления .NET-проектом вполне хватает Visual Studio. Инструменты .NET CLI нужны только тем, кто не использует IDE.

Почему это плохой ответ?

- **Недооценка CLI.** Такой ответ сильно приижает роль CLI-инструментов, представляя их как нечто второстепенное по сравнению с Visual Studio. На самом деле CLI часто дает больше гибкости и скорости, особенно в ситуациях, когда IDE недоступна, — например, при сборке на CI-сервере или в легких редакторах вроде VS Code.
- **Непонимание экосистемы.** Упускается важнейшее преимущество CLI — возможность автоматизации. CLI удобно использовать в сценариях DevOps: команды можно объединять в сценарии и выполнять без участия пользователя. Это упрощает сборку, тестирование и развертывание.

- **Игнорирование кросс-платформенности.** В ответе не учитывается, что CLI работает одинаково на Windows, Linux и macOS. В то время как полная версия Visual Studio доступна только для Windows, CLI-инструменты дают разработчику свободу выбора среды.

Такие ответы часто связаны с ограниченным опытом — когда кандидат работал только в одной IDE и не сталкивался с другими сценариями, где CLI оказывается не просто удобным, а незаменимым.

## 2. Основы Git

*«Расскажите о базовых концепциях и командах Git, которые должен знать каждый .NET-разработчик. Как эти команды обычно применяются в повседневной работе над проектом?»*

### Удачный ответ

Git — это распределенная система контроля версий, которая помогает управлять изменениями в проекте и работать над одним кодом сразу нескольким разработчикам. Она особенно важна в командной разработке и поддержке стабильного процесса интеграции изменений.

Рассмотрим основные понятия и команды, которые должен знать каждый .NET-разработчик.

- **Репозиторий.** Git-репозиторий содержит файлы проекта и полную историю всех изменений. Репозиторий может быть локальным — на вашем компьютере — и удаленным, для совместной работы в команде.
- **Базовые команды:**
  - `git init` — создает репозиторий;
  - `git clone [url]` — копирует удаленный репозиторий;
  - `git add [файл]` — добавляет изменения в индекс для будущего коммита;
  - `git commit -m "[сообщение]"` — сохраняет изменения с описанием;
  - `git push [удаленный_репозиторий] [ветка]` — отправляет изменения на сервер;
  - `git pull [удаленный_репозиторий] [ветка]` — получает и сливает изменения с удаленной ветки.
- **Ветки и слияние:**
  - `git branch [имя]` — создает ветку;
  - `git checkout [имя]` — переключается на указанную ветку;
  - `git merge [ветка]` — объединяет изменения из другой ветки.

- **Рабочий процесс.** В обычном процессе разработки .NET-разработчики клонируют репозиторий, создают отдельную ветку под каждую задачу — будь то новая функциональность или исправление ошибки, — вносят изменения, фиксируют их локально и отправляют ветку в удаленный репозиторий. После реевью изменения сливаются в основную ветку. Команда `git pull` помогает держать локальную копию в актуальном состоянии и избегать конфликтов при слиянии.

Пример: создание новой ветки и отправка ее в удаленный репозиторий:

```
git checkout -b feature/new-feature
# Здесь идет разработка — редактируются файлы, вносятся изменения
git add .
git commit -m "Add new feature"
git push origin feature/new-feature
```

Регулярное использование этих команд помогает команде отслеживать изменения, при необходимости откатываться к предыдущим состояниям и эффективно работать вместе, не мешая друг другу.

## Пример неудачного ответа

Просто используйте `git commit` и `git push` — этого достаточно. Неважно, в какой ветке, главное, чтобы все в итоге попало в `main`.

Почему это плохой ответ?

- **Игнорирование стратегии ветвления.** Такой подход показывает, что кандидат не понимает, зачем вообще нужны ветки. Между тем ветвление — основа для параллельной работы над задачами, исправлениями и релизами. Если все сразу отправлять в `main`, это быстро приведет к конфликтам и нестабильному коду.
- **Слишком упрощенный взгляд на Git.** Использование только двух команд (`commit` и `push`) — это минимум, который не отражает всей мощи Git. В реальной работе важны и `git pull`, и `git branch`, и `git checkout`, особенно когда проект развивается в команде.
- **Нарушение культуры командной разработки.** Такой ответ игнорирует практики код-ревью и аккуратного слияния веток. А без них сложно поддерживать качество кода и отслеживать, кто, когда и зачем внес изменения. Это особенно критично в крупных проектах.

Обычно такой ответ говорит о том, что человек работал в одиночку или не сталкивался с полноценной системой контроля версий. Но в командной разработке такой подход быстро приведет к хаосу.

### 3. Entity Framework Core

«Расскажите, как вы используете Entity Framework Core для работы с базой данных в .NET-приложении. Приведите пример настройки EF Core с SQL Server и опишите, как выполняются базовые CRUD-операции. Зачем вообще использовать EF Core в проекте?»

#### Удачный ответ

Entity Framework Core — это современный ORM-фреймворк, который позволяет работать с базой данных через объекты .NET, избавляя от необходимости писать большую часть SQL-кода вручную. EF Core поддерживает разные базы данных, включая SQL Server, PostgreSQL и SQLite.

Чтобы подключить EF Core в .NET-приложении, сначала нужно настроить контекст базы данных и зарегистрировать его в системе внедрения зависимостей. Для работы с SQL Server создайте класс, унаследованный от `DbContext`, и определите в нем свойства `DbSet<T>` — по одному на каждую таблицу:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    public DbSet<Product> Products { get; set; }
}

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

Затем зарегистрируйте контекст и настройте подключение к SQL Server в файле `Program.cs`:

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(builder.Configuration
        .GetConnectionString("DefaultConnection")));

```

После этого можно выполнять базовые операции с данными:

```
using var scope = app.Services.CreateScope();
var context = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
```

```
// Добавление нового объекта
Product newProduct = new() { Name = "New Product", Price = 9.99M };
context.Products.Add(newProduct);
context.SaveChanges();

// Получение всех объектов
List<Product> products = context.Products.ToList();
```

EF Core предоставляет .NET-разработчику ряд серьезных преимуществ.

- **Скорость разработки.** Позволяет быстро создавать и обновлять структуру базы данных с помощью миграций.
- **Удобство поддержки.** Изменения в модели отражаются в базе с помощью миграций без ручного вмешательства.
- **Чистота кода.** Взаимодействие с базой данных абстрагировано — код становится проще и понятнее.

EF Core тесно интегрируется с остальной инфраструктурой .NET — системой внедрения зависимостей, конфигурацией, логированием, — поэтому хорошо подходит для построения устойчивых и поддерживаемых приложений.

## Пример неудачного ответа

Просто установите Entity Framework Core, создайте таблицы как классы — и все заработает автоматически, без какой-либо настройки.

Почему это плохой ответ?

- **Слишком упрощенный подход.** Этот ответ создает впечатление, что EF Core работает «из коробки», без дополнительной конфигурации. На практике необходимо явно настраивать контекст, указывать строку подключения и выбирать нужного провайдера базы данных.
- **Неверное понимание работы EF Core.** Классы не превращаются в таблицы сами по себе. Чтобы схема базы данных соответствовала структуре моделей, нужно использовать миграции и правильно настроить контекст. Иначе возможны ошибки при выполнении операций.
- **Игнорирование архитектурных практик.** В ответе ничего не сказано о таких вещах, как регистрация контекста в контейнере зависимостей или управление временем жизни подключения к БД, — а это ключевые аспекты в любом масштабируемом .NET-приложении.

Обычно такие ответы дают те, кто использовал EF Core в ограниченном виде — без понимания его полной архитектуры и роли в реальных проектах. Это не критично, но важно показать, что вы понимаете, как технология работает за кадром.

## 4. Интерфейсы и абстрактные классы

«Когда в .NET стоит использовать интерфейс, а когда — абстрактный класс? Приведите примеры ситуаций, в которых один из этих вариантов будет предпочтительнее».



Для вопросов с 4-го по 60-й в книге приведены только их формулировки — без примеров ответов. Полный список вопросов вместе с ответами доступен в репозитории на сайте GitHub [github.com/markjprice/tools-skills-net8/blob/main/docs/interview-qa/readme.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/interview-qa/readme.md).

## 5. Свойства и индексаторы

«В чем разница между свойствами и индексаторами в C#? В каких ситуациях удобнее использовать одно, а в каких — другое?»

## 6. Обобщения

«Зачем в .NET используются обобщения? Приведите пример, где они реально улучшают читаемость, надежность или производительность кода».

## 7. Делегаты и события

«Чем отличаются делегаты от событий в C# и как они связаны с паттерном Наблюдатель? Когда их можно применять?»

## 8. Интегрированный язык запросов (LINQ)

«Что такое LINQ и как он используется в .NET? Какие бывают поставщики LINQ и в каких ситуациях LINQ помогает писать код проще и понятнее?»

## 9. Асинхронное программирование с помощью async и await

«Как работают ключевые слова async и await в C#? Как они улучшают работу приложений .NET? Почему асинхронность важна в .NET-приложениях и в каких случаях она особенно полезна?»

## 10. Управление памятью и сборка мусора

«Как работает сборка мусора в .NET и какую роль она играет в управлении памятью? Какие сложности могут возникнуть при работе с ней и как их можно решать?»

## 11. Различия между современным .NET и .NET Framework

«Чем отличается современный .NET от .NET Framework? В каких ситуациях лучше выбрать один вариант, а в каких – другой?»

## 12. Кросс-платформенные возможности

«Какие кросс-платформенные возможности есть у современного .NET и как они помогают в разработке? Приведите пример, когда поддержка разных платформ особенно важна».

## 13. .NET Standard

«Для чего нужен .NET Standard и насколько он актуален после выхода .NET 5 и новее? Как он влияет на переносимость кода между проектами?»

## 14. Внедрение зависимостей в .NET

«Как реализуется внедрение зависимостей (DI) в .NET-приложении и зачем оно нужно? Приведите пример настройки DI и объясните, как с его помощью управлять зависимостями между сервисами».

## 15. Промежуточное ПО в ASP.NET Core

«Что такое промежуточное ПО (middleware) в ASP.NET Core и как его использовать? Приведите пример кастомного middleware и опишите ситуацию, когда оно может пригодиться».

## 16. Паттерн Configuration и Options

«Как реализовать паттерн Options в .NET-приложении для работы с конфигурацией? Покажите пример, как получать настройки из конфигурационного файла».

## 17. Хостинг и сервер Kestrel

«Что такое Kestrel в ASP.NET Core и как он используется вместе с IIS? Зачем настраивать совместную работу этих серверов в рабочем окружении?»

## 18. Типы данных

«В чем разница между значимыми и ссылочными типами в .NET? Когда уместно использовать каждый из них?»

## 19. Глобализация и локализация

«Как реализовать поддержку нескольких языков и культур в ASP.NET Core MVC-приложении? Что нужно для настройки глобализации и локализации?»

## 20. Структуры управления

«Какие структуры управления потоком есть в C# для работы с условиями и циклами? Приведите примеры, когда лучше использовать каждую из них».

## 21. Обработка исключений

«Зачем нужна обработка исключений в .NET и как ее правильно реализовать? Какие типы исключений вам встречались и как вы с ними работали?»

## 22. Стратегии ветвления в Git

«Какую стратегию ветвления вы используете в проектах? Почему она удобна для управления разработкой и выпуском новых версий?»

## 23. Обзор кода и парное программирование

«Чем полезны обзоры кода и парное программирование в .NET-команде? Приведите пример, как вы применяли эти практики на проекте».

## 24. Методологии Agile и Scrum

«Как Agile и Scrum помогают в .NET-проектах? Расскажите, как вы применяли подход Scrum на практике».

## 25. Стандарты документации

«Почему важны стандарты документации в .NET-разработке? Использовали ли вы такие инструменты, как DocFX или Mermaid, и чем они помогли?»

## 26. Навыки решения задач

«Расскажите о сложной технической задаче в .NET-проекте: как вы к ней подошли, какие инструменты использовали и чем все закончилось?»

## 27. Инструменты управления проектами

«Какие инструменты вы использовали для управления .NET-проектами? Как они помогали следить за задачами и наладить работу в команде?»

## 28. Методы оценки

«Какими методами вы оцениваете сроки и объем задач в .NET-проектах? Как они помогают не выходить за рамки бюджета и дедлайнов?»

## 29. Командное сотрудничество

«Как вы поддерживаете эффективную командную работу на .NET-проектах? Какие подходы и инструменты используете для общения и синхронизации внутри команды?»

## 30. Лидерство и наставничество

«Расскажите, как вы подходите к лидерству и наставничеству в .NET-команде. Как вы помогаете развивать коллег и одновременно двигать проект вперед?»

## 31. Паттерн MVC

«Что такое паттерн MVC и как вы применяли его в своих .NET-проектах? Приведите конкретный пример, где он помог улучшить архитектуру приложения».

## 32. Синтаксис Razor

«Как устроен синтаксис Razor в ASP.NET Core? Покажите на примере, как с его помощью можно формировать HTML на основе модели в .NET 8».

## 33. Разработка веб-API

«Как вы реализуете веб-API с минимальными API в .NET 6 и .NET 8? Расскажите, какие шаги включает разработка и чем такой подход удобнее по сравнению с предыдущими версиями».

## 34. Лучшие практики использования RESTful-сервисов

«Назовите удачные практики проектирования и реализации REST API в .NET. Приведите примеры, как они влияют на удобство поддержки и работу с сервисом».

## 35. SignalR для веб-функциональности в режиме реального времени

«Как с помощью *SignalR* реализовать систему уведомлений в реальном времени в .NET 8? Расскажите, как устроены такие приложения и как обеспечить их масштабируемость».

## 36. Управление состоянием

«Какие стратегии управления состоянием применяются в .NET-приложениях? Приведите примеры, в каких ситуациях что лучше использовать и как вы реализуете такой подход в .NET 8».

## 37. Проверка подлинности и авторизация

«Как настроить проверку подлинности и авторизацию в ASP.NET Core с минимальными API? Приведите конкретные примеры, как вы ограничиваете доступ к данным».

## 38. Blazor WebAssembly

«Как создать клиентское приложение с помощью *Blazor WebAssembly* в .NET 8? Расскажите, как вы настраиваете проект, подключаете зависимости и обрабатываете запросы к API».

## 39. Преимущества микросервисов

«В чем преимущества архитектуры микросервисов при разработке приложений на .NET?»

## 40. Проблемы в архитектуре микросервисов

«С какими трудностями вы сталкивались при реализации микросервисной архитектуры в .NET-проектах? Как вы их решали?»

## 41. Контейнеры Docker и .NET

«Как вы используете *Docker* для разработки, тестирования и развертывания .NET-приложений? Приведите пример настройки и запуска приложения .NET 8 в контейнере».

## 42. Модели взаимодействия микросервисов

«Какие паттерны взаимодействия между микросервисами вы знаете? Как бы вы реализовали их в .NET 8 с минимальными API? Приведите примеры, включая асинхронные вызовы».

## 43. Отказоустойчивость и обработка временных сбоев

«Как бы вы реализовали отказоустойчивость и обработку временных сбоев? Приведите примеры техник и инструментов, которые помогли бы вашему приложению стабильно работать и восстанавливаться после сбоев».

## 44. Распределенная трассировка

«Как бы вы настроили распределенную трассировку для мониторинга и отладки микросервисов? Какие инструменты и подходы использовали бы для этого?»

## 45. Проверка состояния функциональности и мониторинг

«Как бы вы реализовали проверку функциональности и мониторинг в сервисе .NET 8? Какие инструменты и подходы вы бы использовали, чтобы надежно отслеживать состояние приложения?»

## 46. AutoMapper, метод расширения или неявный оператор

«В чем разница между AutoMapper, методами расширения и неявными операторами при сопоставлении объектов в .NET 8? В каких случаях вы бы выбрали каждый из подходов? Приведите примеры с минимальными API».

## 47. Основы ADO.NET

«Как бы вы использовали ADO.NET в .NET 8 с минимальными API? Расскажите, как бы вы управляли подключениями, выполняли запросы и обрабатывали транзакции».

## 48. Настройка производительности Entity Framework Core

«Как бы вы оптимизировали производительность EF Core? Приведите примеры техник и настроек, которые помогли бы ускорить работу с базой данных».

## 49. Фреймворки для модульного тестирования, такие как xUnit

«Как бы вы использовали xUnit для модульного тестирования в .NET? Расскажите, как бы вы настроили проект, писали тесты и обеспечивали качество кода».

## 50. Фреймворки для имитаций типа NSubstitute

«Как бы вы использовали NSubstitute для имитации зависимостей в модульных тестах .NET-приложения? Приведите примеры имитаций зависимостей и проверки их поведения».

## 51. Стратегии интеграционного тестирования

«Как бы вы подошли к интеграционному тестированию в .NET-приложениях? Расскажите, как бы вы планировали, реализовывали и автоматизировали такие тесты. Какие инструменты и фреймворки использовали?»

## 52. Тестирование производительности

«Как бы вы проводили тестирование производительности в .NET? Опишите шаги — от планирования и выбора инструментов до выполнения тестов и анализа результатов».

## 53. Проверка безопасности

«Каков был бы ваш подход к проверке безопасности в .NET-приложении? Какие шаги вы бы предприняли для поиска и устранения уязвимостей в системе безопасности?»

## 54. Автоматизированное тестирование пользовательского интерфейса

«Как бы вы организовали тестирование пользовательского интерфейса в .NET-приложении? Какие инструменты и подходы вы бы использовали для полноценного покрытия UI-тестами?»

## 55. Принципы SOLID

«Что такое принципы SOLID и как они применяются при разработке .NET-приложений?»

## 56. Паттерн Одиночка

«Что такое паттерн Одиночка и как бы вы реализовали его в .NET-приложении? В чем его преимущества и какие могут быть риски при использовании?»

## 57. Паттерн Фабричный метод

«Как работает паттерн Фабричный метод и как бы вы применили его в .NET? Приведите пример, где он оказался бы уместен».

## 58. Идентификация утечек памяти

«Как определить и устраниить утечки памяти в .NET-приложении?»

## 59. Методологии разработки

«С какими методологиями разработки вы работали и как бы вы выбрали подходящую для .NET-проекта?»

## 60. Нотация «О большое»

«Представьте, что у вас есть массив целых чисел — идентификаторов пользователей. Как бы вы реализовали в .NET функцию для поиска дубликатов идентификаторов и какова была бы ее сложность в терминах “О большого”?»

## Ошибки — это путь к росту

Когда я работал тренером и помогал студентам готовиться к экзаменам, некоторые сильно расстраивались, если не могли ответить на мои вопросы. Но такое отношение, на мой взгляд, мешает понять суть практики — будь то перед экзаменом или на собеседовании.

Представьте: я задал вам десять вопросов, и вы ответили на все правильно. Прогресс есть? Вряд ли.

А теперь другой сценарий: вы ошиблись почти во всех вопросах, но после этого разобрались, в чем были неточности, и поняли, как нужно было отвечать. Есть ли прогресс? Да — и очень хороший.

Каждый раз, когда что-то не выходит, вы получаете сигнал: вот область, где можно расти. Это не провал, а ориентир. Ошибки не отдаляют от цели — наоборот, они показывают путь к ней.

## Практика и исследование

Проверьте свои знания и освоение темы, ответив на несколько вопросов, выполнив небольшие практические задания и подробнее изучив ключевые темы главы.

### Упражнение 19.1. Дополнительные онлайн-материалы

План развития для .NET-разработчиков опубликован на сайте [www.educative.io/blog/net-developer-roadmap](http://www.educative.io/blog/net-developer-roadmap).

### Упражнение 19.2. Практические задания

Один из лучших способов подготовки к собеседованию — провести пробное собеседование с языковой моделью, например ChatGPT. Особенно эффективно использовать такие тренировки, если вы добавите в запрос ссылки на свое резюме и описание вакансии.

Вот пример запроса:

*«Пожалуйста, помоги мне подготовиться к собеседованию на должность. Вот объявление о вакансии: [URL\_вакансии]. А вот ссылка на мое резюме: [URL\_резюме]. Кратко опиши основные обязанности и требования. Подскажи, какие пункты моего резюме лучше всего соответствуют вакансии и как акцентировать внимание на них в разговоре. Составь список вопросов, которые мне, скорее всего, зададут, — как технических, так и поведенческих — и предложи стратегии ответов. Приведи примеры вопросов по методу STAR и советы по их структурированию. Подскажи, какие вопросы я мог бы задать рекрутеру, чтобы показать свою заинтересованность. И, если можно, кратко расскажи о компании: ее миссии, ценностях, недавних новостях и других важных деталях, которые помогут мне подготовиться. Спасибо!»*

### Упражнение 19.3. Проверьте свои знания

Некоторые из этих вопросов потребуют от вас чуть больше, чем просто вспомнить материал, — нужно будет самому немного покопаться и разобраться. Я специально не даю все ответы в книге, потому что важно уметь выходить за рамки и смотреть шире.

И вот что действительно стоит запомнить: учитесь спрашивать, искать помошь и разбираться сами. Это как с рыбой — полезнее понять, как ее ловить, чем каждый раз ждать, что кто-то принесет готовую.

1. В чем заключаются обязанности бизнес-аналитика и как .NET-разработчик может с ним взаимодействовать?
2. Что такое парное программирование и как оно реализуется на практике?

3. Может ли языковая модель вроде ChatGPT или Llama 3 заменить .NET-разработчика?
4. Какие вопросы обычно задают на собеседовании?
5. Почему при изучении важно видеть не только правильные, но и ошибочные ответы?



Приложение «Ответы к разделам “Проверьте свои знания”» доступно в конце книги.

## Упражнение 19.4. Дополнительная информация

Дополнительные материалы по темам из этой главы собраны на странице [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-19---your-career-teamwork-and-interviews](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md#chapter-19---your-career-teamwork-and-interviews).

## Резюме

В этой главе вы разобрались, как:

- эффективно работать в команде разработчиков;
- подготовиться к поиску работы и подать заявку на вакансию;
- уверенно отвечать на типичные вопросы на собеседовании.

В следующей главе речь пойдет о том, что можно сделать дальше, чтобы продолжить обучение и развитие в мире C# и .NET.

# 20 Эпилог

Мне хотелось, чтобы эта книга отличалась от большинства технических изданий — чтобы она была не только полезной, но и живой, увлекательной, с практическими упражнениями по каждой теме. Надеюсь, вам было интересно ее читать и с ней работать.

## Следующие шаги на пути к изучению .NET

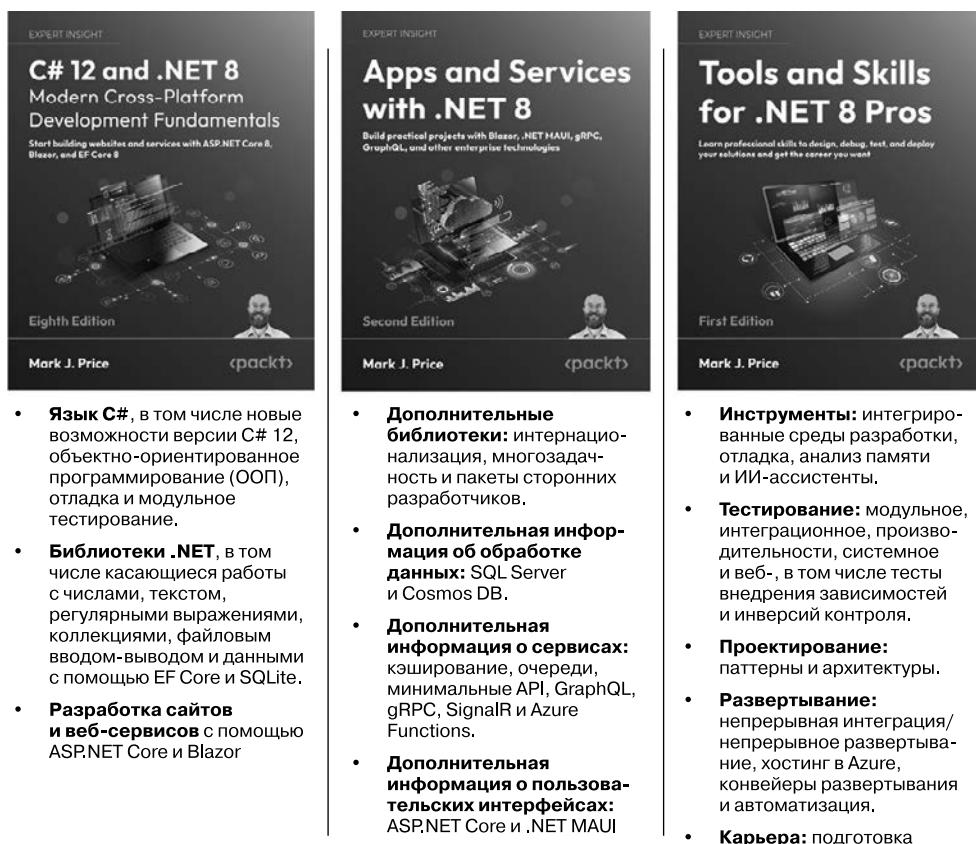
Эта книга — не исчерпывающее руководство. Надеюсь, что примечания, советы и ссылки на GitHub помогут вам углубить знания: [github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md](https://github.com/markjprice/tools-skills-net8/blob/main/docs/book-links.md).

## Сопутствующие книги для продолжения обучения

Я написал трилогию книг, которые охватывают все ключевые аспекты современной разработки на .NET 8. Две другие книги служат логичным продолжением и дополнением к этой.

1. **Первая книга** — *C# 12 and .NET 8 — Modern Cross-Platform Development Fundamentals* — знакомит с основами языка C#, библиотеками .NET и веб-разработкой с использованием ASP.NET Core и Blazor.
2. **Вторая книга** — «*.NET 8: приложения и сервисы*» — посвящена работе с более специализированными библиотеками, разработке сервисов и построению пользовательских интерфейсов для веба, настольных и мобильных приложений с помощью Blazor и .NET MAUI.
3. **Третья книга** — *та, которую вы держите в руках сейчас*, — посвящена важнейшим инструментам и практическим навыкам, необходимым для уверенной и профессиональной работы .NET-разработчиком.

Краткое описание трилогии .NET 8 и ее наиболее важных тем показано на рис. 20.1.



**Рис. 20.1.** Трилогия для изучения C# 12 и .NET 8

Чтобы посмотреть список всех книг, которые я выпустил в издательстве Packt, посетите страницу [subscription.packtpub.com/search?query=mark+j.+price](https://subscription.packtpub.com/search?query=mark+j.+price).

Чтобы ознакомиться с другими моими книгами, загляните на мою авторскую страницу на сайте Amazon: [www.amazon.com/Mark-J-Price/e/B071DW3QGN/](https://www.amazon.com/Mark-J-Price/e/B071DW3QGN/).

## Обновленное девятое издание книги C# и .NET — основы кросс-платформенной разработки

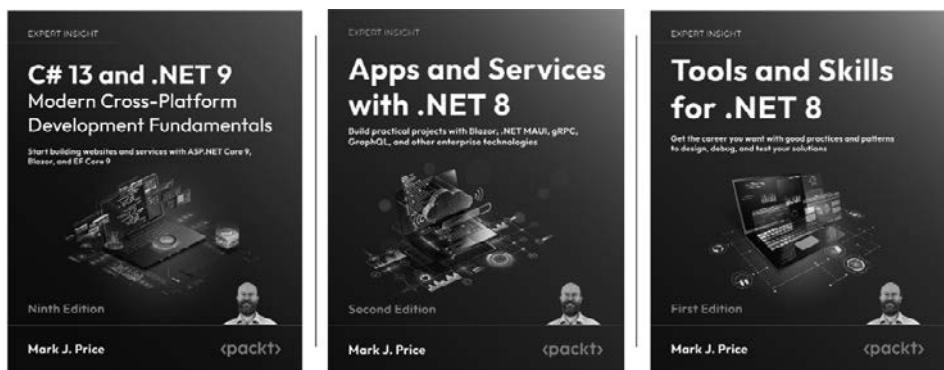
Первая книга из трилогии — *C# 13 and .NET 9 — Modern Cross-Platform Development Fundamentals* — уже доступна в девятом издании. Она была опубликована 12 ноября 2024 года одновременно с релизом .NET 9.

Если у вас уже есть восьмое издание, вам не обязательно покупать новое. Все исправления и улучшения доступны онлайн:

- список исправлений — на сайте [github.com/markjprice/cs12dotnet8/blob/main/docs/errata/errata.md](https://github.com/markjprice/cs12dotnet8/blob/main/docs/errata/errata.md);
- список улучшений и дополнений — на сайте [github.com/markjprice/cs12dotnet8/blob/main/docs/errata/improvements.md](https://github.com/markjprice/cs12dotnet8/blob/main/docs/errata/improvements.md).

В девятое издание вошли новые возможности C# 13, .NET 9, ASP.NET Core 9 и Blazor. В частности, теперь можно использовать коллекции в сигнатурах методов, появились новые методы LINQ, такие как `CountBy` и `Index`, улучшения в гибридном кэшировании, генерации OpenAPI, а также унифицированная модель хостинга.

Если вы еще не приобретали восьмое издание и хотите собрать полную трилогию .NET, рекомендуем выбрать именно девятое (рис. 20.2).



**Рис. 20.2.** Трилогия книг на тему .NET после выхода нового издания

## Будущее трилогии книг о .NET 10

Как вы, возможно, уже догадались, я также планирую выпустить новую версию этой книги — *Tools and Skills for .NET 10* — вместе с обновленными изданиями двух других книг из трилогии. Все три будут опубликованы вскоре после выхода .NET 10 в релиз (GA), который ожидается в ноябре 2025 года.

Работа над новым изданием уже началась. Если у вас есть предложения по темам, которые стоит дополнить, или вы заметили ошибки в тексте или коде, пожалуйста, оставьте отзыв в GitHub-репозитории этой книги по адресу [github.com/markjprice/tools-skills-net8](https://github.com/markjprice/tools-skills-net8).

Желаю вам вдохновения, уверенности и успехов во всех проектах на .NET!

# Приложение. Ответы к разделам «Проверьте свои знания»

В этом приложении содержатся ответы на вопросы раздела «Проверьте свои знания» в конце каждой главы.

## Глава 1

Ниже приведены варианты ответов на эти вопросы.

1. Как относятся к .NET-разработчикам, которые выбрали Rider, по сравнению с теми, кто работает только в Visual Studio?

**Ответ.** Восприятие .NET-разработчиков, выбравших JetBrains Rider, по сравнению с теми, кто работает в Visual Studio, может значительно отличаться в разных кругах технологического сообщества. Обе интегрированные среды разработки (IDE) имеют как сторонников, так и противников. Выбор между ними часто зависит от личных предпочтений, специфики проекта или более широких тенденций в команде или компании.

Visual Studio — проверенная временем IDE корпорации Microsoft, впервые выпущенная в конце 1990-х годов. Ее часто воспринимают как стандартную среду для разработки на .NET. Она глубоко интегрирована с экосистемой Windows и другими фреймворками Microsoft, предлагая широкие возможности «из коробки». Многие считают этот выбор надежным и безопасным: среда хорошо поддерживается, обеспечивает максимальную совместимость с новыми версиями .NET и тесно связана с инструментами Microsoft, такими как Azure, GitHub, DevOps и др.

Rider, напротив, разработан компанией JetBrains, известной по таким продуктам, как IntelliJ IDEA и ReSharper. Он ценится за кроссплатформенность: с его помощью можно разрабатывать .NET-приложения на Windows, macOS и Linux. В Rider встроен ReSharper — мощный инструмент анализа и рефакторинга кода, который многие считают незаменимым. Благодаря этому Rider воспринимается как инструмент для разработчиков, ориентированных на качество кода и высокую производительность.

Разработчики, выбирающие Rider, зачастую воспринимаются как более открытые к новым инструментам и подходам, выходящим за рамки экосистемы

компании Microsoft. Rider нередко выбирают за гибкость и производительность — среда известна как быстродействующая и менее требовательная к ресурсам по сравнению с Visual Studio. В средах, где важны кросс-платформенные возможности и нет жесткой привязки к Windows, пользователи Rider чаще ассоциируются с современными методами разработки и стремлением к технологической независимости. Выбор программы Rider может говорить о стремлении разработчика не ограничиваться стандартными инструментами, а искать оптимальные решения, ориентируясь на задачи, а не на привычные платформы. Выбор среды разработки не обязательно напрямую отражает уровень квалификации. Освоить .NET на высоком уровне можно в любой из IDE — Rider, Visual Studio или другой. Часто выбор определяется личными предпочтениями, спецификой проекта или внутренними правилами в компании. При этом разработчики, уверенно работающие в нескольких средах, обычно воспринимаются как более универсальные и гибкие в техническом плане.

В некоторых корпоративных средах, особенно там, где активно используются продукты компании Microsoft, Visual Studio может фактически считаться стандартом. Напротив, стартапы и компании с более разнообразным технологическим стеком зачастую ценят гибкость, которую предлагает Rider. При найме или формировании команды открытость к разным инструментам может сыграть положительную роль, демонстрируя адаптивность разработчика и его готовность выбирать не просто привычный, а наиболее подходящий для задачи инструмент.

В конечном счете решающими остаются навыки, опыт и способность эффективно использовать выбранные инструменты в конкретном контексте.

2. Какую формулировку стоит использовать, чтобы ChatGPT дал наиболее точный и полезный ответ?

**Ответ.** При работе с ChatGPT, особенно если вы хотите получить развернутый и обоснованный ответ, полезно добавить к запросу фразу: «*Выстраивай мысли поэтапно*». Так вы побуждаете модель отвечать структурированно, шаг за шагом, а не выдавать итог сразу.

3. О чём важно помнить, задавая вопрос на форуме или в чате Discord?

**Ответ.** Прежде чем задавать вопрос на форуме или в чате Discord, потратьте немного времени на самостоятельное исследование. Кратко опишите, что вы уже узнали, и укажите, что именно вызвало затруднение. Будьте конкретны, лаконичны и доброжелательны. Уважительное и терпеливое общение увеличивает шансы получить полезный и быстрый ответ.

4. Вы установили .NET 9 SDK. Как настроить проект так, чтобы он продолжал использовать .NET 8 для надежной поддержки, но при этом позволял пользоваться новыми возможностями компилятора C# 13?

**Ответ.** Чтобы использовать возможности компилятора C# 13 при сохранении ориентации на .NET 8, настройте проект следующим образом: в файле проек-

та (.csproj) оставьте значение `net8.0` в элементе `TargetFramework` и добавьте элемент `LangVersion` со значением `13`. Например:

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

<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <LangVersion>13</LangVersion> <!--Требуется стабильная версия SDK .NET 9--&gt;
    &lt;ImplicitUsings&gt;enable&lt;/ImplicitUsings&gt;
    &lt;Nullable&gt;enable&lt;/Nullable&gt;
&lt;/PropertyGroup&gt;

&lt;/Project&gt;</pre>
```

## 5. Зачем нужен GitHub-репозиторий, связанный с этой книгой?

**Ответ.** Репозиторий GitHub, связанный с этой книгой, используется для хранения кода решений, описанных в главах, а также дополнительных материалов, не вошедших в основной текст. Там можно найти исправления ошибок, полезные ссылки, небольшие улучшения, а также оставить сообщение автору, если у вас возникли вопросы по книге.

# Глава 2

Ниже приведены варианты ответов на эти вопросы.

## 1. Опишите, как воспользоваться функцией рефакторинга Change Signature (Изменить сигнатуру) в программе Visual Studio. Укажите, как получить к ней доступ и какие параметры можно изменить в процессе.

**Ответ.** Чтобы воспользоваться функцией рефакторинга Change Signature (Изменить сигнатуру) в Visual Studio, выполните следующие шаги.

- **Найдите нужный метод.** Перейдите к методу, сигнатуру которого хотите изменить. Это можно сделать, пролистывая код вручную или воспользовавшись функцией Go To (Перейти), — например, с помощью сочетания клавиш Ctrl+T, чтобы найти метод по имени.
- **Откройте инструмент изменения сигнатуры.** Поместите текстовый курсор в область имени метода или его объявления. Затем щелкните правой кнопкой мыши и выберите в контекстном меню команду Refactor ▶ Change Signature (Рефакторинг ▶ Изменить сигнатуру). Также можно воспользоваться сочетанием клавиш Ctrl+R, Ctrl+V, чтобы открыть окно изменения сигнатуры напрямую.
- **Измените сигнатуру.** В открывшемся диалоговом окне Change Signature (Изменение сигнатуры) доступны следующие параметры.
  - ◆ **Добавление параметров.** Нажмите кнопку Add (Добавить), чтобы добавить новый параметр. Укажите его тип, имя и значение по умолчанию (если нужно).

- ◆ **Удаление параметров.** Выделите ненужный параметр и нажмите кнопку Remove (Удалить).
  - ◆ **Изменение порядка параметров.** Выберите параметр и используйте кнопки Up (Вверх) и Down (Вниз) для его перемещения.
  - ◆ **Переименование параметров.** Чтобы изменить имя параметра, просто отредактируйте его в списке.
  - ◆ **Изменение типа параметров.** Тип любого параметра также можно отредактировать непосредственно в списке.
- **Предварительный просмотр изменений.** Visual Studio отобразит предварительный просмотр всех изменений, которые будут внесены в код. Так вы убедитесь, что они корректны, и сможете оценить, как они повлияют на другие участки, где используется метод.
  - **Примените изменения.** Нажмите кнопку OK, чтобы подтвердить свои действия. Программа Visual Studio автоматически обновит все ссылки на метод в проекте, отражая новую сигнатуру.
2. Объясните, как используется файл `.editorconfig` для стандартизации стиля форматирования кода. Что представляет собой этот файл? Приведите пример настройки отступов из четырех пробелов.
- Ответ.** Файл `.editorconfig` предназначен для задания и поддержания единых правил оформления кода в проекте. Он работает во всех популярных редакторах и IDE, включая Visual Studio, и позволяет переопределять глобальные настройки форматирования на уровне конкретного проекта.
- Чтобы использовать `.editorconfig`, выполните следующие шаги.
- 1) Создайте текстовый файл с именем `.editorconfig` в корневой папке проекта.
  - 2) Чтобы добавить отступы в виде пробелов размером 4 символа, добавьте в файл `.editorconfig` следующие строки, настраивающие свойства `indent_style` и `indent_size`:
- ```
root = true
[*]
indent_style = space
indent_size = 4
```
- ◆ `root = true` — указывает, что это главный файл `.editorconfig` для проекта и настройки из вышестоящих папок не применяются;
  - ◆ `[*]` — применяет настройки ко всем файлам в проекте;
  - ◆ `indent_style = space` — отступы будут задаваться пробелами;
  - ◆ `indent_size = 4` — каждый уровень отступа равен четырем пробелам.
3. В чем разница между фрагментом расширения и фрагментом с добавлением кода вокруг? Как ими пользоваться? Приведите пример для каждого варианта.

**Ответ.** Фрагменты расширения (Expansion) — это шаблоны, которые вставляются в текущую позицию текстового курсора. Их используют для быстрого добавления часто повторяющихся конструкций, таких как циклы, условные операторы, определения классов или методов. Чтобы использовать такой фрагмент, введите короткий код (например, `for`) и нажмите клавишу `Tab` — фрагмент развернется в готовый шаблон. Например, в C# после ввода `for` и нажатия клавиши `Tab` вы вставите заготовку цикла `for`, которую можно дополнить нужными условиями и операторами.

Фрагменты с добавлением кода вокруг (Surrounds With) используют, чтобы обернуть уже существующий код в определенную конструкцию — например, оператор `if`, блок `try-catch` или HTML-теги. Такие фрагменты особенно полезны, когда нужно обернуть выделенный фрагмент кода в конструкцию при веб-разработке. Чтобы применить такой фрагмент, выделите нужный участок кода, щелкните правой кнопкой мыши и выберите в контекстном меню команду **Snippets ▶ Surround With** (Фрагменты ▶ Добавить вокруг) или нажмите сочетание клавиш `Ctrl+K, Ctrl+S`.

Например, если ваш фрагмент кода должен выполняться согласно условию, выделите его, запустите функцию **Surrounds With** (Добавить вокруг) и выберите фрагмент `if`, чтобы обернуть этот код в условный оператор.

4. Какой атрибут элемента `Code` обязателен при определении фрагмента кода? Какие значения этого атрибута встречаются чаще всего?

**Ответ.** Обязателен только атрибут `Language`. Среди распространенных значений — `CSharp`, `XAML`, `XML`, `JavaScript`, `SQL` и `HTML`.

5. Какой шаблон необходимо установить для создания пользовательских шаблонов проектов на платформе .NET SDK?

**Ответ.** Чтобы создать пользовательские шаблоны проектов для .NET SDK, нужно установить шаблон `Microsoft.TemplateEngine.Authoring.Templates`:

```
dotnet new install Microsoft.TemplateEngine.Authoring.Templates
```

## Глава 3

Ниже приведены варианты ответов на эти вопросы.

1. Какие пять ключевых характеристик должна обеспечивать система управления исходным кодом?

**Ответ.** Ключевые характеристики системы управления исходным кодом включают следующие.

- **Контроль версий.** Позволяет отслеживать и управлять изменениями в коде с течением времени, фиксируя каждый шаг разработки. Благодаря этому можно возвращаться к предыдущим версиям файлов, видеть, кто и когда вносил изменения, и восстанавливать нужное состояние проекта.

- **Ветвление и слияние.** Позволяет создавать независимые ветки для разработки отдельных функций или исправлений. Появляется возможность работать параллельно, не мешая основной ветке. Когда работа завершена, ее можно объединить с основной линией разработки.
  - **Управление конкурентным доступом.** В командной работе система должна корректно обрабатывать ситуации, когда несколько человек изменяют одни и те же файлы. Для этого применяются механизмы блокировки или интеллектуальные средства слияния, которые помогают разрешать конфликты.
  - **Отслеживание изменений и аудит.** Система должна вести журнал всех изменений: кто, когда и что сделал, включая сообщения коммитов. Так обеспечивается прозрачность и возможность проследить причины тех или иных изменений.
  - **Интеграция с инструментами разработки.** Современные SCM-системы тесно связаны с IDE, средствами сборки, тестирования и развертывания. Это помогает автоматизировать процессы и ускоряет рабочий цикл — например, можно запускать тесты и сборку сразу после коммита.
2. Почему инструмент Git может быть сложным для изучения, особенно для начинающих?
- Ответ.** Работа с Git через терминал может показаться сложной, несмотря на простоту базовых команд, таких как `commit` и `push`. Трудности начинаются, когда требуется восстановиться после ошибочного коммита или слияния, — без понимания внутренней графовой модели Git сделать это непросто. Именно поэтому многие разработчики испытывают затруднения, сталкиваясь с подобными ситуациями.
3. В каких четырех состояниях может находиться файл в каталоге репозитория Git?
- Ответ.** Файл в каталоге Git-репозитория может находиться в одном из четырех состояний: неотслеживаемый (Untracked), подготовленный (Staged), неизмененный (Unmodified) и измененный (Modified).
4. В каких случаях применяется команда `git diff`?
- Ответ.** Команда `git diff` позволяет просматривать различия между разными состояниями файлов в репозитории. Это незаменимый инструмент для анализа изменений, особенно перед фиксацией, при сравнении веток или отслеживании изменений в отдельных файлах. Вот основные случаи, когда она используется.
- **Просмотр изменений перед коммитом.** Прежде чем выполнять коммит, полезно понять, что именно было изменено. Команда `git diff` без параметров показывает различия между рабочим каталогом и индексом (областью подготовки). Так вы увидите, какие строки кода были изменены и что именно будет добавлено в область подготовки.
  - **Проверка подготовленных изменений перед коммитом.** Если вы уже добавили изменения в область подготовки и хотите убедиться, что именно туда

попало, используйте команду `git diff --staged` (или `--cached`). Она покажет разницу между подготовленными файлами и последним коммитом — удобно для финальной проверки перед фиксацией. Такая команда отображает различия между подготовленными файлами и последним коммитом. Полезно для финальной проверки, чтобы убедиться, что в коммит попадут именно те изменения, которые вы планировали.

- **Анализ различий между ветками.** Чтобы понять, чем отличается одна ветка от другой, можно выполнить команду `git diff`, указав имена веток. Например, команда `git diff main feature-branch` покажет, какие изменения были внесены в ветке `feature-branch` по сравнению с `main`. Особенno полезно перед слиянием веток или для предварительной оценки возможных конфликтов.
  - **Сравнение изменений между коммитами.** С помощью команды `git diff` можно просмотреть различия между двумя конкретными коммитами. Указав их хеши, вы получите полную картину изменений между этими точками. Например, `git diff 1a2b3c4d 5e6f7g8h` отобразит, как изменился код от одного коммита к другому — удобно для анализа, аудита или понимания истории развития.
  - **Просмотр изменений в отдельном файле.** Чтобы отследить, как изменился конкретный файл с течением времени, можно указать его имя и диапазон коммитов. Например, `git diff HEAD~1 HEAD myfile.txt` покажет, что изменилось в файле `myfile.txt` между предыдущим и текущим коммитами.
5. Насколько часто используется ветвление в Git?

**Ответ.** Среди распространенных сценариев использования ветвления в Git можно выделить следующие.

- **Разработка функций.** Одно из самых распространенных применений веток в Git — работа над новыми функциями. Обычно разработчики создают отдельную ветку от ветки `main`, чтобы внести нужные изменения, не затрагивая основную кодовую базу. Так они могут вести независимую разработку до тех пор, пока новая возможность не будет готова к проверке и слиянию.
- **Исправление ошибок.** Ветки часто создаются для устранения конкретных дефектов. Это помогает сфокусироваться на решении проблемы, не мешая текущей работе. Когда исправление готово и протестировано, изменения сливаются обратно в `main` — как правило, после реvью.
- **Эксперименты.** Ветвление удобно использовать и для исследовательской работы. Разработчики могут спокойно пробовать новые подходы, технологии или архитектурные решения, не рискуя стабильностью проекта. Если эксперимент удачен, его можно внедрить, а если нет — ветку легко удалить без последствий.
- **Управление релизами.** Когда приложение готово к выпуску, создается отдельная ветка релиза. В ней можно внести последние штрихи — исправить мелкие

ошибки, уточнить документацию или подготовить дистрибутив. При этом основная разработка продолжается в `main`, не мешая релизной подготовке.

- **Быстрое исправление критических ошибок.** Ветки hotfix создаются для устранения серьезных проблем в продакшене. Обычно их ответвляют прямо от нужного тега или релизной ветки. После исправления и тестирования изменения сливаются обратно и в релизную ветку, и в `main`, чтобы они не потерялись при дальнейшей разработке.
- **Сотрудничество.** Ветвление упрощает параллельную работу над проектом: члены команды могут одновременно развивать разные части функциональности в собственных ветках. Такой подход помогает изолировать изменения, избежать конфликтов и улучшить управление задачами в рамках одной кодовой базы.
- **Обзор кода.** Перед слиянием с основной веткой принято использовать отдельные ветки для ревью. Разработчик публикует свою ветку в удаленном репозитории, и коллеги могут проверить изменения, оставить комментарии и предложить улучшения. Это повышает качество кода и способствует командному обучению.

## Глава 4

Ниже приведены варианты ответов на эти вопросы.

1. Какие приемы и стратегии считаются наиболее полезными при отладке?

**Ответ.** Некоторые важные стратегии отладки включают следующие приемы.

- Воспроизведите проблему, чтобы точно понять, в чем она заключается.
  - Соберите как можно больше информации: трассировки стека, журналы, сообщения об ошибках и другие диагностические данные.
  - Используйте инструменты отладки, чтобы изолировать источник ошибки.
  - Пишите модульные тесты, чтобы проверить, как работают отдельные участки кода, и добавляйте утверждения (`assert`), чтобы зафиксировать свои предположения.
  - Ведите журнал своих действий: что вы уже пробовали и к каким результатам это привело.
  - Сохраняйте спокойствие и действуйте последовательно. Иногда полезно на время отвлечься — это помогает взглянуть на проблему свежим взглядом.
2. Чем отличаются панели **Locals** (Локальные переменные) и **Watch** (Отслеживаемые значения) в процессе отладки?
- Ответ.** При отладке в таких средах, как Visual Studio, панели **Locals** (Локальные переменные) и **Watch** (Отслеживаемые значения) различаются по своему назначению.

- Панель **Locals** (Локальные переменные) автоматически отображает все переменные, доступные в текущей области видимости, — например, в методе, где выполнение остановлено на точке останова. Она обновляется в режиме реального времени по мере прохождения кода и показывает только те переменные, которые действительно видимы в текущем контексте. Панель не требует ручной настройки — она просто показывает все локальное окружение. Панель **Locals** (Локальные переменные) особенно полезна, когда вы работаете внутри вложенной функции или метода и хотите быстро оценить состояние всех локальных данных. Это удобный способ понять, что происходит в точке останова, без необходимости вручную добавлять переменные для отслеживания.
  - Панель **Watch** (Отслеживаемые значения) используется для ручного мониторинга определенных переменных или выражений во время сеанса отладки. В отличие от панели **Locals** (Локальные переменные), она не заполняется автоматически — вы сами добавляете интересующие переменные или выражения. В панель **Watch** (Отслеживаемые значения) можно добавить любые допустимые в текущем контексте выражения: это могут быть как локальные, так и глобальные переменные, вызовы методов или вычисляемые выражения. После добавления значения отслеживаются на протяжении всего сеанса отладки (или до тех пор, пока вы их не удалите), и вы можете наблюдать за их изменениями при прохождении кода. Панель **Watch** (Отслеживаемые значения) особенно полезна, если вы хотите следить за переменными, не входящими в текущую область видимости, наблюдать за выражениями, выходящими за рамки одного метода или блока, отслеживать вычисления, например результат вызова метода, наблюдать за изменениями значений в цикле или в ходе сложной логики. Если, например, вы отлаживаете цикл и хотите видеть, как меняется переменная, которая не является локальной, или хотите контролировать результат вызова метода на каждой итерации — панель **Watch** (Отслеживаемые значения) будет идеальным выбором.
3. Как управлять содержимым панелей отладки, например **Locals** (Локальные переменные)?
- Ответ.** Вы можете управлять тем, как ваши типы отображаются на панелях отладки в программе Visual Studio, с помощью специальных атрибутов, таких как `DebuggerDisplay` и `DebuggerBrowsable`. Эти атрибуты позволяют настраивать, какие значения и в каком виде будут показаны во время отладки.
4. По умолчанию строковые значения на панелях отладки отображаются в кавычках. Как скрыть эти кавычки?
- Ответ.** Чтобы убрать кавычки вокруг строки при отображении значения в отладке, добавьте суффикс `,nq` после имени поля:
- ```
[DebuggerDisplay("{ProductId}: {ProductName,nq}")]
```

5. Какие инструменты в Visual Studio и решениях JetBrains позволяют анализировать производительность приложения и использование памяти?

**Ответ.** Для анализа производительности и использования памяти в программе Visual Studio доступны инструменты, такие как профилировщик производительности, инструмент анализа использования памяти и профилирование процессора в режиме реального времени. В экосистеме JetBrains для этих целей используются программы `dotPeek`, `dotTrace`, `dotMemory` и `dotCover`.

## Глава 5

Ниже приведены варианты ответов на эти вопросы.

1. Какие существуют подходы к оптимизации журналирования и наблюдаемости в .NET-приложениях?

**Ответ.** Чтобы повысить качество журналирования и наблюдаемости в .NET-приложениях, стоит обратить внимание на несколько ключевых практик:

- используйте структурированное журналирование;
- настраивайте централизованное журналирование;
- добавляйте идентификаторы корреляции для отслеживания запросов между сервисами;
- применяйте инструменты наблюдаемости — в том числе OpenTelemetry;
- настраивайте оповещения и мониторинг;
- делитесь опытом и обучайте команду.

2. Что важно знать .NET-разработчику об интерфейсе `ILogger` и принципах его использования?

**Ответ.** Интерфейс `ILogger` предоставляет уровень абстракции над разными системами журналирования.

Разработчики могут расширять систему журналирования, реализуя собственных поставщиков. Интерфейс поддерживает разные уровни серьезности сообщений, что помогает классифицировать журналы по важности и упростить их анализ. `ILogger` также поддерживает структурированное журналирование — можно использовать шаблоны сообщений с заполнителями и передавать значения отдельно. Еще одна важная возможность — области журналирования. Они позволяют сгруппировать логические операции в рамках определенного контекста. Это полезно, например, для связывания логов с конкретным запросом или транзакцией — так проще отследить, какие события относятся к одному и тому же рабочему процессу. Интерфейс `ILogger` легко интегрируется со встроенным DI-фреймворком в .NET. Благодаря этому можно внедрять `ILogger<T>` в любой класс, где `T` — тип самого класса, что помогает организовать логику журналирования на уровне компонентов.

3. Почему в методах `LogX` используют шаблоны сообщений, а не просто строки?

**Ответ.** Важно учитывать, что параметр `message` в методах `LogX` представляет собой шаблон сообщения, а не готовую строку. Если передавать каждый раз новую строку вместо шаблона, при каждом вызове будет происходить выделение памяти, что увеличивает ее расход и снижает производительность. Чтобы избежать лишних затрат, стоит заранее определить ограниченное число шаблонов и передавать переменные данные через параметр `args`.

4. В чем разница между журналами, метриками и оповещениями?

**Ответ.** *Журналы* — это подробные записи, которые создаются приложением для фиксации событий, операций или транзакций. Они представляют собой хронологический отчет о том, что происходило и когда. Как правило, журналы включают временные метки, сообщения о событиях или ошибках, а также контекстные данные о состоянии приложения в момент записи. Эти сведения помогают находить и анализировать проблемы уже после того, как они произошли. Кроме того, журналы позволяют отслеживать действия пользователей и изменения в системе. Такой подход важен не только для устранения неполадок, но и для соблюдения нормативных требований в сфере безопасности и аудита.

*Метрики* — это количественные показатели, с помощью которых отслеживаются различные аспекты производительности и работы приложения. Обычно это числовые данные, собираемые с заданной периодичностью и агрегируемые по интервалам времени. Среди типичных метрик — загрузка процессора, использование памяти, время отклика и пропускная способность. Такие данные играют ключевую роль в мониторинге состояния приложения и помогают следить за тем, насколько эффективно используются ресурсы. Анализ метрик в долгосрочной перспективе может подсказать, когда потребуется масштабирование или перераспределение ресурсов, а также выявить тенденции, которые повлияют на развитие системы.

*Оповещения* — это уведомления, которые срабатывают при достижении определенных условий или пороговых значений в отслеживаемых данных, таких как журналы или метрики. Обычно они настраиваются на основе правил. Например, может быть задано оповещение, которое активируется, если загрузка процессора превышает 90 % в течение нескольких минут. Оповещения помогают своевременно замечать потенциальные проблемы — еще до того, как они перерастают в критические. Благодаря этому команды могут быстро реагировать и устранять неполадки. Такой подход снижает необходимость в постоянном ручном контроле, передавая системе задачу сообщать о важных событиях.

Если обобщить: журналы помогают понять *почему* — они содержат подробности произошедшего и связанный с ним контекст; метрики отвечают на вопросы *что* и *сколько* — давая обзор производительности и выявляя тенденции, а оповещения дают сигнал *когда* — срабатывая в реальном времени при наступлении заданных условий.

5. Что такое OpenTelemetry и почему этот инструмент рекомендован Microsoft для проектов на платформе .NET?

**Ответ.** OpenTelemetry (OTel) — это фреймворк, ориентированный на задачи наблюдаемости. Он объединяет в себе API, SDK и вспомогательные инструменты, которые совместно обеспечивают сбор телеметрических данных, таких как метрики и журналы. Среди основных преимуществ использования OpenTelemetry в проектах .NET — единый механизм сбора, согласованные схемы и семантика телеметрических данных, а также API для подключения внешних систем анализа. Так как в .NET уже есть собственный API для журналирования, изучать API OpenTelemetry отдельно не требуется. Вместо этого OpenTelemetry можно интегрировать через интерфейс `ILogger`.

## Глава 6

Ниже приведены варианты ответов на эти вопросы.

1. Как правильно оформлять комментарии в исходном коде?

**Ответ.** Вот полезные советы по комментированию кода.

- **Следите за актуальностью комментариев.** Если комментарий устарел и больше не отражает то, что происходит в коде, он может только запутать. Лучше убрать его совсем, чем оставить неточную информацию.
- **Не комментируйте очевидное.** Комментарии нужны там, где код не говорит сам за себя. Если все и так понятно — лишние пояснения только загромождают.
- **Страйтесь писать понятный код.** Говорящие имена переменных и методов, логичная структура, разбиение сложной логики на простые функции — благодаря этому код становится доступным без дополнительных пояснений. Такой стиль часто называют самодокументирующемся.
- **Документируйте открытые API.** Если вы разрабатываете библиотеку или интерфейс, которым будут пользоваться другие, используйте специальные форматы комментариев для генерации документации. Большинство языков это поддерживают, и такие комментарии действительно помогают другим разобраться в вашем коде.

2. Как задокументировать параметр метода с помощью XML-комментария?

**Ответ.** Чтобы задокументировать параметр метода с помощью XML-комментариев, добавьте элемент `param` с атрибутом `name` и укажите описание для каждого параметра:

```
/// <param name="culture">Код культуры ISO, например fr-FR, en-GB или es-AR.</param>
/// <param name="useComputerCulture">Задайте значение true, чтобы использовать
культуру локального компьютера.</param>
public static void ConfigureConsole(string culture = "en-US",
    bool useComputerCulture = false)
```

3. Можно ли применить стили ко всплывающей подсказке с XML-комментарием?

**Ответ.** В XML-комментариях нет поддержки сложного форматирования, но вы можете использовать следующие элементы:

- `b` — для полужирного текста;
- `i` — для курсива;
- `u` — для подчеркивания;
- `c` — для выделения встроенного фрагмента кода;
- `code` — для оформления блоков кода.

Такие элементы помогают выделять фрагменты кода, имена переменных и другие важные детали на фоне основного текста в документации.

4. Что представляет собой инструмент DocFX и для чего он используется?

**Ответ.** Инструмент DocFX создает статический HTML-сайт на основе вашего исходного кода (для документирования публичных API) и файлов в формате Markdown (чтобы вы могли добавлять дополнительные разделы документации). Макет и стиль сайта можно настраивать с помощью шаблонов.

5. Как отобразить наследование на диаграмме классов, созданной с помощью инструмента Mermaid?

**Ответ.** В диаграммах классов Mermaid наследование обозначается с помощью символа `<|--`, как показано в следующем фрагменте разметки:

```
classDiagram
    class Stream {
        <>abstract<>
        ...
    }

    class MemoryStream {
        ...
    }

    class FileStream {
        ...
    }

    Stream <|-- MemoryStream
    Stream <|-- FileStream
```

## Глава 7

Ниже приведены варианты ответов на эти вопросы.

1. Из каких частей состоит сборка .NET и какие из них обязательны?

**Ответ.** Сборка состоит из трех обязательных частей и одной необязательной.

- Метаданные и манифест сборки — имя, версия, ссылки на другие сборки и т. д.

- Метаданные о типах — информация о типах, их членах и структуре.
  - Код на языке промежуточного представления (IL) — реализация методов, свойств, конструкторов.
  - Встраиваемые ресурсы (необязательно) — изображения, строки, JavaScript и другие элементы.
2. К каким элементам кода можно применять атрибут?

**Ответ.** Существует перечисление (`enum`), которое определяет, к каким элементам кода может быть применен атрибут:

```
namespace System
{
    [Flags]
    public enum AttributeTargets
    {
        Assembly = 1,
        Module = 2,
        Class = 4,
        Struct = 8,
        Enum = 16,
        Constructor = 32,
        Method = 64,
        Property = 128,
        Field = 256,
        Event = 512,
        Interface = 1024,
        Parameter = 2048,
        Delegate = 4096,
        ReturnValue = 8192,
        GenericParameter = 16384,
        All = 32767
    }
}
```

3. Какие части входят в номер версии и что они обозначают при семантическом версионировании?

**Ответ.** Номер версии в .NET включает три обязательных компонента и два необязательных:

- мажорная версия — коренные изменения, несовместимые с предыдущими;
- минорная версия — новые функции и улучшения, не нарушающие совместимость;
- патч-версия — исправления ошибок без изменения функциональности.

Дополнительно могут быть указаны:

- предварительный релиз (Prerelease) — например `beta`, `rc` и т. д.;
- номер сборки (Build number) — часто используется для идентификации ночных сборок или CI-пайплайнов.

4. Как получить объект сборки текущего консольного приложения?

**Ответ.** Для этого используйте метод `Assembly.GetEntryAssembly()`. Он возвращает сборку, с которой было запущено текущее приложение.

5. Как получить все атрибуты, примененные к сборке?

**Ответ.** Вызовите метод `GetCustomAttributes()` для объекта сборки, чтобы получить список всех примененных к ней атрибутов.

6. Как определить и использовать собственный атрибут?

**Ответ.** Создайте класс, унаследованный от `Attribute`, и пометьте его атрибутом `[AttributeUsage]`, чтобы указать, к каким элементам он может применяться. Подробности о допустимых целях см. в ответе на вопрос 2.

7. От какого класса нужно наследоваться, чтобы загружать сборки динамически?

**Ответ.** Наследоваться нужно от класса `AssemblyLoadContext`. Он используется для динамической загрузки и выгрузки сборок во время выполнения.

8. Что такое дерево выражений и где оно используется в .NET?

**Ответ.** Дерево выражений – это структура данных, представляющая код в виде набора узлов. Ее можно анализировать, модифицировать или компилировать в выполняемый делегат.

9. Что такое генератор исходного кода и для чего он нужен?

**Ответ.** Генератор исходного кода позволяет получить доступ к объекту компиляции и сгенерировать дополнительные исходные файлы. Эти файлы автоматически включаются в процесс компиляции.

10. Какой интерфейс должен реализовывать генератор исходного кода и какие методы в него входят?

**Ответ.** Генератор реализует интерфейс `ISourceGenerator`, который включает два метода:

- `Initialize` – выполняется один раз при инициализации;
- `Execute` – вызывается при каждой компиляции и отвечает за генерацию исходного кода.

## Глава 8

Ниже приведены варианты ответов на эти вопросы.

1. Какой алгоритм из стандартной библиотеки .NET рекомендуется использовать для симметричного шифрования?

**Ответ.** Для симметричного шифрования рекомендуется использовать алгоритм AES.

2. Какой алгоритм из стандартной библиотеки .NET предпочтительнее для асимметричного шифрования?

**Ответ.** Для асимметричного шифрования рекомендуется использовать алгоритм RSA.

3. Что такое радужная таблица и как она используется при взломе хешей?

**Ответ.** Радужные таблицы содержат списки заранее вычисленных хешей паролей. При утечке базы данных с хешами злоумышленник может быстро сопоставить их с хешами из радужной таблицы и таким образом определить оригинальные пароли.

4. Для алгоритмов симметричного шифрования предпочтительнее больший или меньший размер блока?

**Ответ.** Для алгоритмов шифрования предпочтителен меньший размер блока, поскольку это повышает стойкость к определенным видам атак.

5. Что такое криптографический хеш и для чего он нужен?

**Ответ.** Криптографический хеш представляет собой выходное значение фиксированного размера, полученное в результате обработки входных данных произвольной длины с помощью хеш-функции. Такие функции являются односторонними — восстановить исходные данные можно лишь с помощью полного перебора и сравнения значений.

6. Что такое криптографическая подпись и как она используется?

**Ответ.** Криптографическая подпись — это значение, добавляемое к цифровому документу с целью подтверждения его подлинности. Действительная подпись гарантирует, что документ создан доверенным отправителем и не был изменен.

7. В чем разница между симметричным и асимметричным шифрованием?

**Ответ.** При симметричном шифровании один и тот же общий ключ используется как для шифрования, так и для расшифровки. В асимметричном шифровании открытый ключ используется для шифрования, а закрытый — для расшифровки.

8. Что означает аббревиатура RSA?

**Ответ.** Аббревиатура RSA образована от фамилий трех ученых — Ривеста (Rivest), Шамира (Shamir) и Адлемана (Adleman), которые впервые публично описали этот алгоритм в 1978 году.

9. Зачем к паролям добавляют соль перед их сохранением?

**Ответ.** Соль к паролям добавляется для повышения их стойкости к атакам с использованием радужных таблиц.

10. Почему использование алгоритма SHA-1 считается небезопасным и не рекомендуется?

**Ответ.** Не используйте SHA-1 — он больше не считается безопасным. Все современные браузеры отказались от поддержки SSL-сертификатов, основанных на этом алгоритме.

## Глава 9

Ниже приведены варианты ответов на эти вопросы.

1. Что такое семантическое ядро?

**Ответ.** Семантическое ядро — это SDK с открытым исходным кодом компании Microsoft, предназначенный для упрощенного создания агентов, способных вызывать ваш существующий код и взаимодействовать с языковыми моделями. С его помощью можно работать с моделями от OpenAI, Azure OpenAI, Hugging Face и других платформ. Объединяя такие модели с проектами на C# и .NET, вы сможете создавать интеллектуальных агентов, которые не только отвечают на вопросы, но и автоматизируют различные процессы.

2. Зачем используется метод `CreateFunctionFromMethod`?

**Ответ.** Метод `CreateFunctionFromMethod` позволяет расширить возможности большой языковой модели (LLM) с помощью пользовательских функций, написанных на C#. Каждая функция описывается на естественном языке в формате JSON, чтобы модель понимала, что делает функция и какие параметры она принимает. Процесс вызова функции происходит по следующей схеме.

- Пользователь отправляет сообщение, сопровождаемое набором функций, сопоставленных с методами .NET.
- LLM выбирает, вызвать одну или несколько функций либо генерировать ответ без вызова функции.
- Приложение выполняет соответствующий метод и передает его результат обратно в LLM.
- LLM использует полученные данные как дополнительный контекст при формировании окончательного ответа.

3. Почему по умолчанию каждый запрос в LLM через Semantic Kernel обрабатывается отдельно и как добавить сессионную память для учета предыдущих сообщений?

**Ответ.** Чтобы добавить сессионную память в чат, создайте экземпляр класса `ChatHistory` и вызывайте его метод `AddUserMessage` при каждом пользовательском запросе. Затем передавайте этот экземпляр в метод `GetChatMessageContentAsync`, чтобы вся история переписки включалась в запрос к LLM. После получения ответа от модели также вызовите метод `AddAssistantMessage`,

чтобы сохранить ответ в истории. Таким образом, модель будет получать полный контекст предыдущего общения при каждом новом запросе.

4. Как включить потоковую передачу ответов от LLM, если по умолчанию необходимо ждать полный результат?

**Ответ.** Чтобы включить потоковую передачу ответов от модели, используйте асинхронный оператор `foreach`:

```
await foreach (StreamingChatMessageContent message  
    in completion.GetStreamingChatMessageContentsAsync(history))
```

5. Что такое Hugging Face?

**Ответ.** Hugging Face – это компания, ставшая одной из ведущих в сфере искусственного интеллекта, особенно в области обработки естественного языка (NLP). Она приобрела широкую известность благодаря своей активной роли в развитии решений с открытым исходным кодом, в частности библиотеки Transformers. Эта библиотека предоставляет доступ к современным моделям машинного обучения, которые можно применять для таких задач, как классификация текста, извлечение информации, ответы на вопросы, обобщение и перевод. Кроме библиотеки, компания Hugging Face поддерживает онлайн-платформу для размещения и распространения моделей, где опубликованы тысячи предварительно обученных моделей, предоставленных сообществом. Эти модели можно легко загрузить и интегрировать в проекты на Python и других языках, связанных с NLP.

## Глава 10

Ниже приведены варианты ответов на эти вопросы.

1. В чем заключается основная идея внедрения зависимостей (DI)?

**Ответ.** Основная идея внедрения зависимостей (DI) заключается в том, чтобы отделить создание зависимостей объекта от его поведения. Благодаря этому код становится более модульным, тестируемым и удобным в сопровождении. Вместо того чтобы объект сам создавал свои зависимости, они предоставляются во время выполнения, как правило, с помощью внешнего контейнера или фреймворка.

2. Какие существуют три основных способа внедрения зависимостей? Какой из них не поддерживается в .NET напрямую, если не использовать специальные подходы?

**Ответ.** Существует три основных способа получения зависимостей в .NET:

- через конструктор — зависимости передаются через параметры конструктора. Считается наилучшей практикой, особенно в контексте модульного тестирования;
- через метод — зависимости указываются в сигнатуре метода и могут передаваться через DI-контейнер;

- через свойства — зависимости присваиваются свойствам класса. В .NET этот способ не поддерживается по умолчанию, но может быть реализован с помощью сторонних контейнеров, например Autofac.
3. Какие типы времени жизни используются при регистрации сервисов в .NET и чем они отличаются?
- Ответ.** В .NET поддерживаются три типа времени жизни зависимостей:
- времененная (Transient): каждый раз при запросе создается новый экземпляр. Подходит для легких объектов без состояния;
  - область видимости (Scoped): создается один экземпляр на каждый клиентский запрос. Актуально для веб-приложений;
  - одиночка (Singleton): создается один экземпляр на все время работы приложения. Может быть создан автоматически или предоставлен вручную при регистрации.

4. Какие события можно обрабатывать при работе с сервисом узла?

**Ответ.** Интерфейс `IHostedService` определяет два метода, связанных с жизненным циклом сервиса:

- `StartAsync` — вызывается при запуске приложения;
- `StopAsync` — вызывается при завершении работы.

Кроме того, можно подписаться на события через `IHostApplicationLifetime`: `ApplicationStarted`, `ApplicationStopping` и `ApplicationStopped`.

5. В какой момент запускаются сервисы узла в жизненном цикле приложения?

**Ответ.** Сервисы, реализующие `IHostedService`, запускаются после вызова метода `RunAsync()` универсального узла (`Host`). С этого момента узел начинает управлять жизненным циклом размещенных сервисов.

## Глава 11

Ниже приведены варианты ответов на эти вопросы.

1. Каким требованиям должен соответствовать качественный модульный тест?

**Ответ.** Качественный модульный тест должен соответствовать следующим критериям.

- Проверять только одну единицу поведения. Например, отдельный метод, реализующий бизнес-логику.
- Выполняться как можно быстрее. Для ускорения тестирования бизнес-логики вместо производственной базы данных можно использовать хранилище в оперативной памяти. Качественный фреймворк для тестирования также позволяет установить тайм-аут, чтобы избежать излишне долгого выполнения тестов.
- Работать изолированно — как от других тестов, так и, при необходимости, от внешних зависимостей.

2. Что такое MUT и какую роль он играет в модульном тестировании?

**Ответ.** Тестируемый метод (Method Under Test, MUT) — это метод внутри тестируемой системы (System Under Test, SUT). Тестируемая система — это тип, который проверяется в тестах. Обычно для SUT создают отдельный тестовый класс, включающий несколько тестовых методов, чтобы сгруппировать все проверки, относящиеся к этой системе.

3. Что означает формула «красный — зеленый — рефакторинг» в подходе TDD?

**Ответ.** Три этапа TDD, известные как «красный — зеленый — рефакторинг», включают следующее.

- **Красный.** Напишите тест для следующей части функциональности, которую планируете добавить. Этот тест должен завершиться неудачей, поскольку сама функциональность еще не реализована. Благодаря этому вы убедитесь, что тесты действительно содержательны и проверяют именно требуемое поведение.
- **Зеленый.** Напишите минимально необходимый код, который позволит тесту пройти успешно. Такой подход помогает создавать простую и сосредоточенную реализацию без лишней сложности.
- **Рефакторинг.** Улучшите новый код, сделав его более чистым, удобочитаемым и соответствующим передовым практикам. При этом все тесты должны продолжать успешно выполняться, чтобы можно было быть уверенными: существующая функциональность осталась неизменной и не возникло новых ошибок.

4. Как выполняется настройка тестов в библиотеке xUnit?

**Ответ.** Тесты в xUnit настраиваются с помощью атрибутов .NET, что делает код тестов легкочитаемым и понятным. Для стандартных тестов используется атрибут **[Fact]**, а для параметризованных — атрибут **[Theory]** вместе с **[InlineData]**, **[ClassData]** или **[MemberData]**, что позволяет организовать тестирование на основе данных.

5. Какой параметр является общим для атрибутов **[Fact]** и **[Theory]** в xUnit и зачем его указывать?

**Ответ.** Атрибуты **[Fact]** и **[Theory]** позволяют задать тайм-аут выполнения теста в миллисекундах. Например:

```
[Fact(Timeout = 3000)] // Тест завершится через 3 секунды
```

Тайм-аут помогает избежать ситуаций, когда тесты выполняются слишком долго.

6. При использовании **[Theory]** и атрибута **[ClassData]** какие требования предъявляются к предоставляемому даннмые классу?

**Ответ.** Класс, который указывается через атрибут **[ClassData]**, должен реализовать интерфейс **IEnumerable<object[]>**. Это требует реализации как обоб-

щенного, так и необобщенного метода `GetEnumerator`, возвращающего последовательность массивов объектов. В качестве альтернативы можно использовать наследование от класса `TheoryData`, чтобы создать сильно типизированный источник данных.

7. Как правильно обрабатывать как успешные, так и провальные результаты модульного теста?

**Ответ.** Положительные и отрицательные результаты тестирования лучше разделять на отдельные методы с соответствующими наборами параметров. Если сценарий оказывается сложным и его суть не сразу понятна при просмотре параметров, стоит создать отдельный тестовый метод с атрибутом `[Fact]`, чтобы сделать логику теста более очевидной.

8. На какие настораживающие признаки стоит обратить внимание при написании или проверке модульных тестов?

**Ответ.** Вот признаки, на которые стоит обращать внимание при написании модульных тестов.

- Использование выражений `if` или `switch` считается антипаттерном, поскольку в модульных тестах не должно быть ветвлений.
- Если секция подготовки становится слишком громоздкой, разумно вынести часть кода подготовки в приватные методы, чтобы сделать тест чище и проще для понимания.
- Раздел действия, как правило, должен состоять из одного выражения. Если для проверки одной единицы поведения требуется несколько вызовов методов, это может указывать на недостатки в дизайне публичного API.
- Секция проверки обычно включает одно выражение. При необходимости можно проверить несколько выходов, но в таком случае стоит подумать о перепроектировании API, чтобы возвращать тип записи, объединяющий несколько значений. Так можно будет сформулировать одно аккуратное утверждение.

9. Каким образом в xUnit можно выводить сообщения во время выполнения теста?

**Ответ.** В xUnit нельзя использовать метод `Console.WriteLine`, поэтому для вывода информации во время выполнения теста применяется интерфейс `ITestOutputHelper`. Его следует внедрить через конструктор тестового класса.

10. Как с помощью NSubstitute задать возвращаемое значение у подставного метода?

**Ответ.** При работе с библиотекой NSubstitute настройка возвращаемого значения метода выполняется с помощью метода `Returns` для объекта-заменителя. Пример настройки:

```
ICalculator calc = Substitute.For<ICalculator>();  
calc.Add(2, 3).Returns(5);
```

## Глава 12

Ниже приведены варианты ответов на эти вопросы.

1. Как соотносятся между собой тестовые дублеры: имитации, шпионы, заглушки, фальшивки и пустышки?

**Ответ.** Тестовые дублеры — это общее название для любых поддельных зависимостей, которые используются в тестах вместо реальных компонентов. Они помогают упростить настройку тестового окружения, когда использование настоящих зависимостей было бы сложным или непредсказуемым. Существует несколько типов тестовых дублеров. Наиболее распространенные из них — имитации (mock-объекты) и заглушки (stub).

- *Имитации* предназначены для подмены исходящих взаимодействий. Например, в тесте можно использовать имитацию зависимости, которая симулирует отправку электронного письма при регистрации пользователя. При этом во внешней системе состояние может изменяться. Обычно имитации создаются с помощью специальных фреймворков. Если имитация создается вручную, ее иногда называют *шпионом*.
  - *Заглушки* предназначены для подмены входящих взаимодействий. Например, тест может использовать заглушку зависимости, которая возвращает информацию о товаре из базы данных. При этом состояние во внешней системе не изменяется. Если зависимость еще не реализована — например, в процессе разработки по методологии TDD, — такую заглушку называют *фальшивкой*. Если же заглушка представляет собой простое значение и не влияет на исход теста, ее называют *пустышкой*.
2. Какие внешние зависимости в интеграционных тестах стоит тестиировать напрямую, а какие — заменять имитациями?

**Ответ.** Внешние системы можно условно разделить на два типа: находящиеся под вашим контролем и вне вашего контроля. К внешним системам, находящимся под контролем, относятся, например, хранилища данных, доступ к которым имеет только ваш проект и которые не обновляются другими системами. Такие внешние системы рекомендуется подключать напрямую. Внешние системы, которые находятся вне вашего контроля, включают почтовые сервисы, публичные API, такие как прогноз погоды, а также государственные системы. В таких случаях предпочтительнее использовать имитации, чтобы обеспечить стабильность тестов и независимость от внешних факторов.

3. Что такое туннель разработки и в чем его польза для .NET-разработчика?

**Ответ.** Туннели разработки позволяют безопасно предоставлять локальную среду через интернет. Этот подход особенно полезен, когда требуется продемонстрировать разработку коллегам, клиентам или сторонним сервисам без необходимости развертывать приложение в публичной тестовой среде. Для разработчиков .NET, использующих Visual Studio, доступнастроенная инте-

трация с инструментами туннелей, что упрощает их настройку. Возможность быстро делиться результатами работы и оперативно получать обратную связь помогает ускорить цикл разработки и повысить качество конечного продукта.

#### 4. С чего начать проверку безопасности .NET-приложения?

**Ответ.** Профессиональный разработчик .NET при организации проверки безопасности должен начинать с изучения отчета OWASP Top 10, в котором перечислены наиболее важные риски безопасности веб-приложений. Особое внимание следует уделить практике безопасного программирования: правильной обработке пользовательского ввода, надежной проверке подлинности и авторизации, безопасному управлению сеансами и шифрованию конфиденциальных данных. Соблюдение этих принципов помогает значительно снизить вероятность появления уязвимостей.

#### 5. Что такое жизненный цикл разработки безопасного ПО Microsoft и как он применяется на практике?

**Ответ.** Жизненный цикл безопасной разработки Microsoft (Security Development Lifecycle, SDL) представляет собой процесс, направленный на создание более безопасного ПО и выполнение требований по обеспечению безопасности при снижении затрат на разработку. SDL предлагает четкое руководство, практические методы и инструменты для всех этапов жизненного цикла ПО, включая моделирование угроз. Следование этим практикам помогает встроить безопасность в проект на ранних этапах и поддерживать ее на всем протяжении разработки.

## Глава 13

Ниже приведены варианты ответов на эти вопросы.

#### 1. Что такое нотация «O большое»?

**Ответ.** Нотация «O большое» (Big O) – это математический способ описания производительности или сложности алгоритма. Она характеризует временные и пространственные затраты в зависимости от размера входных данных (обозначается как  $n$ ). Нотация «O большое» показывает верхнюю границу ресурсов, которые может потребовать алгоритм, и позволяет сравнивать эффективность разных алгоритмов без привязки к деталям реализации или особенностям оборудования. Как правило, «O большое» описывает поведение алгоритма в наихудшем случае, помогая понять, сколько времени или памяти потребуется при росте объема данных. Благодаря этому можно оценить масштабируемость алгоритмов и заранее представить, насколько они будут эффективны в реальных условиях. Понимание нотации «O большое» помогает разработчикам и специалистам в области вычислений выбирать алгоритмы, наиболее подходящие для решения задач, особенно когда приходится работать с большими объемами данных.

2. Какая типичная ошибка встречается в процессе оценки производительности и что предлагает BenchmarkDotNet, чтобы получить надежные результаты?

**Ответ.** К распространенным ошибкам при проведении оценки производительности относятся:

- отсутствие изоляции тестируемого кода от логики подготовки и завершения работы. Пакет BenchmarkDotNet предоставляет атрибуты [`GlobalSetup`] и [`GlobalCleanup`], позволяя отделить код настройки и очистки от кода, который проходит фактическое тестирование;
- игнорирование влияния компилятора just-in-time (JIT), который оптимизирует код во время выполнения. Это может исказить результаты тестирования производительности. Пакет BenchmarkDotNet автоматически учитывает прогрев JIT-компилятора, поэтому важно доверять инструменту и не пытаться управлять этим процессом вручную;
- сравнение результатов, полученных на разных компьютерах, в разных средах или конфигурациях. Такие различия могут привести к ошибочным выводам. BenchmarkDotNet позволяет экспортить и сравнивать результаты, но важно, чтобы тесты проводились в одинаковых условиях, а особенности оборудования и среды обязательно учитывались.

3. В чем разница между нагрузочным тестированием и стресс-тестированием и в каких случаях применяется каждое из них?

**Ответ.** Нагрузочное тестирование направлено на оценку производительности системы при ожидаемой нагрузке. Под нагрузкой понимаются условия реальной эксплуатации, такие как количество одновременно работающих пользователей или объем обрабатываемых транзакций. Во время нагрузочного тестирования система подвергается постепенно увеличивающемуся объему работы — например, росту количества пользовательских запросов или операций с базой данных, — чтобы смоделировать типичные условия эксплуатации. Цель состоит в том, чтобы определить, сохраняет ли система стабильность и надлежащую производительность при ожидаемой нагрузке и в какой момент могут возникнуть сбои или снижение эффективности. Типичным примером является подготовка коммерческого сайта к распродаже в «черную пятницу», когда ожидается пик активности пользователей, но все еще в рамках прогнозируемых значений.

Стресс-тестирование, в отличие от нагрузочного, проверяет отказоустойчивость системы в экстремальных условиях. Оно направлено на определение предела возможностей системы за счет создания нагрузки, значительно превышающей нормальные эксплуатационные уровни. В ходе стресс-тестирования систему подвергают чрезмерным нагрузкам, чтобы понять, когда и как произойдет сбой и насколько эффективно система сможет восстановиться. Проверяются такие аспекты, как утечки памяти, ошибки синхронизации и устойчивость к повреждению данных. Примером может служить имитация резкого всплеска пользовательского трафика или перегрузка ресурсов процессора и памяти.

Таким образом, нагрузочное тестирование помогает убедиться, что система стабильно работает при ожидаемой нагрузке, тогда как стресс-тестирование

выявляет пределы ее возможностей и проверяет поведение при экстремальных условиях. Нагрузочные тесты помогают оптимизировать производительность в повседневной работе, а стресс-тесты — убедиться в устойчивости и способности системы восстанавливаться после серьезных сбоев.

- Что представляет собой инструмент Bombardier и чем он полезен при тестировании производительности?

**Ответ.** Bombardier — это высокопроизводительный кроссплатформенный инструмент для оценки производительности HTTP, написанный на языке Go. Он предназначен для создания значительной нагрузки с минимальным воздействием на систему, что позволяет проверять производительность веб-серверов и сервисов. Bombardier способен генерировать большое количество запросов в секунду, благодаря чему подходит для стресс-тестирования и нагрузочного тестирования веб-приложений, помогая понять, как они работают в условиях интенсивного трафика.

- Из чего состоит сценарий нагрузки в NBomber?

**Ответ.** Сценарий фреймворка NBomber включает в себя следующие элементы.

- Этапы* — отдельные операции или действия, которые выполняются в ходе теста. Например, для веб-приложения таким шагом может быть выполнение HTTP-запроса к определенной конечной точке.
- Моделирование нагрузки* — определяет способ генерации нагрузки: указываются количество одновременных пользователей или запросов и продолжительность теста. NBomber поддерживает различные стратегии моделирования, включая постоянную нагрузку или постепенное увеличение и уменьшение нагрузки.
- Настройки сценария* — дополнительные параметры, такие как глобальные заголовки для HTTP-запросов или действия по настройке и очистке, которые выполняются до и после тестирования.

## Глава 14

Ниже приведены варианты ответов на эти вопросы.

- Какие браузеры используются при выполнении тестов в Playwright?

**Ответ.** Фреймворк Playwright использует сборки Chromium с открытым исходным кодом. Он может работать с фирменными браузерами, установленными на вашем компьютере. В частности, текущая версия Playwright поддерживает каналы *Stable* и *Beta* для таких браузеров. Версия Firefox для Playwright основана на самой свежей сборке Firefox Stable. А версия WebKit для Playwright использует последнюю доступную сборку WebKit Trunk, которая затем попадает в Apple Safari и другие браузеры на базе движка WebKit. Playwright не работает с фирменными версиями Firefox или Safari, поскольку для их поддержки требуются дополнительные патчи.

2. Какие ключевые интерфейсы отвечают за работу с важными объектами при написании тестов в Playwright?

**Ответ.** Основные интерфейсы, которые представляют важные объекты при написании тестов для Playwright: `IPlaywright`, `IBrowser`, `IBrowserContext`, `IResponse`, `IPage` и `ILocator`.

3. Какими способами в Playwright можно получить один или несколько элементов на веб-странице?

**Ответ.** Фреймворк Playwright предоставляет несколько полезных методов для поиска одного или нескольких элементов на веб-странице. Среди них — `GetByRole`, `GetByLabel`, `GetByPlaceholder`, `GetById`, `GetByText`, `GetByTitle` и `GetByAltText`.

4. Что произойдет, если совпадет несколько элементов, а вы вызовете метод, предназначенный для одного элемента DOM, например `ClickAsync`?

**Ответ.** Если локатор находит более одного элемента, а вызывается метод, предназначенный для одного элемента (например, `ClickAsync`), будет выброшено исключение. Например, если локатор сопоставляет все кнопки на странице и вызывает `ClickAsync`, тест завершится с ошибкой.

5. Для чего используется Playwright Inspector?

**Ответ.** Инструмент Playwright Inspector позволяет создавать полноценные тестовые сценарии за небольшую часть времени, которое потребовалось бы на их написание вручную. Он с высокой точностью фиксирует взаимодействия пользователя с веб-приложением, снижая риск ошибок, которые могут возникнуть при ручной разработке тестов.

## Глава 15

Ниже приведены варианты ответов на эти вопросы.

1. Почему контейнеры считаются более «легкими» по сравнению с виртуальными машинами?

**Ответ.** Контейнеры работают на ядре операционной системы одного компьютера и совместно используют это ядро с другими контейнерами. Они остаются легкими, потому что не требуют гипервизора, который нужен для управления *виртуальными машинами (ВМ)*. Контейнеры запускаются напрямую в ядре хост-машины. Благодаря этому они работают быстрее, эффективнее и потребляют меньше ресурсов по сравнению с виртуальными машинами, для которых требуется полноценная отдельная операционная система.

2. Как связаны Docker и Kubernetes?

**Ответ.** Docker и Kubernetes — это тесно связанные технологии, которые играют ключевую роль в контейнеризации и управлении приложениями.

- *Docker* – это платформа, позволяющая разработчикам упаковывать приложения в контейнеры. Контейнер объединяет исходный код приложения, библиотеки операционной системы и все зависимости, необходимые для запуска в любой среде. Docker упрощает создание, развертывание и выполнение приложений за счет стандартизации и изоляции.
- *Kubernetes* – это платформа с открытым исходным кодом, предназначенная для автоматизации развертывания, масштабирования и эксплуатации контейнерных приложений. Она решает задачи, которые возникают при управлении множеством контейнеров на разных серверах: автоматизация масштабирования, балансировка нагрузки, отказоустойчивость и многое другое.

Docker и Kubernetes отлично дополняют друг друга. Docker предлагает простой и удобный способ контейнеризации приложений, а Kubernetes обеспечивает управление их запуском в производственной среде. Kubernetes часто использует Docker как среду выполнения контейнеров. В типичном рабочем процессе разработчик пишет приложение, упаковывает его в контейнер Docker и загружает в реестр, например Docker Hub. Затем Kubernetes извлекает этот контейнер и управляет его развертыванием в кластере: настраивает масштабирование, балансировку нагрузки и обеспечивает отказоустойчивость.

В целом Docker облегчает создание и тестирование приложений, а Kubernetes помогает управлять контейнерными приложениями в масштабе, решая задачи сетевого взаимодействия, хранения данных и жизненного цикла контейнеров.

### 3. В чем разница между реестром Docker, образом и контейнером?

**Ответ.** Образ Docker – это статический снимок конфигурации контейнера. В образ входит все необходимое для запуска приложения: код, среда выполнения, библиотеки, переменные среды и файлы конфигурации. Образы служат основой для создания контейнеров.

Контейнер Docker – это исполняемый экземпляр образа. Когда вы запускаете образ, Docker создает контейнер, который работает в изолированной среде с собственной файловой системой. Контейнер запускает приложение, определенное в образе, и может быть запущен, остановлен, перемещен или удален.

Реестр Docker – это система для хранения и доставки образов. Пользователи загружают образы в реестр с помощью команды `docker push` и получают их с помощью `docker pull`. Самый популярный публичный реестр – Docker Hub. Также существуют частные реестры, например Amazon Elastic Container Registry (ECR) и Microsoft Azure Container Registry (ACR).

### 4. Чем отличаются команды `docker start` и `docker run`?

**Ответ.** Команда `docker start` запускает ранее созданный и остановленный контейнер. Она не создает новый контейнер, а просто возобновляет работу

существующего. Конфигурация контейнера и его внутреннее состояние сохраняются такими, какими они были на момент остановки. При запуске новых слоев образа не создается. Эта команда полезна, когда нужно перезапустить контейнер без потерь данных и без изменений настроек.

Команда `docker run` используется для создания нового контейнера из указанного образа и его запуска. Если образ отсутствует в локальной системе, Docker сначала загрузит его из реестра. Эта команда сочетает несколько действий: создает контейнер, запускает его, может выделить псевдо-TTY и подключиться к стандартному вводу, выводу и стандартной ошибке, в зависимости от указанных параметров. В команде `docker run` можно задать сопоставление портов, привязку томов, переменные среды и сетевые настройки.

Команда `docker start` запускает ранее созданный и остановленный контейнер. Она не создает новый экземпляр и не меняет его конфигурацию. Контейнер запускается в том виде, в каком был остановлен, без создания новых слоев.

Коротко:

- `docker run` используют для создания и запуска нового контейнера, причем можно задать начальные параметры;
- `docker start` применяют для запуска существующего контейнера без изменений в его настройках.

## 5. Как задать базовый образ в файле `Dockerfile`?

**Ответ.** В файле `Dockerfile` команда `FROM` начинает сборку образа на основе базового образа, содержащего среду выполнения .NET:

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0
```

# Глава 16

Ниже приведены варианты ответов на эти вопросы.

## 1. Какие три типа ресурсов используются в Aspire?

**Ответ.** В Aspire предусмотрены три встроенных типа ресурсов:

- Project – проект .NET, например веб-сервис или веб-приложение ASP.NET Core;
- Container – образ контейнера, например Docker-образ для Redis или RabbitMQ;
- Executable – исполняемый файл.

## 2. Какую функцию выполняет проект `AppHost` в решении Aspire?

**Ответ.** Проект `AppHost` – это консольное приложение, которое запускает все остальные проекты и настраивает ресурсы и конечные точки. В проекте `AppHost` определяется, из каких ресурсов состоит приложение, включая проекты .NET, контейнеры, исполняемые файлы и облачные ресурсы. Кроме того, `AppHost` описывает, как эти ресурсы взаимодействуют между собой.

3. Какова роль проекта `ServiceDefaults` в решении Aspire?

**Ответ.** Проект `ServiceDefaults` — это библиотека классов, в которой централизованно настраиваются все ресурсы Aspire, включая базы данных, проекты .NET и другие компоненты.

4. Для чего используется метод `AddProject`?

**Ответ.** Метод `AddProject` добавляет ресурс проекта .NET в модель приложения Aspire. Он динамически сканирует указанный проект, чтобы найти настройки, например URL-адреса, в файле `launchSettings.json`. Например, чтобы добавить проект веб-API на ASP.NET Core, нужно вызвать метод `AddProject`:

```
var apiService = builder.AddProject<Projects.AspireStarter_ApiService>("apiservice");
```

5. Для чего используется метод `WithReference`?

**Ответ.** Метод `WithReference` связывает ресурсы, чтобы они могли правильно настраивать взаимодействие друг с другом, передавая информацию об обнаружении сервисов для указанных проектов. Например, чтобы добавить проект веб-фронтиенда, который будет обращаться к веб-сервису и контейнеру Redis, можно использовать следующий код:

```
builder.AddProject<Projects.AspireStarter_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
    .WithReference(apiService);
```

6. Какие сведения отображаются на информационной панели разработчика Aspire?

**Ответ.** Информационная панель разработчика Aspire состоит из пяти разделов.

- **Resources** (Ресурсы) — показывает контейнеры, проекты, исполняемые файлы и их состояние.
- **Console** (Консоль) — отображает текстовые журналы всех ресурсов.
- **Structured** (Структурированные) — позволяет работать со структурированными журналами, которые можно фильтровать и искать.
- **Traces** (Трассировки) — показывает распределенные трассировки действий между сервисами.
- **Metrics** (Метрики) — отображает метрики работы приложений и сервисов.

7. Какие преимущества дает использование пакета компонента Aspire вместо обычного NuGet-пакета, например для Redis?

**Ответ.** Компоненты Aspire — это библиотеки классов-оберток, которые настраивают такие функциональные возможности, как Redis, для работы в облачной среде. Они созданы, чтобы упростить облачную разработку, устранив основную сложность — необходимость вручную настраивать множество параметров, зачастую без четких рекомендаций, с чего начинать. Каждый компонент Aspire заранее определяет, какие функции он должен обеспечивать, и обязательно включает отказоустойчивость по умолчанию, проверки состояния, настройку телеметрии и интеграцию с контейнером внедрения зависимостей (DI).

8. Чем Aspire отличается от решений вроде Dapr и Orleans?

**Ответ.** Можно использовать Dapr, Orleans и Aspire вместе. Каждый из этих инструментов решает разные, но дополняющие друг друга задачи. Dapr (Distributed Application Runtime) – это проект с открытым исходным кодом, созданный в целях упрощения разработки отказоустойчивых, кросс-платформенных микросервисных приложений. Orleans – это фреймворк для построения распределенных приложений с высокой масштабируемостью. Он основан на модели виртуальных акторов, которые во фреймворке называются «зернами». Зерно является основной единицей вычислений и хранения состояния, а сам фреймворк упрощает работу с параллелизмом и распределением нагрузки.

9. Какие контейнерные технологии поддерживаются в Aspire?

**Ответ.** Aspire поддерживает Docker и Podman для работы с контейнерами. По умолчанию используется Docker.

10. Как проверить, что для доступа к базам данных, таким как PostgreSQL, используется надежный пароль?

**Ответ.** Некоторые решения Aspire требуют задания постоянного пароля вместо использования случайно сгенерированного при каждом запуске. Самый простой способ установить постоянный пароль – сохранить его в пользовательских секретах проекта AppHost с ключом `Parameters:mypassword`.

## Глава 17

Ниже приведены варианты ответов на эти вопросы.

1. Как расшифровывается аббревиатура SOLID и какие принципы она включает?

**Ответ.** SOLID включает пять основных принципов объектно-ориентированного проектирования:

- принцип единой ответственности (Single Responsibility Principle, SRP);
- принцип открытости/закрытости (Open/Closed Principle, OCP);
- принцип подстановки Лисков (Liskov Substitution Principle, LSP);
- принцип разделения интерфейса (Interface Segregation Principle, ISP);
- принцип инверсии зависимостей (Dependency Inversion Principle, DIP).

2. Какие три группы паттернов проектирования считаются основными?

**Ответ.** Три группы паттернов проектирования:

- порождающие (Creational);
- структурные (Structural);
- поведенческие (Behavioral).

3. В чем суть паттерна Стратегия? Приведите пример из базовой библиотеки классов .NET.

**Ответ.** Паттерн проектирования Стратегия описывает семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Такой подход позволяет изменять алгоритмы независимо от клиентов, которые их используют. Примером в библиотеке базовых классов .NET служат интерфейсы `IComparer` и `IComparable`.

4. Паттерн Одиночка в .NET часто реализуют через `static`-класс со свойством `Instance` и приватным конструктором. Какой способ предпочтительнее?

**Ответ.** Встроенный контейнер внедрения зависимостей (DI) в ASP.NET Core позволяет разработчикам настраивать сервисы с различным временем жизни, в том числе как одиночки. При регистрации сервиса как одиночки контейнер гарантирует, что в приложении будет создан и использоваться только один экземпляр этого сервиса. Это связано с классическим паттерном проектирования Одиночка, который ограничивает создание экземпляра класса одним объектом.

5. В чем заключается принцип проектирования, известный как закон Деметры?

**Ответ.** Закон Деметры (LoD), также известный как принцип наименьшего знания, — это руководящий принцип проектирования объектно-ориентированных систем. Он предполагает, что модуль должен знать только о своих ближайших зависимостях и взаимодействовать напрямую только с ними. На практике это означает, что объект должен избегать вызовов методов на результатах других вызовов, чтобы не создавать сильную связность кода. Кратко суть принципа можно выразить так: «Общайтесь только со своими близкими друзьями».

## Глава 18

Ниже приведены варианты ответов на эти вопросы.

1. В чем разница между архитектурой ПО и архитектурой решений?

**Ответ.** *Архитектура ПО* фокусируется на структуре и проектировании программных решений. Она охватывает определение решения, которое удовлетворяет техническим и эксплуатационным требованиям, а также оптимизирует важнейшие атрибуты качества, такие как производительность, безопасность и управляемость.

*Архитектура решения* более целостна и ориентирована на бизнес. Она включает не только ПО, но также аппаратные средства, человеческие ресурсы и процессы, необходимые для достижения бизнес-целей или выполнения конкретных бизнес-требований.

2. Кратко опишите три стиля архитектуры ПО.

**Ответ.** Существуют три стиля архитектуры ПО.

- **Модульная монолитная.** Эволюция классической монолитной архитектуры, призванная смягчить некоторые ее недостатки, сохранив при этом ее преимущества. Она организует монолитное приложение в виде модулей или компонентов, каждый из которых отвечает за определенную бизнес-область

или функциональность и в значительной степени отделен от других модулей. Хотя приложение по-прежнему развертывается как единое целое, отдельные модули могут масштабироваться более независимо, например с использованием многопоточности. Процесс развертывания остается таким же простым, как и в классическом монолите, но модульность улучшает время сборки и открывает возможности для целенаправленных оптимизаций.

- **Микросервисная.** Представьте сложную машину, состоящую из независимых компонентов, каждый из которых выполняет определенную функцию и способен работать самостоятельно. Это и есть микросервисная архитектура. Она структурирует приложение как набор сервисов, которые удобно сопровождать и тестировать, у них слабая связанность, они могут развертываться независимо друг от друга и организовываться на основе бизнес-возможностей. Микросервисы отлично подходят для крупных, сложных приложений, требующих высокой масштабируемости и гибкости. Компании Netflix и Amazon широко используют микросервисную архитектуру для масштабирования своих глобальных сервисов.
  - **Чистая.** Чистая архитектура направлена на создание систем, которые легко сопровождать, тестировать и адаптировать с течением времени, с особым акцентом на разделение ответственности. Ее применение требует строгого разделения слоев системы, особенно бизнес-логики и внешних зависимостей. Хотя для небольших проектов чистая архитектура может казаться сложной или избыточной, ее преимущества становятся очевидными по мере роста проекта. Этот подход высоко ценится в сообществе разработчиков ПО.
3. Что такое предметно-ориентированное проектирование (Domain-Driven Design, DDD)?
- Ответ.** Предметно-ориентированное проектирование (Domain-Driven Design, DDD) – это подход к разработке ПО, который ставит в центр внимания глубокое понимание и моделирование проблемной области, с которой работает система. Вместо акцента на технологиях DDD фокусируется на самой предметной области. Вот ключевые идеи DDD.
- DDD поощряет создание модели предметной области – системы абstractions, описывающей основные понятия, правила и взаимосвязи внутри проблемной области. Такая модель помогает разработчикам точнее решать задачи бизнеса и служит связующим звеном между реальным миром и кодом.
  - Одним из основополагающих принципов DDD является единый язык. Это согласованная терминология, которой пользуются как технические специалисты, так и эксперты предметной области. Единый язык применяется как при моделировании, так и при формулировке требований к системе.
  - DDD акцентируется на тесном сотрудничестве между разработчиками и экспертами бизнес-области. Вместе они постепенно уточняют концептуальную модель, чтобы как можно точнее отразить реальные процессы и требования. Такое взаимодействие помогает создавать более точные и жизнеспособные системы.

- В многослойной архитектуре объектно-ориентированных приложений слой предметной области отвечает за инкапсуляцию основной бизнес-логики и моделей домена.
  - DDD различает несколько типов моделей. Например, сущность — это объект с уникальной идентичностью (например, место в самолете с определенным номером). Объект-значение — это неизменяемый объект, важный своим содержимым, а не собственной индивидуальностью (например, визитная карточка — важна информация на карточке (ее атрибуты), а не различия между каждой уникальной карточкой).
4. Что означает сокращение CQRS и в чем суть этого подхода?

**Ответ.** Command Query Responsibility Segregation (CQRS) — это паттерн проектирования и архитектурный стиль, который разделяет модели для чтения и обновления данных. Этот подход опирается на фундаментальный принцип информатики — разделение команд и запросов (Command-Query Separation, CQS). Согласно этому принципу, каждый метод должен быть либо командой, которая выполняет действие, либо запросом, который возвращает данные вызывающей стороне, но не совмещать обе функции. CQRS развивает эту идею, распространяя разделение обязанностей на архитектурный уровень.

5. Кто такой Роберт Мартин и почему его считают важной фигурой в мире .NET-разработки?

**Ответ.** Робертом С. Мартином (он же «дядя Боб») было введено понятие чистой архитектуры. Он предложил структурированный подход к проектированию программных решений, который помогает создавать код, удобный для сопровождения и развития. Чистая архитектура — это философия проектирования ПО, которая делает акцент на четком разделении ответственности между различными слоями приложения. Такой подход позволяет строить системы, не зависящие от фреймворков, пользовательского интерфейса, базы данных и других внешних компонентов.

## Глава 19

Ниже приведены варианты ответов на эти вопросы.

1. В чем заключаются обязанности бизнес-аналитика и как .NET-разработчик может с ним взаимодействовать?

**Ответ.** Бизнес-аналитик отвечает за сбор и формализацию бизнес-требований от заинтересованных сторон, перевод этих требований в понятные технические спецификации, а также за проверку того, насколько реализованные функции соответствуют изначальным ожиданиям бизнеса. .NET-разработчик тесно взаимодействует с бизнес-аналитиком: он должен глубоко понимать бизнес-логику, изложенную бизнес-аналитиком, чтобы технические решения точно отражали потребности бизнеса. Кроме того, разработчик может консультировать

бизнес-аналитика, подсказывая более эффективные или реалистичные технические варианты реализации предложенных требований.

2. Что такое парное программирование и как оно используется на практике?

**Ответ.** Парное программирование — это метод разработки ПО, при котором два программиста работают вместе за одним компьютером. Один из них, «водитель», набирает код, сосредоточившись на его написании. Второй, «наблюдатель» или «штурман», внимательно следит за процессом, анализирует код и предлагает улучшения или замечает возможные ошибки. Роли периодически меняются, что способствует совместной работе, взаимному обучению и более эффективному решению задач.

3. Может ли языковая модель вроде ChatGPT или Llama 3 заменить .NET-разработчика?

**Ответ.** Llama 3 и ChatGPT — это модели класса LLM (Large Language Model, большая языковая модель) — мощные инструменты для анализа и генерации естественного языка. Однако их роль и возможности отличаются от задач .NET-разработчика. LLM разработаны для решения задач обработки естественного языка, таких как генерация человекоподобных ответов на запросы, предоставление справочной информации, перевод текстов, сжатие длинных материалов в краткие резюме и определение эмоциональной окраски текстов.

.NET-разработчик — это специалист, который разрабатывает программные решения на основе платформы .NET, используя такие технологии, как C#, ASP.NET и др. Он тесно сотрудничает с экспертами предметной области, менеджерами проектов и другими членами команды, чтобы создавать надежные и поддерживаемые системы.

LLM могут помочь разработчикам, например, при составлении документации, написании комментариев к коду или формулировании требований на естественном языке. Однако создание сложных программных решений требует технических знаний, понимания бизнес-потребностей и умения проектировать эффективные системы, чем обладают именно разработчики.

LLM не способны самостоятельно писать сложные алгоритмы, оптимизировать код для повышения производительности или решать низкоуровневые технические задачи. .NET-разработчики также интегрируют базы данных, сторонние API и библиотеки, обеспечивая стабильность и безопасность систем. Разработка ПО часто предполагает сотрудничество, общение и принятие решений — области, в которых LLM не справляются.

Таким образом, LLM, такие как Llama 3 и ChatGPT, могут выступать в роли вспомогательных инструментов, ускоряя работу и облегчая некоторые рутинные задачи. Однако заменить опытного .NET-разработчика они не могут.

4. Какие вопросы обычно задают на собеседовании?

**Ответ.** Вопросы на собеседовании обычно направлены на оценку широты и глубины технических навыков кандидата, а также на его соответствие культуре компании. Основные типы вопросов:

- **на соответствие должности:** вопросы, проверяющие наличие специфических навыков и знаний, необходимых для выполнения работы;
  - **на проверку навыков решения задач:** задания, позволяющие оценить способность кандидата находить решения под давлением и в ограниченное время;
  - **на оценку потенциала для роста:** вопросы, помогающие выяснить, насколько кандидат готов учиться, развиваться и осваивать новые области;
  - **на культурное соответствие:** вопросы, которые позволяют понять, разделяет ли кандидат ценности компании и сможет ли органично вписаться в команду.
5. Почему при изучении важно видеть не только правильные, но и ошибочные ответы?
- Ответ.** Умение видеть как правильные, так и неправильные ответы при изучении темы дает ряд преимуществ.
- **Контраст:** сравнивая правильные и неправильные ответы, вы лучше понимаете тонкости и границы темы.
  - **Распространенные ошибки:** анализируя неправильные варианты, вы узнаете о типичных заблуждениях кандидатов и избегаете их в будущем.
  - **Критическое мышление:** оценка неверных ответов развивает способность анализировать доводы и сомневаться в предположениях.
  - **Коренные причины:** изучение ошибок помогает выявить базовые концепции, которые требуют более глубокого понимания.
  - **Промежуточное повторение:** многократное сопоставление верных и неверных ответов укрепляет запоминание.
  - **Активные воспоминания:** исправление ошибок заставляет мозг работать активнее, укрепляя нейронные связи.
  - **Реальные сценарии:** в реальной жизни вопросы часто формулируются неидеально, и работа с ошибками помогает подготовиться к неоднозначности.
  - **Обратная связь:** неправильные ответы дают полезную обратную связь и указывают на области, где нужно больше практики.
  - **Предотвращение излишней самоуверенности:** регулярные ошибки напоминают, что обучение — это процесс, и помогают сохранять здравый баланс уверенности.
  - **Открытость к исправлениям:** готовность видеть и исправлять ошибки способствует развитию гибкого мышления и росту компетентности.
- В целом работа с ошибками делает обучение глубже, усиливает критическое мышление и помогает лучше усваивать материал.



Марк Прайс

**.NET 8: приложения и сервисы.  
Практика создания проектов  
с использованием Blazor,  
.NET MAUI, gRPC, GraphQL. 2-е изд.**

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

