

ASP.NET Core

в действии

Третье издание

Эндрю Лок



MANNING



DOT
NET
.RU

Эндрю Лок

ASP.NET Core в действии

Third edition

ASP.NET Core in Action

ANDREW LOCK



MANNING
Shelter Island

Третье издание

ASP.NET Core в действии

ЭНДРЮ ЛОК



Москва, 2025

УДК 004.438.NET
ББК 32.973.26-018.2
Л73

Под редакцией сообщества .NET разработчиков DotNet.Ru

Эндрю Лок

Л73 ASP.NET Core в действии / пер. с англ. Д. А. Беликова. 3-е изд. – М.: ДМК Пресс, 2024. – 1046 с.: ил.

ISBN 978-5-93700-183-2

Эта книга знакомит читателей с основами фреймворка ASP.NET Core, такими как промежуточное ПО, внедрение зависимостей и конфигурация. Автор показывает, как настроить их в соответствии с пользовательскими требованиями. Речь пойдет о том, как добавить аутентификацию и авторизацию в свои приложения, как повысить их безопасность, а также как развертывать их и осуществлять мониторинг. Рассматривается тестирование приложений с использованием модульных и интеграционных тестов. Основное внимание будет уделено тому, как создавать приложения с отрисовкой на стороне сервера, используя страницы Razor и веб-API, а также контроллеры MVC.

В третьем издании показано, как создавать веб-приложения для эксплуатации в промышленном окружении с помощью ASP.NET Core 7.0. Вы будете учиться на практических примерах, содержательных иллюстрациях и коде с подробными пояснениями. В числе новинок: создание минимальных API, обеспечение безопасности API с помощью токенов на предъявителя, WebApplicationBuilder и многое другое.

Книга подойдет как тем, кто является новичком в веб-разработке, так и тем, кто уже имеет опыт использования фреймворка ASP.NET.

УДК 004.438.NET
ББК 32.973.26-018.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 978-1-63343-862-0
ISBN (рус.) 978-5-93700-183-2

© 2023 by Manning Publications Co.
© Оформление, издание, перевод,
ДМК Пресс, 2024

Краткое оглавление

1 ■ <i>Начало работы с ASP.NET Core</i>	34
ЧАСТЬ I. Начало работы с минимальными API	
2 ■ <i>Что такое ASP.NET Core</i>	47
3 ■ <i>Наше первое приложение.....</i>	67
4 ■ <i>Обработка ошибок с помощью конвейера промежуточного ПО.....</i>	94
5 ■ <i>Создание JSON API с помощью минимальных API</i>	126
6 ■ <i>Сопоставление URL-адресов с конечными точками с помощью маршрутизации.....</i>	163
7 ■ <i>Привязка модели и валидация в минимальных API</i>	187
ЧАСТЬ II. Создание полноценных приложений.....221	
8 ■ <i>Введение во внедрение зависимостей</i>	223
9 ■ <i>Регистрация сервисов с помощью внедрения зависимостей.....</i>	242
10 ■ <i>Конфигурирование приложения ASP. Net Core.....</i>	269
11 ■ <i>Документирование API с помощью OpenAPI</i>	306
12 ■ <i>Сохранение данных с Entity Framework Core</i>	340
ЧАСТЬ III. Генерация HTML-кода с помощью Razor Pages и MVC.....377	
13 ■ <i>Создание сайта с помощью страниц Razor.....</i>	379
14 ■ <i>Сопоставление URL-адресов с Razor Pages с использованием маршрутизации</i>	405
15 ■ <i>Создание ответов с помощью обработчиков страниц в Razor Pages</i>	427
16 ■ <i>Привязка и валидация запросов с помощью Razor Pages</i>	445
17 ■ <i>Отрисовка HTML-кода с использованием представлений Razor</i>	476
18 ■ <i>Создание форм с помощью тег-хелперов</i>	508
19 ■ <i>Создание сайта с использованием контроллеров MVC</i>	541

20 ■ <i>Создание HTTP API с использованием контроллеров веб-API</i>	562
21 ■ <i>Конвейер фильтров MVC и Razor Pages</i>	595
22 ■ <i>Создание собственных фильтров MVC и страниц Razor</i>	612
ЧАСТЬ IV. Защита и развертывание приложений.....	639
23 ■ <i>Аутентификация: добавление пользователей в приложение с помощью ASP.NET Core Identity</i>	641
24 ■ <i>Авторизация: обеспечиваем защиту приложения.....</i>	677
25 ■ <i>Аутентификация и авторизация для API</i>	713
26 ■ <i>Мониторинг и устранение ошибок с помощью журналирования</i>	747
27 ■ <i>Публикация и развертывание приложения</i>	780
28 ■ <i>Добавляем протокол HTTPS в приложение.</i>	805
29 ■ <i>Повышаем безопасность приложения</i>	826
ЧАСТЬ V. Дальнейшая работа с ASP.NET Core	855
30 ■ <i>Создание приложений ASP.NET Core с помощью универсального узла и класса Startup</i>	857
31 ■ <i>Расширенная настройка ASP.NET Core</i>	877
32 ■ <i>Создание специальных компонентов MVC и Razor Pages</i>	907
33 ■ <i>Вызов удаленных API с помощью IHttpClientFactory</i>	937
34 ■ <i>Создание фоновых задач и сервисов</i>	960
35 ■ <i>Тестирование приложений с xUnit</i>	989
36 ■ <i>Тестирование приложений ASP.NET Core</i>	1006

Оглавление

Предисловие от издательства	21
Вступительное слово от сообщества DotNet.Ru.....	22
Предисловие	24
Благодарности	26
Об этой книге	28
Об авторе	33
Об иллюстрации на обложке.....	33
1 Начало работы с ASP.NET Core	34
1.1 Что такое ASP.NET Core?.....	34
1.2 Какие типы приложений можно создавать?	35
1.3 Выбор ASP.NET Core	36
1.4 Как работает ASP.NET Core?	38
1.4.1 Как работает веб-запрос по протоколу HTTP?	38
1.4.2 Как ASP.NET Core обрабатывает запрос?	41
1.5 Что вы узнаете из этой книги	42
Резюме	43
ЧАСТЬ I. Начало работы с минимальными API	45
2 Что такое ASP.NET Core.....	47
2.1 Использование фреймворка для создания веб-приложений.....	47
2.2 Для чего был создан ASP.NET Core.....	48
2.3 Парадигмы ASP.NET Core	53
2.4 Когда следует выбирать ASP.NET Core.....	56
2.4.1 Если вы новичок в разработке .NET	57
2.4.2 Если вы .NET-разработчик, создающий новое приложение.....	59
2.4.3 Перенос существующего ASP.NET-приложения на ASP.NET Core.....	64
Резюме	66
3 Наше первое приложение.....	67
3.1 Краткий обзор приложения ASP.NET Core.....	68
3.2 Создаем наше первое приложение ASP.NET Core	70
3.2.1 Использование шаблона	72
3.2.2 Сборка приложения.....	75
3.3 Запуск веб-приложения	76
3.4 Разбираемся с шаблоном проекта	77
3.5 Файл проекта .csproj: объявление зависимостей	79
3.6 Файл Program.cs: определение приложения	81
3.7 Добавляем функциональность в приложение	84
3.7.1 Добавление и настройка сервисов	87
3.7.2 Определение способа обработки запросов с помощью промежуточного ПО и конечных точек.....	89
Резюме	92

4 Обработка ошибок с помощью конвейера промежуточного ПО ... 94

4.1	Что такое промежуточное ПО	96
4.2	Объединение компонентов в конвейер	100
4.2.1	Простой сценарий конвейера 1: страница приветствия.....	101
4.2.2	Простой сценарий конвейера 2: обработка статических файлов	104
4.2.3	Простой сценарий конвейера 3: приложение с минимальным API	108
4.3	Обработка ошибок с помощью промежуточного ПО	115
4.3.1	Просмотр исключений в окружении разработки: DeveloperExceptionPage	117
4.3.2	Обработка исключений в промышленном окружении: ExceptionHandlerMiddleware	119
	Резюме	124

5 Создание JSON API с помощью минимальных API 126

5.1	Что такое HTTP API и когда его следует использовать?	127
5.2	Определение конечных точек минимальных API	131
5.2.1	Извлечение значений из URL-адреса с помощью маршрутизации.....	132
5.2.2	Сопоставление HTTP-методов с конечными точками.....	133
5.2.3	Определение обработчиков маршрутов с помощью функций...	135
5.3	Генерация ответов с помощью IResult	138
5.3.1	Возврат кодов состояния с помощью Results и TypedResults.....	139
5.3.2	Возврат полезных данных об ошибках с помощью Problem Details.....	141
5.3.3	Преобразование всех ответов в Problem Details.....	143
	Преобразование исключений в Problem Details	144
	Преобразование кодов состояния ошибок в формат Problem Details	146
5.4	Запуск общего кода с фильтрами конечных точек	148
5.4.1	Добавление нескольких фильтров к конечной точке	151
5.4.2	Фильтры или промежуточное ПО: что выбрать?	153
5.4.3	Обобщенные фильтры конечных точек.....	154
5.4.4	Реализация интерфейса IEndpointFilter	157
5.5	Организация API с помощью групп маршрутов	158
	Резюме	161

6 Сопоставление URL-адресов с конечными точками с помощью маршрутизации 163

6.1	Что такое маршрутизация?	164
6.2	Маршрутизация конечных точек в ASP.NET Core	168
6.3	Изучение синтаксиса шаблона маршрута	172
6.3.1	Работа с параметрами и литеральными сегментами.....	172
6.3.2	Использование необязательных значений и значений по умолчанию.....	174
6.3.3	Добавление дополнительных ограничений к параметрам маршрута.....	175

6.3.4 Сопоставление произвольных URL-адресов с помощью универсального параметра	178
6.4 Генерация URL-адресов из параметров маршрута.....	179
6.4.1 Генерация URL-адресов для конечной точки минимального API с помощью класса LinkGenerator	180
6.4.2 Генерация URL-адресов с помощью IResults.....	182
6.4.3 Управление созданными URL-адресами с помощью RouteOptions	183
Резюме	185
7 Привязка модели и валидация в минимальных API	187
7.1 Извлечение значений из запроса с привязкой модели.....	188
7.2 Привязка простых типов к запросу.....	190
7.3 Привязка сложных типов к телу запроса в формате JSON.....	195
7.4 Массивы: простые типы или сложные?.....	197
7.5 Делаем параметры необязательными с помощью типов, допускающих значение NULL	200
7.6 Привязка сервисов и специальных типов.....	203
7.6.1 Внедрение хорошо известных типов.....	203
7.6.2 Внедрение сервисов	204
7.6.3 Привязка загрузки файлов с помощью IFormFile и IFormFileCollection	205
7.7 Собственная привязка с помощью BindAsync	207
7.8 Выбор источника привязки.....	208
7.9 Упрощение обработчиков с помощью AsParameters.....	209
7.10 Обработка пользовательского ввода с помощью валидации модели....	211
7.10.1 Необходимость валидации модели	211
7.10.2 Использование атрибутов DataAnnotations для валидации	212
7.10.3 Добавление фильтра валидации в минимальные API	215
Резюме	218
ЧАСТЬ II. Создание полноценных приложений	221
8 Введение во внедрение зависимостей	223
8.1 Преимущества внедрения зависимостей	224
8.2 Создание слабосвязанного кода.....	231
8.3 Использование внедрения зависимостей в ASP.NET Core	233
8.4 Добавление сервисов ASP.NET Core в контейнер.....	235
8.5 Использование сервисов из контейнера внедрения зависимостей ...	238
Резюме	240
9 Регистрация сервисов с помощью внедрения зависимостей	242
9.1 Регистрация собственных сервисов в контейнере внедрения зависимостей	243
9.2 Регистрация сервисов с использованием объектов и лямбда-функций	247
9.3 Многократная регистрация сервиса в контейнере	251
9.3.1 Внедрение нескольких реализаций интерфейса	252

9.3.2 Внедрение одной реализации при регистрации нескольких сервисов	254
9.3.3 Условная регистрация сервисов с помощью TryAdd	254
9.4 Жизненный цикл: когда создаются сервисы?.....	255
9.4.1 Transient: уникален каждый	258
9.4.2 Scoped: держимся вместе	259
9.4.3 Singleton: может быть только один	260
9.4.4 Следите за захваченными зависимостями	262
9.5 Разрешение сервисов с жизненным циклом scoped за пределами запроса.....	265
Резюме	267

10*Конфигурирование приложения ASP. Net Core* **269**

10.1 Представляем модель конфигурации ASP.NET Core	270
10.2 Создание объекта конфигурации для приложения.....	272
10.2.1 Добавление поставщика конфигурации в файле Program.cs	275
10.2.2 Использование нескольких поставщиков для переопределения значений конфигурации.....	278
10.2.3 Безопасное хранение секретов конфигурации.....	280
Сохранение секретов в переменных окружения в промышленном окружении	281
Хранение секретов с помощью менеджера User Secrets в окружении разработки	282
10.2.4 Перезагрузка значений конфигурации при их изменении.....	284
10.3 Использование строго типизированных настроек с паттерном «Параметры».....	286
10.3.1 Знакомство с интерфейсом IOptions.....	288
10.3.2 Перезагрузка строго типизированных параметров с помощью IOptionsSnapshot.....	289
10.3.3 Разработка классов параметров для автоматической привязки	291
10.3.4 Привязка строго типизированных параметров без интерфейса IOptions.....	293
10.4 Настройка приложения для нескольких окружений	295
10.4.1 Определение окружения размещения.....	296
10.4.2 Загрузка файлов конфигурации для конкретного окружения ...	297
10.4.3 Задаем окружение размещения.....	299
Резюме	303

11*Документирование API с помощью OpenAPI* **306**

11.1 Добавление описания OpenAPI в приложение.....	307
11.2 Тестирование API с помощью Swagger UI	310
11.3 Добавление метаданных в минимальные API.....	312
11.4 Создание строго типизированных клиентов с помощью NSwag	315
11.4.1 Генерация клиента с помощью Visual Studio.....	316
11.4.2 Создание клиента с помощью инструмента .NET Global	320
11.4.3 Использование генерированного клиента для вызова API	322
11.4.4 Настройка генерированного кода.....	323
11.4.5 Обновление описания OpenAPI	326

11.5 Добавление описаний и сводок в конечные точки	327
11.5.1 Использование текущих методов для добавления описаний....	328
11.5.2 Использование атрибутов для добавления метаданных.....	330
11.6 Ограничения OpenAPI.....	334
11.6.1 Не все API можно описать, используя OpenAPI	334
11.6.2 Сгенерированный код чрезмерно категоричен	335
11.6.3 Инструменты часто отстают от спецификации.....	336
Резюме	337

12 Сохранение данных с Entity Framework Core 340

12.1 Знакомство с Entity Framework Core	342
12.1.1 Что такое EF Core?	342
12.1.2 Зачем использовать инструмент объектно-реляционного отображения?.....	344
12.1.3 Когда следует выбирать EF Core?	345
12.1.4 Отображение базы данных в код приложения	346
12.2 Добавляем EF Core в приложение	348
12.2.1 Выбор провайдера базы данных и установка EF Core.....	350
12.2.2 Создание модели данных	351
12.2.3 Регистрация контекста данных	354
12.3 Управление изменениями с помощью миграций.....	355
12.3.1 Создаем первую миграцию	356
12.3.2 Добавляем вторую миграцию	359
12.4 Выполнение запроса к базе данных и сохранение в ней данных	362
12.4.1 Создание записи	363
12.4.2 Загрузка списка записей.....	365
12.4.3 Загрузка отдельной записи	367
12.4.4 Обновление модели с изменениями	369
12.5 Использование EF Core в промышленных приложениях	373
Резюме	375

ЧАСТЬ III. Генерация HTML-кода с помощью Razor Pages и MVC 377

13 Создание сайта с помощью Razor Pages 379

13.1 Наше первое приложение Razor Pages	380
13.1.1 Использование шаблона «Веб-приложение».....	380
13.1.2 Добавление и настройка сервисов	384
13.1.3 Создание HTML с помощью Razor Pages.....	386
13.1.4 Логика обработки запросов с помощью моделей страницы и обработчиков.....	388
13.2 Изучение типичной страницы Razor	390
13.3 Паттерн проектирования MVC.....	393
13.4 Применение паттерна проектирования MVC к Razor Pages.....	395
13.4.1 Направление запроса на страницу Razor и создание модели привязки	397
13.4.2 Выполнение обработчика с использованием модели приложения	399
13.4.3 Генерация HTML-кода с использованием модели представления	400
13.4.4 Собираем все вместе: полный запрос страницы Razor.....	401
Резюме	403

14 Сопоставление URL-адресов с Razor Pages с использованием маршрутизации	405
14.1 Маршрутизация в ASP.NET Core	406
14.2 Маршрутизация на основе соглашений и явная маршрутизация	407
14.3 Маршрутизация запросов в Razor Pages	410
14.4 Настройка шаблонов маршрутов страницы Razor	412
14.4.1 Добавление сегмента в шаблон маршрута	413
14.4.2 Полная замена шаблона маршрута	414
14.5 Генерация URL-адресов для страниц Razor	415
14.5.1 Генерация URL-адресов для страницы Razor	415
14.5.2 Генерация URL-адресов для контроллера MVC	416
14.5.3 Генерация URL-адресов с помощью класса LinkGenerator	418
14.6 Настройка соглашений с помощью Razor Pages	419
Резюме	425
15 Создание ответов с помощью обработчиков страниц в Razor Pages.....	427
15.1 Razor Pages и обработчики страниц	428
15.2 Выбор обработчика страницы для вызова	429
15.3 Прием параметров в обработчиках страниц	432
15.4 Возврат ответов IActionResult	434
15.4.1 PageResult и RedirectToPageResult	436
15.4.2 NotFoundResult и StatusCodeResult	436
15.5 Обработка кодов состояния с помощью StatusCodePagesMiddleware	438
Резюме	443
16 Привязка и валидация запросов с помощью Razor Pages	445
16.1 Модели в Razor Pages и MVC	446
16.2 От запроса к модели: делаем запрос полезным	449
16.2.1 Связывание простых типов	454
16.2.2 Связывание сложных типов	457
Упрощение параметров метода привязкой к сложным объектам	458
Привязка коллекций и словарей	459
Привязка загрузки файлов с помощью IFormFile	461
16.2.3 Выбор источника привязки	461
16.3 Валидация моделей привязки	464
16.3.1 Валидация в Razor Pages	464
16.3.2 Валидация на сервере в целях безопасности	466
16.3.3 Валидация на стороне клиента для улучшения пользовательского опыта	470
16.4 Организация моделей привязки в Razor Pages	471
Резюме	474
17 Отрисовка HTML-кода с использованием представлений Razor	476
17.1 Представления: отрисовка пользовательского интерфейса	478

17.2 Создание представлений Razor.....	482
17.2.1 Представления Razor и сопутствующий код	482
17.2.2 Знакомство с шаблонами Razor.....	483
17.2.3 Передача данных в представления	485
17.3 Создание динамических веб-страниц с помощью Razor.....	488
17.3.1 Использование кода C# в шаблонах Razor	489
17.3.2 Добавление циклов и условий в шаблоны Razor.....	490
17.3.3 Отрисовка HTML-кода с помощью метода Raw	493
17.4 Макеты, частичные представления и _ViewStart	495
17.4.1 Использование макетов для общей разметки	496
17.4.2 Переопределение родительских макетов с помощью секций....	499
17.4.3 Использование частичных представлений для инкапсуляции разметки	501
17.4.4 Выполнение кода в каждом представлении с помощью _ViewStart и _ViewImports.....	504
Импорт общих директив с помощью _ViewImports.....	504
Выполнение кода для каждого представления с помощью _ViewStart	505
Резюме	506

18 Создание форм с помощью тег-хелперов..... 508

18.1 Редакторы кода и тег-хелперы	510
18.2 Создание форм с помощью тег-хелперов	513
18.2.1 Тег-хелпер формы	518
18.2.2 Тег-хелпер метки	519
18.2.3 Тег-хелперы текстового ввода.....	521
18.2.4 Тег-хелпер раскрывающегося списка.....	525
18.2.5 Тег-хелперы сообщений валидации и сводок валидации	531
18.3 Создание ссылок с помощью тег-хелпера привязки ссылки	534
18.4 Сброс кеша с помощью тег-хелпера добавления версии	535
18.5 Использование условной разметки с помощью тег-хелпера окружения.....	537
Резюме	538

19 Создание сайта с использованием контроллеров MVC 541

19.1 Razor Pages и MVC в ASP.NET Core	542
19.2 Наше первое веб-приложение MVC	543
19.3 Сравнение контроллера MVC с PageModel из Razor Page	547
19.4 Выбор представления из контроллера MVC	549
19.5 Выбор между Razor Pages и контроллерами MVC.....	556
19.5.1 Преимущества Razor Page	556
19.5.2 Когда выбирать контроллеры MVC вместо Razor Pages	559
Резюме	561

20 Создание HTTP API с использованием контроллеров веб-API 562

20.1 Создание первого проекта веб-API.....	563
20.2 Применение паттерна проектирования MVC к веб-API.....	570
20.3 Маршрутизация на основе атрибутов: связывание методов действий с URL-адресами	574

20.3.1 Сочетание атрибутов маршрута для следования принципу DRY	576
20.3.2 Использование замены маркера для уменьшения дублирования при маршрутизации на основе атрибутов.....	578
20.3.3 Обработка HTTP-методов с помощью маршрутизации на основе атрибутов.....	579
20.4 Использование общепринятых соглашений с атрибутом [ApiController]	580
20.5 Генерация ответа от модели	583
20.5.1 Настройка форматеров по умолчанию: добавляем поддержку XML.....	586
20.5.2 Выбор формата ответа с помощью согласования типа содержимого	587
20.6 Контроллеры веб-API и минимальные API: что выбрать?.....	589
Резюме	592
21 Конвейер фильтров MVC и Razor Pages	595
21.1 Что такое конвейер фильтров MVC.....	596
21.2 Конвейер фильтров Razor Pages	599
21.3 Фильтры или промежуточное ПО: что выбрать?	601
21.4 Создание простого фильтра	602
21.5 Добавляем фильтры к действиям и страницам Razor Pages.....	605
21.6 Порядок выполнения фильтров	608
21.6.1 Порядок выполнения фильтров по умолчанию.....	608
21.6.2 Переопределение порядка выполнения фильтров по умолчанию с помощью интерфейса IOrderedFilter	609
Резюме	610
22 Создание собственных фильтров MVC и страниц Razor	612
22.1 Создание собственных фильтров для приложения.....	613
22.1.1 Фильтры авторизации: защита API	616
22.1.2 Фильтры ресурсов: прерывание выполнения методов действий.....	617
22.1.3 Фильтры действий: настройка привязки модели и результатов действий	619
22.1.4 Фильтры исключений: собственная обработка исключений для методов действий.....	625
22.1.5 Фильтры результатов: настройка результатов действий перед их выполнением	627
22.1.6 Фильтры страниц: настройка привязки модели для Razor Pages.....	629
22.2 Прерывание выполнения конвейера.....	631
22.3 Использование внедрения зависимостей с атрибутами фильтра	634
Резюме	637
ЧАСТЬ IV. Защита и развертывание приложений	639
23 Аутентификация: добавление пользователей в приложение с помощью ASP.NET Core Identity.....	641

23.1 Знакомство с аутентификацией и авторизацией.....	643
23.1.1 Пользователи и утверждения в ASP.NET Core.....	643
23.1.2 Аутентификация в ASP.NET Core: сервисы и промежуточное ПО	644
Вход в приложение ASP.NET Core	645
Аутентификация пользователей для последующих запросов	646
23.2 Что такое ASP.NET Core Identity?	649
23.3 Создание проекта, в котором используется ASP.NET Core Identity....	652
23.3.1 Создание проекта из шаблона.....	652
23.3.2 Изучение шаблона в Обозревателе решений	654
23.3.3 Модель данных ASP.NET Core Identity	658
23.3.4 Взаимодействие с ASP.NET Core Identity	660
23.4 Добавляем ASP.NET Core Identity в существующий проект.....	663
23.4.1 Настройка сервисов ASP.NET Core Identity и промежуточного ПО	664
23.4.2 Обновление модели данных EF Core для поддержки Identity ...	665
23.4.3 Обновление представлений Razor для связи с пользовательским интерфейсом Identity	666
23.5 Настройка страницы по умолчанию в пользовательском интерфейсе ASP.NET Core Identity	668
23.6 Управление пользователями: добавление специальных данных для пользователей	672
Резюме	674

24 Авторизация: обеспечиваем защиту приложения 677

24.1 Знакомство с авторизацией.....	679
24.2 Авторизация в ASP.NET Core	682
24.2.1 Предотвращение доступа анонимных пользователей к приложению.....	683
24.2.2 Обработка запросов, не прошедших аутентификацию	685
24.3 Использование политик для авторизации на основе утверждений... 689	
24.4 Создание пользовательских политик авторизации.....	692
24.4.1 Требования и обработчики: строительные блоки политики	693
24.4.2 Создание политики со специальным требованием и обработчиком	694
Создание IAuthorizationRequirement для представления требования	694
Создание политики с несколькими требованиями	695
Создаем обработчики авторизации, чтобы отвечать требованиям	696
24.5 Управление доступом с авторизацией на основе ресурсов	700
24.5.1 Ручная авторизация запросов с помощью IAuthorizationService.....	702
24.5.2 Создание обработчика AuthorizationHandler на основе ресурсов.....	705
24.6 Скрытие HTML-элементов от неавторизованных пользователей	708
Резюме	711

25 Аутентификация и авторизация для API 713

25.1 Аутентификация для API и распределенных приложений	714
25.1.1 Распространение аутентификации на несколько приложений	714

25.1.2 Централизация аутентификации в поставщике идентификационной информации	716
25.1.3 OpenID Connect и OAuth 2.0	719
25.2 Аутентификация с токеном на предъявителя	722
25.3 Добавление аутентификации на основе JWT-токенов на предъявителя в минимальные API	729
25.4 Использование инструмента user-jwts для локального тестирования JWT-токенов	731
25.4.1 Создание JWT-токенов с помощью инструмента user-jwts.....	732
25.4.2 Настройка JWT-токенов.....	734
25.4.3 Управление локальными JWT	735
25.5 Описание требований к аутентификации для OpenAPI	736
25.6 Применение политик авторизации к конечным точкам минимальных API	740
Резюме	744

26 Мониторинг и устранение ошибок с помощью журналирования 747

26.1 Эффективное использование журналирования в промышленном приложении	749
26.1.1 Выявление проблем с помощью специальных сообщений журнала	750
26.1.2 Абстракции журналирования ASP.NET Core	752
26.2 Добавление сообщений журнала в ваше приложение	753
26.2.1 Уровень сообщения журнала: насколько важно сообщение журнала?	757
26.2.2 Категория журнала: какой компонент создал журнал?	759
26.2.3 Форматирование сообщений и сбор значений параметров...	760
26.3 Контроль места записи журналов с помощью поставщиков журналирования	762
26.4 Изменение избыточности сообщений журналов с помощью фильтрации	766
26.5 Структурное журналирование: создание полезных сообщений журналов с возможностью поиска	771
26.5.1 Добавление поставщика структурного журналирования в приложение	773
26.5.2 Использование областей журналирования для добавления дополнительных свойств в сообщения журнала.....	776
Резюме	778

27 Публикация и развертывание приложения 780

27.1 Что такое модель хостинга ASP.NET Core	781
27.1.1 Запуск и публикация приложения ASP.NET Core.....	784
27.1.2 Выбор метода развертывания для приложения.....	788
27.2 Публикация приложения в IIS	790
27.2.1 Конфигурирование IIS для ASP.NET Core	790
27.2.2 Подготовка и публикация приложения в IIS	792
27.3 Размещение приложения в Linux.....	795

27.3.1 Запуск приложения ASP.NET Core за обратным прокси-сервером в Linux	795
27.3.2 Подготовка приложения к развертыванию в Linux.....	798
27.4 Настройка URL-адресов приложения	800
Резюме	804
28 Добавляем протокол <i>HTTPS</i> в приложение.....	805
28.1 Для чего нужен протокол HTTPS?.....	806
28.2 Использование HTTPS-сертификатов для разработки	810
28.3 Настройка Kestrel для использования сертификата HTTPS в промышленном окружении	813
28.4 Делаем так, чтобы протокол HTTPS использовался для всего приложения	815
28.4.1 Принудительное использование протокола HTTPS с помощью заголовков HTTP Strict Transport Security.....	816
28.4.2 Переадресация с HTTP на HTTPS с помощью компоненты <code>HttpsRedirectionMiddleware</code>	820
28.4.3 Отклонение HTTP-запросов в приложениях API.....	823
Резюме	824
29 Повышаем безопасность приложения	826
29.1 Защита от межсайтового скрипtingа	827
29.2 Защита от межсайтовой подделки запросов (CSRF)	831
29.3 Вызов веб-API из других доменов с помощью CORS	837
29.3.1 Что такое CORS и как он работает	838
29.3.2 Добавление глобальной политики CORS ко всему приложению.....	840
29.3.3 Добавление CORS к определенным конечным точкам с помощью метаданных <code>EnableCors</code>	842
29.3.4 Настройка политик CORS	844
29.4 Изучение других векторов атак	846
29.4.1 Обнаружение и предотвращение атак с открытым перенаправлением	846
29.4.2 Предотвращение атак с использованием внедрения SQL-кода с помощью EF Core и параметризации	848
29.4.3 Предотвращение небезопасных прямых ссылок на объекты....	850
29.4.4 Защита паролей и данных пользователей.....	851
Резюме	853
ЧАСТЬ V. Дальнейшая работа с ASP.NET Core	855
30 Создание приложений ASP.NET Core с помощью обобщенного хоста и класса <i>Startup</i>	857
30.1 Разделение ответственостей между двумя файлами	858
30.2 Класс <code>Program</code> : сборка веб-хоста.....	859
30.3 Класс <code>Startup</code> : настройка приложения.....	861
30.4 Создание собственного экземпляра <code>IHostBuilder</code>	866
30.5 Сложность обобщенного хоста	869

30.6 Выбор между обобщенным хостом и минимальным хостингом	872
Резюме	874

31 Расширенная настройка ASP.NET Core..... 877

31.1 Настройка конвейера промежуточного ПО	878
31.1.1 Создание простых приложений с помощью метода расширения Run	879
31.1.2 Ветвление конвейера с помощью метода расширения Map	881
31.1.3 Добавление в конвейер с помощью метода расширения Use....	885
31.1.4 Создание пользовательского компонента промежуточного программного обеспечения	888
31.1.5 Преобразование промежуточного ПО в конечные точки	892
31.2 Использование внедрения зависимостей с OptionsBuilder и IConfigureOptions	896
31.3 Использование стороннего контейнера внедрения зависимостей ...	902
Резюме	905

32 Создание собственных компонентов MVC и Razor Pages..... 907

32.1 Создание собственного тег-хелпера Razor.....	908
32.1.1 Вывод информации об окружении с помощью собственного тег-хелпера	909
32.1.2 Создание специального тег-хелпера для условного скрытия элементов	913
32.1.3 Создание тег-хелпера для преобразования Markdown в HTML....	915
32.2 Компоненты представления: добавление логики в частичные представления	917
32.3 Создание собственного атрибута валидации.....	923
32.4 Замена фреймворка валидации на FluentValidation	928
32.4.1 Сравнение FluentValidation и атрибутов DataAnnotations	929
32.4.2 Добавляем FluentValidation в приложение	933
Резюме	934

33 Вызов удаленных API с помощью IHttpClientFactory..... 937

33.1 Вызов API по протоколу HTTP: проблема с классом HttpClient	938
33.2 Создание экземпляров класса HttpClient с помощью интерфейса IHttpClientFactory	944
33.2.1 Использование IHttpClientFactory для управления жизненным циклом HttpClientHandler	944
33.2.2 Настройка именованных клиентов во время регистрации	948
33.2.3 Использование типизированных клиентов для инкапсуляции HTTP-вызовов.....	949
33.3 Обработка временных ошибок HTTP с помощью библиотеки Polly....	952
33.4 Создание специального обработчика HttpResponseMessageHandler.....	955
Резюме	958

34 Создание фоновых задач и сервисов..... 960

34.1 Запуск фоновых задач с помощью IHostedService	961
34.1.1 Запуск фоновых задач по таймеру	962

34.1.2 Использование сервисов с жизненным циклом Scoped в фоновых задачах	967
34.2 Создание сервисов рабочей роли без пользовательского интерфейса с использованием IHost	969
34.2.1 Создание сервиса рабочей роли из шаблона.....	971
34.2.2 Запуск сервисов рабочей роли в промышленном окружении	974
34.3 Координация фоновых задач с помощью Quartz.NET.....	977
34.3.1 Установка Quartz.NET в приложение ASP.NET Core	978
34.3.2 Настройка задания для запуска по расписанию с помощью Quartz.NET	980
34.3.3 Использование кластеризации для повышения избыточности фоновых задач.....	983
Резюме	987

35 Тестирование приложений с помощью xUnit..... 989

35.1 Тестирование в ASP.NET Core	990
35.2 Создание первого тестового проекта с xUnit.....	992
35.3 Запуск тестов командой dotnet test.....	995
35.4 Ссылка на ваше приложение из тестового проекта	996
35.5 Добавление модульных тестов с атрибутами Fact и Theory	999
35.6 Тестирование условий сбоя	1003
Резюме	1004

36 Тестирование приложений ASP.NET Core 1006

36.1 Модульное тестирование пользовательского промежуточного ПО.....	1007
36.2 Модульное тестирование контроллеров API и конечных точек минимальных API.....	1010
36.3 Интеграционное тестирование: тестирование всего приложения в памяти.....	1015
36.3.1 Создание TestServer с помощью пакета Test Host	1016
36.3.2 Тестирование приложения с помощью WebApplicationFactory	1019
36.3.3 Замена зависимостей в классе WebApplicationFactory	1022
36.3.4 Уменьшение дублирования кода за счет создания своего класса WebApplicationFactory	1024
36.4 Изоляция базы данных с помощью поставщика EF Core в памяти....	1026
Резюме	1031

Приложение А. Подготовка окружения разработки 1033

A.1 Установка .NET SDK	1034
A.2 Выбор интегрированной среды разработки или редактора кода	1035
A.2.1 Visual Studio (Windows).....	1036
A.2.2 Rider от компании JetBrains (Windows, Linux, macOS)	1037
A.2.3 Visual Studio для Mac (macOS)	1038
A.2.4 Visual Studio Code (Windows, Linux, macOS)	1038

<i>Приложение Б. Полезные ссылки</i>	1040
Б.1 Список литературы	1040
Б.2 Анонсы в блогах	1041
Б.3 Документация Microsoft	1042
Б.4 Ссылки по темам, связанным с безопасностью	1042
Б.5 Репозитории ASP.NET Core на GitHub.....	1043
Б.6 Инструменты и сервисы	1043
Б.7 Блоги ASP.NET Core.....	1043
Б.8 Ссылки на видео.....	1044

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в изда-

тельство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступительное слово от сообщества DotNet.Ru

.NET уже много лет является одним из лидирующих фреймворков для разработки веб-приложений. Пройдя длинный путь от ASP.NET до современного ASP.NET Core, он вобрал в себя все лучшие подходы к разработке приложений с отрисовкой на стороне сервера и веб-API-приложений. ASP.NET Core – продукт с открытым исходным кодом, а значит, каждый может изучить любой аспект его работы. Однако объем кода велик, и не так просто сразу понять, что именно искать и как с этим разбираться. Microsoft предоставляет отличную документацию по ASP.NET Core и основам серверной веб-разработки на официальном сайте, но этого может быть недостаточно для выстраивания целостной картины.

Много полезной информации можно почерпнуть из книг, и по ASP.NET Core их написано уже немало. Одной из таких книг стала «ASP.NET Core в действии» (2-е издание) от Эндрю Лока, вышедшая несколько лет назад. Но .NET не стоит на месте, появляются новые концепции, изменяются распространенные сценарии использования, и чтобы быть в курсе современных стилей и подходов к программированию на .NET с использованием ASP.NET Core, мы представляем вам третье издание этой прекрасной книги.

Автор превосходно знает ASP.NET Core, работал с ним с первых версий и как никто другой понимает, какие аспекты фреймворка наиболее важны для его успешного использования. Разработчику, помимо работы с основной логикой приложения, важно понимать, как работать с настройками, журналированием, авторизацией, как обеспечивать безопасность приложений. Все эти темы тщательно рассмотрены в книге. Третье издание пополнилось новыми главами про минимальные API, а также существенно расширился материал о безопасной разработке. Автору удалось охватить широту фреймворка, рассмотрев большое количество различных аспектов и при этом достаточно глубоко разобрав многие из них. Все это позволяет рассматривать эту книгу как отличный способ подробного знакомства с разработкой серверных приложений на .NET.

Систематизированной информации об ASP.NET Core на русском языке мало. Фреймворк стремительно развивается, постоянно появляются новые термины, и даже те, что давно используются, не всегда имеют устоявшийся в профессиональной среде перевод. Мы обсуждали, спорили, думали о том, как читатели будут искать термины в сети

Интернет, как они звучат в неформальных беседах. Что-то получилось хорошо, что-то можно улучшить, но в целом мы довольны результатом и рады, что такая интересная и полезная книга доступна теперь и на русском языке. Отдельная благодарность автору за простые и понятные примеры кода и отличные иллюстрации, наглядно демонстрирующие объясняемые концепции.

Добро пожаловать в мир ASP.NET Core и приятного чтения!

Российское сообщество .NET-разработчиков DotNet.Ru

Над переводом работали представители сообщества DotNet.Ru:

- Игорь Лабутин;
- Дмитрий Павлов;
- Рустам Сафин;
- Максим Молоканов;
- Сергей Бензенко;
- Радмир Тагиров;
- Андрей Брижак;
- Евгений Асташев;
- Кирилл Вишневский;
- Азамат Сулейманов;
- Дмитрий Кунченко;
- Владимир Майоров;
- Алексей Ростов;
- Андрей Беленцов;
- Максим Шошин;
- Вадим Мингажев;
- Эмиль Янгиров;
- Анатолий Кулаков.

Предисловие

У ASP.NET долгая история; Microsoft выпустила первую версию ASP.NET в 2002 году как часть исходной платформы .NET Framework 1.0. С тех пор она прошла несколько выпусков, в каждом из которых были добавлены новые функции и расширяемость. Однако каждый выпуск был построен на основе .NET Framework, поэтому она предустановлена во всех версиях Windows.

Это дает неоднозначные преимущества: с одной стороны, сегодня ASP.NET 4.x является надежной, проверенной в боях платформой для создания современных приложений для ОС Windows. С другой – она ограничена этой зависимостью: изменения в базовой платформе .NET Framework имеют далеко идущие последствия, в результате чего наблюдается замедление скорости распространения, а это оставляет за бортом многих разработчиков, создающих и развертывающих приложения для Linux или macOS.

Когда я впервые начал изучать ASP.NET Core, я был одним из таких разработчиков. Будучи в душе пользователем Windows, я получил от своего работодателя компьютер с Mac и поэтому весь день работал на виртуальной машине. ASP.NET Core обещал все это изменить, позволив вести разработку и на компьютере с Windows, и на компьютере с Mac.

Можно сказать, что я опоздал во многих отношениях, проявляя активный интерес только перед выходом релиз-кандидата ASP.NET Core RC2. К тому моменту существовало уже восемь (!) бета-версий, многие из которых содержали существенные критические изменения. Не погружаясь во все это полностью до выхода RC2, я избежал боли от хитроумных инструментов и изменений API.

То, что я увидел в тот момент, меня очень впечатлило. ASP.NET Core позволяет разработчикам использовать имеющиеся у них знания о платформе .NET и приложениях ASP.NET MVC в частности, применяя текущие передовые практики, такие как внедрение зависимостей, строго типизированная конфигурация и журналирование. Кроме того, можно было создавать и развертывать кросс-платформенные приложения. Я не устоял.

Эта книга появилась во многом благодаря моему подходу к изучению ASP.NET Core. Вместо того чтобы просто читать документацию и посты в блогах, я решил попробовать что-то новое и начать писать о том, что я узнал. Каждую неделю я посвящал время изучению нового аспекта ASP.NET Core и писал об этом сообщение в блоге. Когда появилась возможность написать книгу, я ухватился за этот шанс – это еще один повод подробно изучить данный фреймворк!

С тех пор, как я начал писать эту книгу, многое изменилось в отношении как самой книги, так и ASP.NET Core. Первый крупный выпуск фреймворка в июне 2016 года по-прежнему имел много шероховатостей, в частности что касалось работы с инструментами. С выпуском .NET 7 в ноябре 2022 года ASP.NET Core действительно стал самостоятельным: API и инструменты достигли зрелого уровня.

Обновления фреймворка в .NET 6 и .NET 7, в которых благодаря введению минимального хостинга и минимального API обеспечивается более краткий и простой подход к написанию API, что намного ближе к опыту других языков, значительно упростили процесс начала работы для новичков. Вы можете сразу приступать к созданию функциональности своего приложения без необходимости сначала разбираться в архитектуре.

Некоторым опытным разработчикам ASP.NET Core эти изменения могут показаться отбрасывающими назад и неструктуризованными. Если вы являетесь таковым, призываю вас дать им шанс и начать построение собственной структуры и паттернов. Чтобы примеры в этой книге были краткими и ясными, я часто помещаю весь код приложения в один файл, но не думайте, что именно так нужно писать настоящие приложения. Вы можете создавать вспомогательные методы, классы и любую структуру, которая помогает поддерживать удобство обслуживания ваших приложений, одновременно используя преимущества производительности минимальных API.

В этой книге рассказывается обо всем, что вам нужно для начала работы с ASP.NET Core, независимо от того, новичок ли вы в веб-разработке или уже являетесь разработчиком ASP.NET. В ней очень много внимания уделяется самому фреймворку, поэтому я не буду вдаваться в подробности, касающиеся клиентских фреймворков, таких как Angular и React, или таких технологий, как Docker. Я также не описываю все новые функции .NET 7, такие как Blazor и gRPC. Вместо этого я даю ссылки, по которым вы можете найти дополнительную информацию.

В этом издании я значительно расширил и переставил многие главы по сравнению с предыдущими изданиями книги; некоторые главы были разделены на более удобные размеры. В первых главах содержится много новой информации, посвященной минимальным API и минимальному хостингу, представленным в .NET 6.

Если сравнивать с приложениями, использующими предыдущую версию ASP.NET, то лично мне приятно работать с приложениями ASP.NET Core, и надеюсь, что эта страсть проявится в данной книге!

Благодарности

Хотя на обложке этой книги только одно имя, множество людей внесли свой вклад как в ее написание, так и в публикацию. В этом разделе я хотел бы поблагодарить всех, кто поддерживал меня, оказывал содействие и терпел меня в течение прошлого года.

Прежде всего я хочу поблагодарить свою девушку Бекки. Твоя постоянная поддержка и воодушевление – все для меня. Они помогли мне пережить этот напряженный период. Ты приняла на себя всю тяжесть этого стресса и давления, и я бесконечно благодарен тебе. Безмерно люблю тебя.

Я также хотел бы поблагодарить всю свою семью за их поддержку. В частности, моих родителей, Джен и Боба, за то, что терпели мои разглагольствования, и свою сестру, Аманду, за все ее веселые беседы.

На профессиональном уровне я хотел бы поблагодарить издательство Manning за предоставленную мне возможность. Брайан Сойер «нашел» меня во время работы с первым изданием этой книги и побудил меня заняться вторым изданием. Марина Майклс стала моим редактором-консультантом по аудитории второй раз подряд и снова была то дотошной, то критически настроенной, то обнадеживающей и восторженной. Книга, несомненно, стала лучше благодаря вашему участию.

Спасибо моему редактору Адриане Сабо и всем рецензентам: Алексею Аданичу, Бену Макнамаре, Беле Исток, Даррину Бишопу, Деннису Лиабенову, Элу Пежевски, Эммануилу Чардаласу, Фостеру Хейнсу, Онофрею Джорджу, Джону Гатри, Жану-Франсуа Морену, Петро Сероменью, Джо Куэвасу, Хосе Антонио Мартинесу Пересу, Джо Суши, Луису Му, Милану Шаренцу, Милораду Имбра, Нику Римингтону, Нитину Айнани, Оливеур Кортену, Раушану Джа, Ричарду Янгу, Рику Бирендонку, Рону Лизу, Рубену Вандегинсте, Сумиту К. Сингху, Тоухидулу Башару, Дэниелу Вассесу и Уиллу Лопесу. Ваши предложения помогли сделать эту книгу лучше.

Выражая благодарность техническому редактору этой книги Филиппу Войцешину, который является основателем и сопровождающим нескольких популярных проектов с открытым исходным кодом, частым докладчиком на конференциях и обладателем награды Microsoft MVP. Филипп предоставил неоценимую помощь, подчеркнув мои неверные предположения и технические предубеждения, также удостоверившись, что все написанное мною является верным с технической точки зрения.

Я благодарен еще техническому корректору Тане Уилке. Потрясающие плодотворно работая с главами, она удостоверилась, что написанный мной код действительно работает и что он не лишен смысла.

От всего сердца благодарю всех сотрудников Manning, которые помогли издать эту книгу и вывести ее на рынок. Я также хотел бы поблагодарить всех рецензентов MEAP за их комментарии, которые помогли улучшить книгу.

Я бы никогда не смог написать ее, если бы не отличный контент, созданный сообществом .NET и теми пользователями, на которых я подписан в социальных сетях.

Наконец, спасибо всем друзьям, которые воодушевляли и поддерживали меня, и в целом проявляли интерес. Возможно, нам не удавалось встречаться настолько часто, насколько нам хотелось бы, но я с нетерпением жду возможности как можно скорее собраться вместе и выпить.

Об этой книге

Данная книга посвящена фреймворку ASP.NET Core: в ней рассказывается о том, что это такое и как использовать его для создания веб-приложений. Хотя часть этой информации уже доступна в Интернете, она разбросана по Сети в виде разрозненных документов и сообщений в блогах. Эта книга показывает, как создать свое первое приложение, наращивая сложность по мере того, как вы будете закреплять предыдущие концепции.

Я представляю каждую тему на относительно небольших примерах, вместо того чтобы создавать одно-единственное приложение на протяжении всей книги. У обоих подходов есть свои достоинства, но я хотел убедиться, что основное внимание уделяется конкретным изучаемым темам, без умственных затрат на навигацию по растущему проекту.

К концу книги вы должны иметь твердое представление о том, как создавать приложения с помощью ASP.NET Core, знать сильные и слабые стороны этого фреймворка и то, как использовать его функциональные возможности для безопасного создания приложений. Хотя я не трачу много времени на архитектуру приложений, я непременно привожу передовые практики, особенно там, где лишь поверхностно рассказываю об архитектуре для краткости.

Кому адресована эта книга

Данная книга ориентирована на C#-разработчиков, которые заинтересованы в изучении кросс-платформенного веб-фреймворка. Она не предполагает, что у вас есть какой-либо опыт создания веб-приложений, например вы можете разрабатывать приложения для мобильных устройств или ПК, хотя предыдущий опыт работы с ASP.NET или другим веб-фреймворком, несомненно, полезен.

Помимо практических знаний C# и .NET, предполагается наличие знания общих объектно ориентированных практик и базового понимания реляционных баз данных. Я предполагаю, что вы немного знакомы с HTML и CSS, а также с JavaScript в роли языка сценариев на стороне клиента. Вам не нужно знать JavaScript- или CSS-фреймворки для работы с этой книгой, хотя ASP.NET Core хорошо работает с ними, если это ваша сильная сторона.

Веб-фреймворки естественным образом затрагивают широкий круг тем, начиная с базы данных и работы с сетью и заканчивая визуальным дизайном и написанием скриптов на стороне клиента. Я предоставляю как можно больше контекста и включаю ссылки на сайты и книги, где можно получить более подробную информацию.

Как устроена эта книга: дорожная карта

Эта книга состоит из пяти частей, 36 глав и двух приложений. В идеале нужно прочитать ее от корки до корки, а затем использовать ее в качестве справочника, но я понимаю, что такой вариант подойдет не всем. Хотя я применяю небольшие примеры приложений для демонстрации той или иной темы, некоторые главы основаны на предыдущих, поэтому содержание книги будет иметь больше смысла, если вы будете читать главы последовательно.

Я настоятельно рекомендую читать главы первой части последовательно, поскольку каждая глава основывается на темах, представленных в предыдущих главах. Вторую часть также лучше читать последовательно, но большинство глав не зависят друг от друга, и если вы хотите, то можете перескакивать от одной к другой. Третью часть, опять же, лучше читать последовательно. Вы получите лучший опыт, прочитав главы четвертой и пятой частей последовательно, но многие темы независимы, поэтому вы можете читать их не по порядку, если хотите. Но я рекомендую делать это только после того, как вы прошли части с первой по третью.

Первая часть представляет собой общее введение в ASP.NET Core, уделяя особое внимание созданию небольших API для JSON с использованием новейших функциональных возможностей, представленных в .NET 7. После того как мы рассмотрим основы, мы изучим создание приложений с минимальными API, которые предоставляют простейшую модель программирования для веб-приложений ASP.NET Core:

- глава 1 знакомит вас с ASP.NET Core и его местом в среде веб-разработки. В ней описаны типы приложений, которые вы можете создавать, причины выбора ASP.NET Core, а также основы веб-запросов в приложении ASP.NET Core;
- в главе 2 рассказывается, почему следует рассмотреть возможность использования веб-фреймворка, почему был создан ASP.NET Core и различные парадигмы приложений, которые можно использовать с ASP.NET Core. Наконец, рассматриваются ситуации, когда следует или не следует выбирать ASP.NET Core;
- в главе 3 рассматриваются все компоненты базового минимального API, обсуждается их роль и как они объединяются для генерирования ответа на веб-запрос;
- в главе 4 описан конвейер промежуточного программного обеспечения – основной конвейер приложений в ASP.NET Core, который определяет, как обрабатываются входящие запросы и как должен генерироваться ответ;
- в главе 5 показано, как использовать конечные точки минимальных API для создания HTTP JSON API, который может вызываться клиентскими и серверными приложениями или мобильными устройствами;
- в главе 6 описывается система маршрутизации ASP.NET Core. Маршрутизация – это процесс сопоставления URL-адресов вхо-

дящих запросов с определенным методом обработчика, который выполняется для генерирования ответа;

- в главе 7 рассматривается привязка модели в минимальных API, процесс сопоставления данных формы и параметров URL, передаваемых в запросе, с конкретными объектами C#.

Вторая часть охватывает важные темы для создания полнофункциональных веб-приложений, после того как вы разберетесь с основами:

- в главе 8 представлена концепция внедрения зависимостей (DI) и описан контейнер внедрения зависимостей, встроенный в ASP.NET Core;
- глава 9 является продолжением главы 8 и описывает, как регистрировать собственные сервисы в DI-контейнере, какие шаблоны можно использовать и что такое жизненный цикл сервисов, создаваемых этим контейнером;
- в главе 10 обсуждается, как читать настройки и секреты в ASP.NET Core и как сопоставлять их со строго типизированными объектами;
- в главе 11 описывается, как документировать API с использованием стандарта OpenAPI и как это помогает в сценариях тестирования и автоматическом создании клиентов для вызова API;
- глава 12 знакомит с библиотекой Entity Framework Core, которая используется для сохранения данных в реляционной базе данных.

В третьей части рассматривается, как создавать HTML-приложения на основе страниц с отрисовкой на стороне сервера, используя Razor Pages и архитектуру «модель–представление–контроллер» (MVC):

- в главе 13 показано, как использовать Razor Pages для создания многостраничных веб-сайтов. Razor Pages – это рекомендуемый способ создания приложений с отрисовкой на стороне сервера в ASP.NET Core, они предназначены для многостраничных приложений;
- в главе 14 описываются система маршрутизации Razor Pages и ее отличие от минимальных API;
- в главе 15 рассматриваются обработчики страниц в Razor Pages, которые отвечают за выбор способа ответа на запрос и выбор того, какой ответ генерировать;
- в главе 16 рассматривается привязка модели в Razor Pages, ее отличие от минимальных API и важность валидации моделей;
- в главе 17 показано, как создавать HTML-страницы с помощью языка шаблонов Razor;
- глава 18 основана на главе 17 и представляет тег-хелперы, которые могут значительно сократить объем кода, необходимого для создания форм и веб-страниц;
- в главе 19 представлены контроллеры MVC как альтернативный подход к созданию как HTML-приложений с отрисовкой на стороне сервера, так и приложений API;
- в главе 20 описывается, как использовать контроллеры MVC для создания API, которые могут вызываться клиентскими приложениями в качестве альтернативы минимальным API;

- в главе 21 представлен конвейер фильтров Razor Pages и MVC, показано, как он работает, и описан ряд фильтров, встроенных во фреймворк;
- глава 22 развивает главу 21 и показывает, как создавать собственные фильтры, чтобы уменьшить дублирование кода в приложениях MVC и Razor Pages.

Главы, из которых состоит четвертая часть, охватывают важные сквозные аспекты разработки ASP.NET Core:

- в главе 23 описывается, как добавить профили пользователей и аутентификацию в приложение с помощью ASP.NET Core Identity;
- глава 24 развивает предыдущую главу и рассказывает об авторизации пользователей, чтобы вы могли ограничить доступ к страницам, к которым может получить доступ выполнивший вход пользователь;
- в главе 25 обсуждаются аутентификация и авторизация для приложений API, их отличие от аутентификации в HTML-приложениях и начало работы с аутентификацией в API ASP.NET Core;
- в главе 26 показано, как настроить журналирование в приложении и как записывать сообщения журнала в разные хранилища;
- в главе 27 рассказывается, как опубликовать приложение и настроить его для промышленного окружения;
- в главе 28 обсуждается причина добавления протокола HTTPS в приложение, как использовать его при локальной разработке и в промышленном окружении, а также как принудительно использовать HTTPS для всего приложения;
- в главе 29 рассматриваются другие вопросы безопасности, которые следует учитывать при разработке приложения, а также способы обеспечения безопасности с помощью ASP.NET Core.

В пятой части рассматриваются различные темы, которые помогут вам совершенствовать свои приложения ASP.NET Core, включая приложения, работающие за пределами браузера, собственную конфигурацию и компоненты, а также тестирование:

- в главе 30 обсуждается альтернативный подход к начальной загрузке для приложений ASP.NET Core с использованием универсального узла и класса `Startup`;
- в главе 31 описывается, как создавать и использовать различные пользовательские компоненты, например специальное промежуточное программное обеспечение, а также как выполнять сложные требования к конфигурации;
- глава 32 расширяет главу 31 и показывает, как создавать пользовательские компоненты Razor Page, например собственные тег-хелперы и атрибуты валидации;
- в главе 33 обсуждаются служба `IHttpClientFactory` и способы ее использования для создания экземпляров `HttpClient` для вызова удаленных API;

- в главе 34 рассматривается общая абстракция `IHost`, которую можно использовать для создания служб Windows и демонов Linux. Вы также научитесь запускать фоновые задачи в своих приложениях;
- в главе 35 показано, как протестировать приложение ASP.NET Core с помощью фреймворка `xUnit`;
- глава 36 продолжает главу 35 и показывает, как именно тестируются приложения ASP.NET Core. В ней описываются модульные и интеграционные тесты с использованием `Test Host`.

В двух приложениях представлена дополнительная информация:

- в приложении А описано, как настроить среду разработки независимо от того, используете ли вы OS Windows, Linux или macOS;
- приложение В содержит ссылки, которые я нашел полезными для изучения ASP.NET Core.

Соглашения об оформлении программного кода

Исходный код предоставляется для всех глав, кроме глав 1, 2, 21 и 27, в которых нет кода. Код каждой главы можно посмотреть в моем репозитории GitHub на странице <https://github.com/andrewlock/asp-dot-net-core-in-action-3e>. ZIP-файл, содержащий весь исходный код, также доступен на сайте издателя на странице <https://www.manning.com/books/asp-net-core-in-action-Third-edition>. Исполняемые фрагменты кода можно получить из онлайн-версии этой книги для liveBook на странице <https://livebook.manning.com/book/asp-dot-net-core-in-action-Third-edition>.

Во всех примерах кода в этой книге используется .NET 7. Они создаются с применением Visual Studio и Visual Studio Code. Для сборки и запуска примеров необходимо установить пакет разработки программного обеспечения (SDK) .NET, как описано в приложении А.

Данная книга содержит множество примеров исходного кода как в пронумерованных листингах, так и в обычном тексте. В обоих случаях исходный код отформатирован шрифтом фиксированной ширины, подобным этому, чтобы отделить его от обычного текста. Иногда также используется жирный шрифт, чтобы выделить код, который изменился по сравнению с предыдущими шагами, например когда в существующую строку кода добавляется новая функция.

Во многих случаях оригинальный исходный код был переформатирован; мы добавили разрывы строк и переработали отступы, чтобы подогнать доступное пространство для страниц в книге. Кроме того, комментарии в исходном коде часто удаляются из листингов, если описание кода приводится в тексте. Многие листинги сопровождаются аннотациями к коду, выделяя важные концепции.

Об авторе



Эндрю Лок – разработчик .NET и обладатель награды MVP Microsoft. Он окончил Кембриджский университет со степенью инженера по специальности «Разработка программного обеспечения» и получил докторскую степень в области обработки цифровых изображений. Он профессионально занимается разработкой с применением .NET с 2010 года, используя широкий спектр технологий, включая Win-Forms, ASP.NET WebForms, ASP.NET MVC, веб-страницы ASP.NET и, совсем недавно, ASP.NET Core. Эндрю запустил в производство множество приложений ASP.NET Core с момента выпуска версии 1 в 2016 году. Он ведет блог на странице <https://andrewlock.net>, посвященный ASP.NET Core. Этот блог часто упоминался в центре внимания сообщества командой ASP.NET в Microsoft, в блоге .NET и на еженедельных выступлениях перед сообществом.

Об иллюстрации на обложке

Подпись к иллюстрации на обложке книги гласит: «Капитан Паша. Капудан-паша, адмирал турецкого флота». Иллюстрация взята из коллекции костюмов Османской империи, опубликованной 1 января 1802 года Уильямом Миллером.

В те времена по одежде было легко определить, где живут люди и чем они занимаются или их положение в обществе. Издательство Manning ценит изобретательность и инициативность компьютерного бизнеса с помощью обложек книг, основанных на богатом разнообразии региональной культуры много веков назад, которое оживает благодаря картинкам из этой коллекции.

1

Начало работы с ASP.NET Core

В этой главе:

- что такое ASP.NET Core;
- что можно создать с помощью ASP.NET Core;
- как работает ASP.NET Core.

Решение изучать новый фреймворк и использовать его для разработки – это серьезные инвестиции, поэтому важно заранее определить, подходит ли он вам. В этой главе рассказывается об ASP.NET Core: что это такое, как он работает и почему следует использовать его для создания своих веб-приложений.

К концу данной главы вы должны вполне четко представлять себе, что такое .NET, какова роль .NET 7, и знать основные механизмы работы ASP.NET Core. Итак, не теряя времени, приступим!

1.1 Что такое ASP.NET Core?

ASP.NET Core – это кроссплатформенный фреймворк с открытым исходным кодом, который можно использовать для быстрого создания динамических веб-приложений. Вы можете использовать ASP.NET Core для создания веб-приложений с отрисовкой на стороне сервера,

серверных приложений, API для протокола HTTP, которые могут применяться мобильными приложениями, и многое другое. ASP.NET Core работает на .NET 7. Это последняя версия .NET Core – высокопроизводительной кросс-платформенной среды выполнения с открытым исходным кодом.

ASP.NET Core предоставляет структуру, вспомогательные функции и фреймворк для создания приложений, что избавляет вас от необходимости писать большую часть этого кода самостоятельно. Затем код фреймворка ASP.NET Core вызывает «обработчики», которые, в свою очередь, вызывают методы бизнес-логики приложения, как показано на рис. 1.1. Эта бизнес-логика является ядром вашего приложения. Здесь вы можете взаимодействовать с другими службами, такими как базы данных или удаленные API, но обычно бизнес-логика не зависит *напрямую* от ASP.NET Core.

Приложения ASP.NET Core могут обслуживать клиентов на основе браузера или могут предоставлять API-интерфейсы для мобильных и других клиентов

Код фреймворка ASP.NET обрабатывает низкоуровневые запросы и вызывает «обработчики» контроллеров Razor Pages и веб-API

Вы пишете эти обработчики, используя примитивы, предоставляемые фреймворком. Обычно они вызывают методы в логике предметной области

Предметная область может использовать внешние сервисы и базы данных для выполнения своих функций и сохранения данных

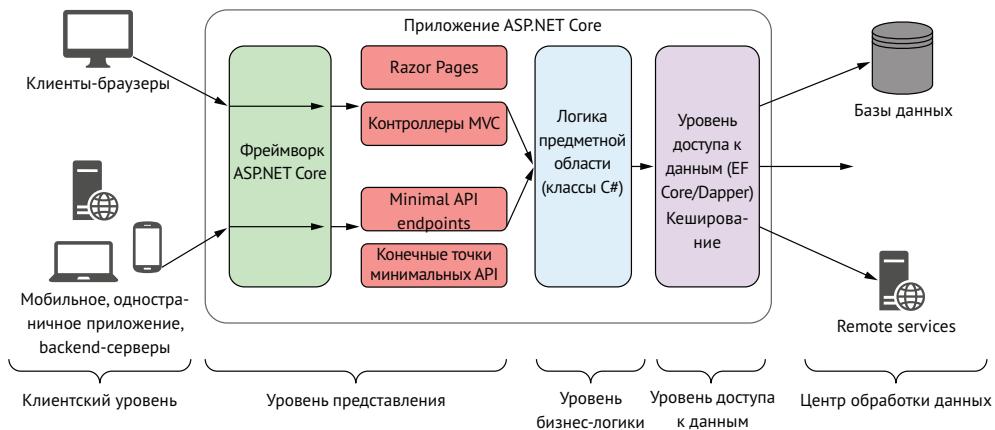


Рис. 1.1 Типичное приложение ASP.NET Core состоит из нескольких уровней. Код фреймворка ASP.NET Core обрабатывает запросы от клиента, работая со сложным сетевым кодом. Затем фреймворк вызывает обработчики (Razor Pages и контроллеры веб-API), которые пишутся с использованием примитивов, предоставляемых фреймворком. В конце эти обработчики вызывают логику предметной области приложения, которая обычно представляет собой классы и объекты C# без каких-либо зависимостей, специфичных для ASP.NET Core

1.2 Какие типы приложений можно создавать?

ASP.NET Core – это универсальный веб-фреймворк, который можно использовать для создания самых разных приложений. Он включает в себя API, поддерживающие множество парадигм:

- **минимальные API** – простые API для протокола HTTP, которые могут использоваться мобильными приложениями или одностраничными браузерными приложениями;

- *веб-API* – альтернативный подход к созданию API для протокола HTTP, который добавляет больше структуры и функциональных возможностей по сравнению с минимальными API;
- *gRPC API* – используются для создания эффективных двоичных API для обмена данными между серверами с использованием протокола gRPC;
- *Razor Pages* – используется для создания многостраничных приложений с отрисовкой на стороне сервера;
- *контроллеры MVC* – аналогичны Razor Pages и предназначены для серверных приложений, но без страничной парадигмы;
- *Blazor WebAssembly* – фреймворк для создания одностраничных приложений на основе браузера, использующий стандарт WebAssembly. Это аналог таких JavaScript-фреймворков, как Angular, React и Vue;
- *Blazor Server* – используется для создания приложений с отслеживанием состояния и отрисовкой на стороне сервера, которые отправляют события пользовательского интерфейса и обновления страниц через WebSockets, чтобы обеспечить ощущение одностораничного приложения на стороне клиента, но с легкостью разработки приложения с отрисовкой на стороне сервера.

Все эти парадигмы основаны на одних и тех же строительных блоках ASP.NET Core, таких как библиотеки конфигурирования и журналирования с добавлением дополнительных функциональных возможностей. Лучший выбор для вашего приложения зависит от множества факторов, включая требования к API, детали существующих приложений, с которыми необходимо взаимодействовать, детали браузеров и эксплуатационного окружения ваших клиентов, а также масштабируемость и требования к бесперебойной работе. Не обязательно выбирать только одну из них; ASP.NET Core может сочетать несколько парадигм в одном приложении.

1.3 Выбор ASP.NET Core

Надеюсь, теперь у вас есть общее представление о том, что такое ASP.NET Core и какие типы приложений можно создавать с его помощью. Но остается один вопрос: стоит ли его использовать? Microsoft рекомендует, чтобы все новые веб-разработки .NET использовали ASP.NET Core, но переход на новый веб-стек или его изучение – серьезная задача для любого разработчика или компании.

Если вы только начинаете свой путь в .NET-разработке и рассматриваете возможность использования ASP.NET Core, тогда добро пожаловать! Microsoft продвигает ASP.NET Core как привлекательный вариант для новичков в веб-разработке, но кросс-платформенность .NET означает, что он конкурирует со многими другими платформами на их собственной территории. У ASP.NET Core есть множество преимуществ по сравнению с другими кросс-платформенными фреймворками для создания веб-приложений:

- это современный высокопроизводительный веб-фреймворк с открытым исходным кодом;

- в нем используются знакомые паттерны проектирования и парадигмы;
- C# – отличный язык (но при желании можно использовать VB.NET или F#);
- вы можете выполнять сборку и запуск на любой платформе.

ASP.NET Core – это переосмысление фреймворка ASP.NET, созданное с использованием современных принципов проектирования программного обеспечения на основе новой платформы .NET. Несмотря на то что .NET (которая ранее называлась *.NET Core*) в каком-то смысле новая, она широко используется в промышленных целях с 2016 года и в значительной степени опирается на зрелый, стабильный и надежный фреймворк .NET, который используется уже более двух десятилетий. Вы можете быть спокойны, зная, что, выбрав ASP.NET Core и .NET 7, вы получаете надежную платформу, а также полнофункциональный фреймворк для создания веб-приложений.

Одним из основных преимуществ ASP.NET Core и .NET 7 является возможность разработки и запуска на любой платформе. Независимо от того, используете ли вы компьютер с Mac, Windows или Linux, вы можете запускать одни и те же приложения ASP.NET Core и разрабатывать их в нескольких окружениях. Для пользователей Linux существует поддержка широкого спектра дистрибутивов: RHEL, Ubuntu, Debian, CentOS, Fedora и openSUSE, и это лишь некоторые из них. ASP.NET Core даже работает на крошечном дистрибутиве Alpine, что обеспечивает по-настоящему компактное развертывание в контейнерах, поэтому вы можете быть уверены, что выбранная вами операционная система будет подходящим вариантом.

Если вы уже являетесь .NET-разработчиком, то выбор того, инвестировать ли в ASP.NET Core при разработке новых приложений, во многом зависел от времени. В ранних версиях .NET Core отсутствовали некоторые функциональные возможности, которые затрудняли внедрение, но в последних версиях .NET этой проблемы больше нет. Теперь Microsoft напрямую рекомендует, чтобы все новые приложения .NET использовали .NET 7 (или более новую версию).

Microsoft пообещала предоставлять исправления ошибок и безопасности для старой платформы ASP.NET, но больше не будет предоставлять никаких обновлений функциональных возможностей. .NET Framework никуда не уходит, поэтому ваши старые приложения продолжат работать, но его не следует использовать для разработки новых.

Основные преимущества ASP.NET Core по сравнению с предыдущей платформой ASP.NET:

- кроссплатформенная разработка и внедрение;
- фокус на производительности как на функциональной особенности;
- упрощенная модель размещения;
- регулярные выпуски с более коротким циклом;
- открытый исходный код;
- модульный подход;

- дополнительные параметры парадигмы приложения;
- возможность упаковывать .NET-приложение при публикации для автономного развертывания.

Будучи разработчиком .NET, который переходит на ASP.NET Core, ваша способность создавать и развертывать кросс-платформенные приложения открывает двери для совершенно новых возможностей приложений, среди которых – использование преимуществ более дешевого хостинга с виртуальными машинами Linux в облаке, использование контейнеров Docker для повторяемой непрерывной интеграции или написание кода .NET на компьютере с Mac без необходимости запуска виртуальной машины Windows. Все это становится возможным благодаря ASP.NET Core в сочетании с .NET 7.

Это не значит, что ваш опыт развертывания приложений ASP.NET в Windows и Internet Information Services (IIS) не пригодится. Напротив, ASP.NET Core использует многие из тех же концепций, что и предыдущая платформа ASP.NET, и вы по-прежнему можете запускать приложения ASP.NET Core в IIS, поэтому переход на ASP.NET Core не означает, что нужно все начинать с нуля.

1.4 Как работает ASP.NET Core?

Я вкратце поведал вам, что такое ASP.NET Core, для чего следует его использовать и почему следует рассмотреть такой вариант. В этом разделе вы увидите, как работает приложение, созданное с помощью ASP.NET Core, от запроса пользователем URL-адреса до страницы, отображаемой в браузере. Сначала вы увидите, как веб-сервер обрабатывает HTTP-запрос, а затем мы покажем, как ASP.NET Core расширяет этот процесс для создания динамических веб-страниц.

1.4.1 Как работает веб-запрос по протоколу HTTP?

Как вы знаете, ASP.NET Core – это фреймворк для создания веб-приложений, которые обслуживают данные с сервера. Один из наиболее распространенных сценариев для веб-разработчиков – создание веб-приложения, которое можно просматривать в браузере. На рис. 1.2 показан процесс обработки запроса страницы любым веб-сервером.

Процесс начинается, когда пользователь переходит на сайт или вводит URL-адрес в своем браузере. URL-адрес или веб-адрес состоит из имени хоста и пути к некоему ресурсу в веб-приложении. При переходе по адресу в браузере запрос с компьютера пользователя отправляется на сервер, на котором размещено веб-приложение, с использованием протокола HTTP.

ОПРЕДЕЛЕНИЕ Имя хоста веб-сайта однозначно определяет его местоположение в интернете путем сопоставления с IP-адресом с помощью системы доменных имен (DNS). В качестве примеров можно упомянуть сайты microsoft.com, www.google.co.uk и facebook.com.

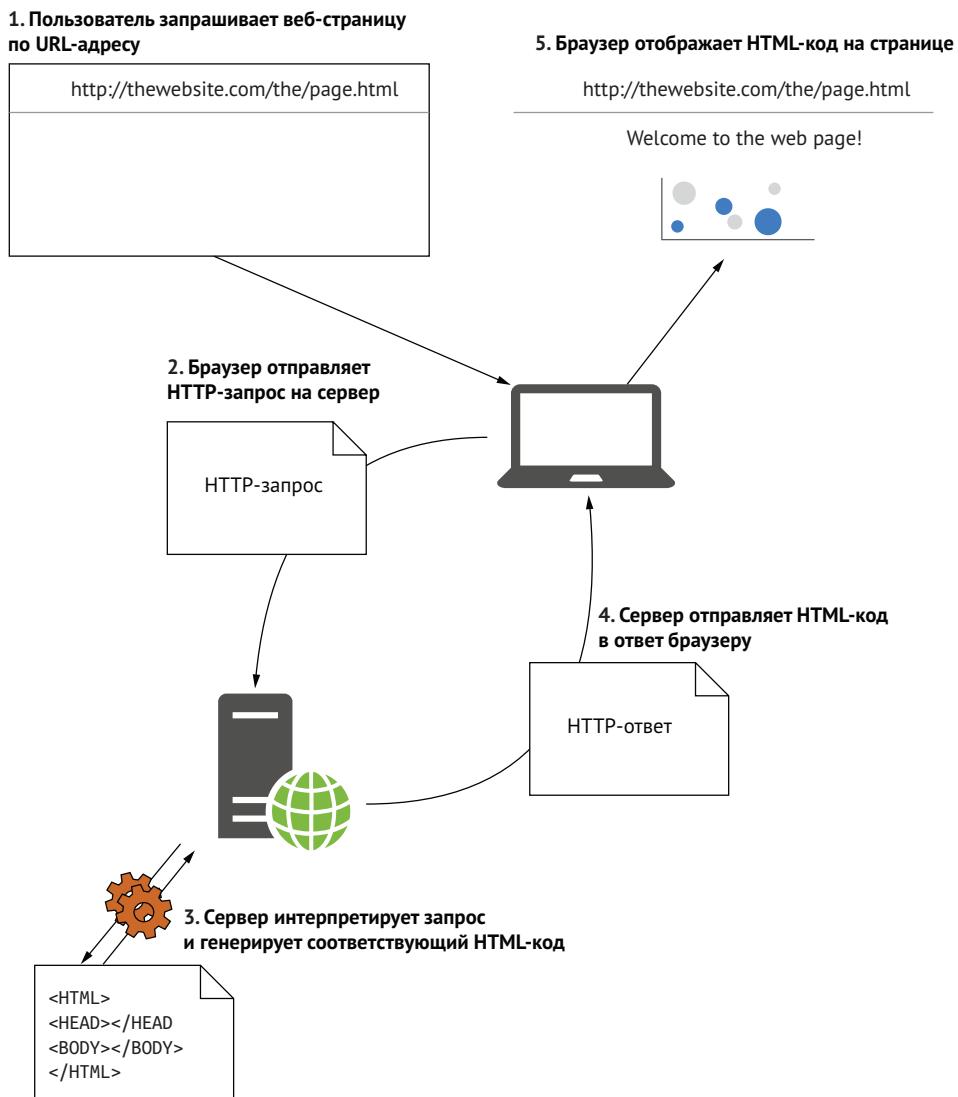


Рис. 1.2 Запрос веб-страницы. Пользователь начинает с запроса веб-страницы, что вызывает отправку HTTP-запроса на сервер. Сервер интерпретирует запрос, генерирует необходимый HTML и отправляет его обратно в HTTP-ответе. Затем браузер может отобразить веб-страницу

Краткое руководство по HTTP

Протокол передачи гипертекста (HTTP) – это протокол прикладного уровня, обеспечивающий работу в сети. Это протокол запроса–ответа без сохранения состояния, работающий по схеме «запрос–ответ». Клиентский компьютер отправляет запрос на сервер, который, в свою очередь, передает ответ.

Каждый HTTP-запрос состоит из глагола, указывающего «тип» запроса, и пути, указывающего ресурс, с которым нужно взаимодействовать. Обычно они также содержат заголовки, представляющие собой пары «ключ–значение», а в некоторых случаях тело, например содержимое формы, при отправке данных на сервер.

Ответ содержит код состояния, указывающий на то, был ли запрос успешным, а также (необязательно) заголовки и тело.

Для получения более подробной информации о самом протоколе HTTP, а также чтобы увидеть другие примеры, см. раздел 1.3 («Краткое введение в HTTP») в книге Go Web Programming Cay Шонга Чанга (Manning 2016 г., <https://livebook.manning.com/book/go-web-programming/chapter-1/point-9018-55-145-1>). Вы также можете посетить страницу <https://www.rfc-editor.org/rfc/rfc9110.txt>, если предпочтаете читать содержимое файлов в формате .txt!

Запрос проходит через интернет, возможно, на другую сторону света, пока, наконец, не попадет на сервер, связанный с данным именем хоста, на котором запущено веб-приложение. По пути запрос потенциально может быть получен и ретранслирован на нескольких маршрутизаторах, но обрабатывается только после подключения к серверу, связанному с именем хоста.

Как только сервер получает запрос, он проверяет его и генерирует ответ. В зависимости от запроса этот ответ может быть веб-страницей, изображением, файлом JavaScript или простым подтверждением либо любым другим файлом. В данном примере предполагается, что пользователь перешел на домашнюю страницу веб-приложения, поэтому сервер в ответ выдает некий HTML-код. Код добавляется к ответу, который затем отправляется обратно через интернет в браузер, отправивший запрос.

Как только браузер пользователя начинает получать ответ, он может приступить к отображению содержимого на экране, но HTML-страница может также ссылаться на другие страницы и ссылки на сервере. Чтобы отобразить веб-страницу целиком, вместо статического бесцветного низкоуровневого HTML-файла браузер должен повторить процесс запроса, извлекая каждый файл, на который есть ссылка. HTML, изображения, CSS-стили и файлы JavaScript, обеспечивающие дополнительное поведение, – все они извлекаются с использованием одного и того же HTTP-запроса.

Практически все взаимодействия, происходящие в интернете, являются фасадом одного и того же базового процесса. Для полной отрисовки базовой веб-страницы может потребоваться всего несколько простых запросов, тогда как для современной большой веб-страницы могут потребоваться сотни запросов. На момент написания этих строк домашняя страница Amazon.com (www.amazon.com), например, делает 410 запросов, включая запросы на 4 файла CSS, 12 файлов JavaScript и 299 файлов изображений!

Теперь, когда вы прочувствовали процесс, посмотрим, как ASP.NET Core динамически генерирует ответ на сервере.

1.4.2 Как ASP.NET Core обрабатывает запрос?

Когда вы создаете веб-приложение с помощью ASP.NET Core, браузеры по-прежнему будут использовать тот же протокол HTTP, что и раньше, для обмена данными с вашим приложением. Сам ASP.NET Core охватывает все, что происходит на сервере для обработки запроса, включая проверку корректности запроса, обработку данных для входа и генерирование HTML.

Как и в случае с примером веб-страницы, процесс запроса начинается, когда браузер пользователя отправляет HTTP-запрос на сервер, как показано на рис. 1.3.

Приложение получает запрос из сети. Каждое приложение ASP.NET Core имеет встроенный веб-сервер, по умолчанию это Kestrel, который отвечает за получение низкоуровневых запросов и создание внутреннего представления данных, объекта `HttpContext`, который может использоваться остальной частью приложения.

Исходя из этого представления, приложение должно иметь все детали, необходимые для создания соответствующего ответа на запрос. Оно может использовать данные, хранящиеся в `HttpContext`, для формирования соответствующего ответа. Это может быть генерирование некоего HTML, возврат сообщения «Доступ запрещен» или отправка электронного письма в зависимости от требований приложения.

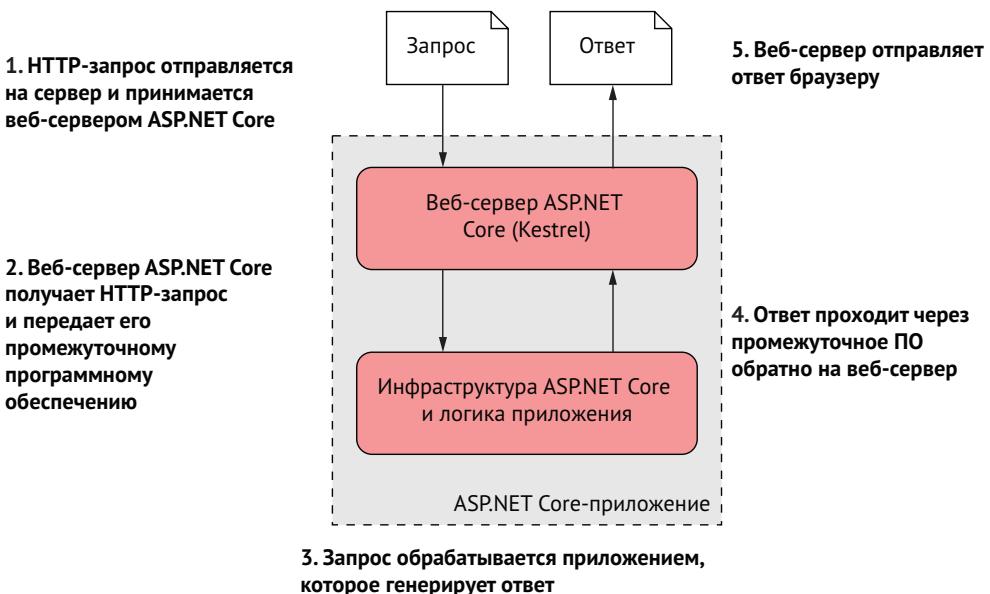


Рис. 1.3 Как приложение ASP.NET Core обрабатывает запрос. Приложение ASP.NET Core получает запрос и запускает собственный веб-сервер. Веб-сервер обрабатывает запрос и передает его телу приложения, которое генерирует ответ и возвращает его веб-серверу. Веб-сервер отправляет этот ответ браузеру

Как только приложение завершит обработку запроса, оно вернет ответ веб-серверу. Веб-сервер ASP.NET Core преобразует представление в низкоуровневый HTTP-ответ и отправит его в сеть, которая направит его в браузер пользователя.

Для пользователя этот процесс выглядит так же, как и для обычного HTTP-запроса, показанного на рис. 1.2, – пользователь отправил запрос и получил ответ. Все различия относятся к серверной части приложения.

Вы видели, как запросы и ответы попадают в приложение ASP.NET Core и из него, но я еще не коснулся того, как генерируется ответ. В этой книге мы рассмотрим компоненты, из которых состоит типичное приложение ASP.NET Core, и то, как они сочетаются друг с другом. Для генерирования ответа в ASP.NET Core требуется много всего, обычно это происходит в течение доли секунды, но на протяжении книги мы будем медленно изучать приложение, подробно описывая каждый из компонентов.

1.5 Что вы узнаете из этой книги

В этой книге вы подробно изучите фреймворк ASP.NET Core. Чтобы извлечь пользу из книги, вы должны быть знакомы с C# или аналогичным объектно ориентированным языком. Также будет полезно базовое знакомство с такими веб-концепциями, как HTML и JavaScript. Вы узнаете:

- как создавать API для протокола HTTP с помощью минимальных API;
- как создавать многостраничные приложения с помощью Razor Pages;
- ключевые концепции ASP.NET Core, среди которых – привязка модели, валидация и маршрутизация;
- как генерировать HTML для веб-страниц с помощью синтаксиса Razor и тег-хелперов;
- как использовать такие возможности, как внедрение зависимостей, конфигурирование и журналирование, по мере того как ваши приложения будут становиться все сложнее;
- как защитить свое приложение с помощью передовых методов безопасности.

На протяжении всей книги мы будем использовать множество примеров, чтобы изучать и исследовать различные концепции. Приведенные здесь примеры, как правило, небольшие и автономные, поэтому мы можем сосредоточиться на одной функциональной возможности за раз.

Для большинства примеров в этой книге я буду использовать Visual Studio, но вы можете применять свой любимый редактор или интегрированную среду разработки. В приложении A содержится подробная информация о настройке редактора или интегрированной среды разработки и установке набора средств разработки .NET 7. Несмотря на то что в примерах этой книги показаны инструменты для Windows, все, что вы видите, в одинаковой мере применимо для платформ Linux или Mac.

СОВЕТ .NET 7 можно скачать на странице <https://dotnet.microsoft.com/download>. В приложении А содержатся дополнительные сведения о том, как настроить среду разработки для работы с ASP.NET Core и .NET 7.

В главе 2 мы более подробно изучим типы приложений, которые можно создавать с помощью ASP.NET Core, а также рассмотрим его преимущества перед более старыми платформами ASP.NET и .NET Framework.

Резюме

- ASP.NET Core – это кросс-платформенный высокопроизводительный фреймворк для создания веб-приложений с открытым исходным кодом;
- ASP.NET Core работает на платформе .NET, которая ранее носила название .NET Core;
- для создания многостраничных веб-приложений с отрисовкой на стороне сервера можно использовать Razor Pages или контроллеры MVC;
- для создания REST-совместимых API или HTTP API можно использовать минимальные API или веб-API;
- для создания высокоэффективных межсервисных RPC-приложений можно использовать gRPC;
- вы можете использовать Blazor WebAssembly для создания клиентских приложений, которые запускаются в браузере, и Blazor Server для создания приложений с отслеживанием состояния и отрисовкой на стороне сервера, которые отправляют обновления пользовательского интерфейса через соединение WebSocket;
- Microsoft рекомендует ASP.NET Core и .NET 7 или более позднюю версию для всех новых веб-разработок взамен устаревших платформ ASP.NET и .NET Framework;
- получение веб-страницы включает отправку HTTP-запроса и получение HTTP-ответа;
- ASP.NET Core позволяет динамически формировать ответы на отправленный запрос;
- приложение ASP.NET Core содержит веб-сервер, который служит точкой входа для запроса.

Часть I

Начало работы с минимальными API

В наши дни веб-приложения повсюду: от веб-приложений для социальных сетей и новостных сайтов до приложений на вашем телефоне. За кулисами почти всегда есть сервер, на котором работает веб-приложение или HTTP API. И сейчас потенциальные заказчики, как правило, хотят получать веб-приложения легко масштабируемые, с развертыванием в облаке и с высокой производительностью. Так что начало работы над проектом такого веб-приложения может оказаться непростым делом, а сделать это с такими высокими ожиданиями может оказаться еще более сложной задачей.

Хорошей новостью для вас как читателя является то, что ASP.NET Core был разработан с учетом этих требований. Нужен ли вам простой сайт, сложное веб-приложение для электронной коммерции или распределенная сеть микросервисов, вы можете использовать свои знания ASP.NET Core для создания экономичных веб-приложений, соответствующих вашим потребностям. ASP.NET Core позволяет создавать и запускать веб-приложения в ОС Windows, Linux или macOS. Он имеет модульную структуру, поэтому используются только нужные вам компоненты, при этом приложение остается максимально компактным и производительным.

В первой части мы пройдем весь путь от начала до создания своих первых API. В главе 2 представлен общий обзор ASP.NET Core, кото-

Часть I Начало работы с минимальными API

рый окажется особенно полезным, если вы новичок в веб-разработке в целом. Вы получите первое представление о полноценном приложении ASP.NET Core в главе 3; мы рассмотрим каждый компонент приложения по очереди и посмотрим, как они работают сообща, чтобы сгенерировать ответ.

В главе 4 подробно рассматривается конвейер промежуточного программного обеспечения, который определяет, как обрабатываются входящие веб-запросы и генерируется ответ. Мы рассмотрим несколько стандартных частей промежуточного программного обеспечения и посмотрим, как их можно объединить для создания конвейера нашего приложения.

Главы с 5 по 7 посвящены созданию приложений ASP.NET Core с конечными точками минимальных API, которые представляют собой новый упрощенный подход к созданию API для JSON в приложениях ASP.NET Core. В главе 5 вы узнаете, как создавать конечные точки, генерирующие данные в формате JSON, как использовать фильтры для выявления распространенного поведения и как использовать группы маршрутов для организации своих API. В главе 6 вы узнаете о маршрутизации – процессе сопоставления URL-адресов с конечными точками. А в главе 7 – о привязке модели и валидации.

В первой части много информации, но к концу вы уже будете на пути к созданию простых API с помощью ASP.NET Core. Разумеется, я не буду останавливаться на некоторых более сложных аспектах настройки фреймворка, но вы должны хорошо понимать минимальные API и то, как их можно использовать для создания простых API. В последующих частях книги вы узнаете, как настроить приложение и добавить дополнительные возможности, например профили пользователей и взаимодействие с базой данных. Мы также рассмотрим, как создавать другие типы приложений, например веб-приложения с отрисовкой на стороне сервера, с помощью Razor Pages.

Что такое ASP.NET Core



В этой главе:

- почему был создан ASP.NET Core;
- множество парадигм приложений ASP.NET Core;
- подходы к миграции существующего приложения на ASP.NET Core.

В этой главе я немного расскажу об ASP.NET Core: в чем польза фреймворков для разработки веб-приложений, почему был создан ASP.NET Core и как выбрать, когда использовать ASP.NET Core. Если вы новичок в разработке .NET, то эта глава поможет вам получить краткое представление о .NET. Если вы уже являетесь .NET-разработчиком, я дам вам рекомендации относительно того, подходящее ли сейчас время, для того чтобы рассмотреть возможность перехода на .NET Core и .NET 7, а также расскажу о преимуществах, которые ASP.NET Core может предложить по сравнению с предыдущими версиями ASP.NET.

2.1 Использование фреймворка для создания веб-приложений

Если вы новичок в веб-разработке, возможно, вам будет непросто перейти в область с таким количеством модных словечек и множеством постоянно меняющихся продуктов. Вы можете задаться вопросом, необходимы ли все эти продукты. Насколько сложно вернуть файл с сервера?

Что ж, вполне возможно создать статическое веб-приложение без использования фреймворка, но его возможности будут ограничены. Как только вы захотите обеспечить хоть какую-то безопасность или динамический интерфейс, то, скорее всего, столкнетесь с трудностями и первоначальная простота такого подхода уже не покажется вам хорошим решением.

Подобно тому, как фреймворки для разработки настольных или мобильных приложений могут помочь вам создавать собственные приложения, ASP.NET Core делает написание веб-приложений быстрее, проще и безопаснее, по сравнению с попытками создать все с нуля. Он содержит библиотеки для таких распространенных вещей, как:

- создание динамически изменяющихся веб-страниц;
- возможность входа пользователей в веб-приложение;
- разрешение пользователям использовать свои учетные записи Facebook для входа в веб-приложение;
- обеспечение общей структуры для создания удобных в сопровождении приложений;
- чтение файлов конфигурации;
- обслуживание файлов изображений;
- регистрация запросов, выполняемых к веб-приложению.

Ключом к любому современному веб-приложению является возможность создавать динамические веб-страницы. *Динамическая веб-страница* может отображать разные данные в зависимости от выполнившего вход пользователя или отображать контент, предоставленный пользователями. Без динамической структуры было бы невозможно заходить на сайты или отображать какие-либо персонализированные данные на странице. То есть существование таких сайтов, как Amazon, eBay и Stack Overflow (см. рис. 2.1), было бы невозможным. Веб-фреймворки для создания динамических веб-страниц почти такие же древние, как и сама сеть, и за эти годы Microsoft создала несколько фреймворков, так зачем создавать новый?

2.2 Для чего был создан ASP.NET Core

Разработка ASP.NET Core компанией Microsoft была мотивирована желанием создать фреймворк для разработки веб-приложений с пятью основными целями:

- кросс-платформенный запуск и разработка;
- наличие модульной архитектуры для упрощения сопровождения;
- фреймворк должен разрабатываться как программное обеспечение с открытым исходным кодом;
- соблюдение веб-стандартов;
- возможность применения к текущим тенденциям в веб-разработке, например клиентские приложения и развертывание в облачных окружениях.

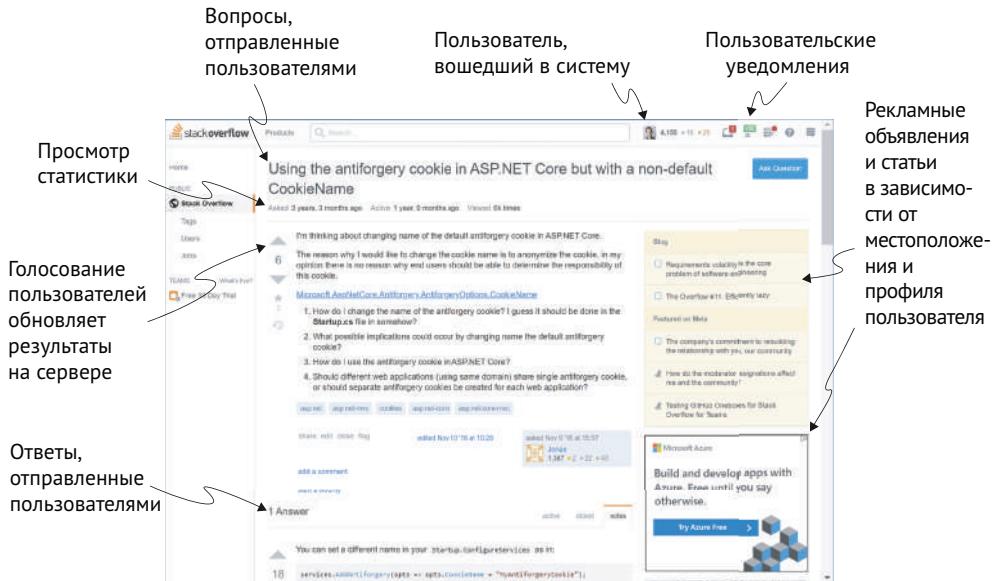


Рис. 2.1 Сайт Stack Overflow (<https://stackoverflow.com>) создан с использованием ASP.NET и почти полностью является динамическим

Для достижения всех этих целей Microsoft требовалась платформа, которая могла бы предоставить основные библиотеки для создания базовых объектов, таких как списки и словари, а также для выполнения таких задач, как простые операции с файлами. До этого момента разработка ASP.NET всегда была сосредоточена на .NET Framework (и зависела от него), предназначенный только для ОС Windows. Для ASP.NET Core компания Microsoft создала легковесную платформу, работающую в Windows, Linux и macOS, под названием .NET Core (впоследствии .NET), как показано на рис. 2.2.

ОПРЕДЕЛЕНИЕ .NET 5 была следующей версией .NET Core после 3.1, за ней последовали .NET 6 и .NET 7. Она представляет собой объединение .NET Core и других платформ .NET в единой среде выполнения и фреймворке. Считалось, что это будущее .NET, поэтому Microsoft решила исключить слово «Core» из названия. Для соответствия языку Microsoft я использую термин .NET 5+ для обозначения .NET 5, .NET 6 и .NET 7, а термин .NET Core – для обозначения предыдущих версий.

.NET Core (и его преемник .NET 5+) использует многие из тех же API, что и .NET Framework, но является более модульным. Он реализует набор возможностей, отличный от тех, что есть в .NET Framework, с целью предоставить более простую модель программирования и современные API. Это отдельная платформа, а не ответвление .NET Framework, хотя она использует аналогичный код для многих своих API.

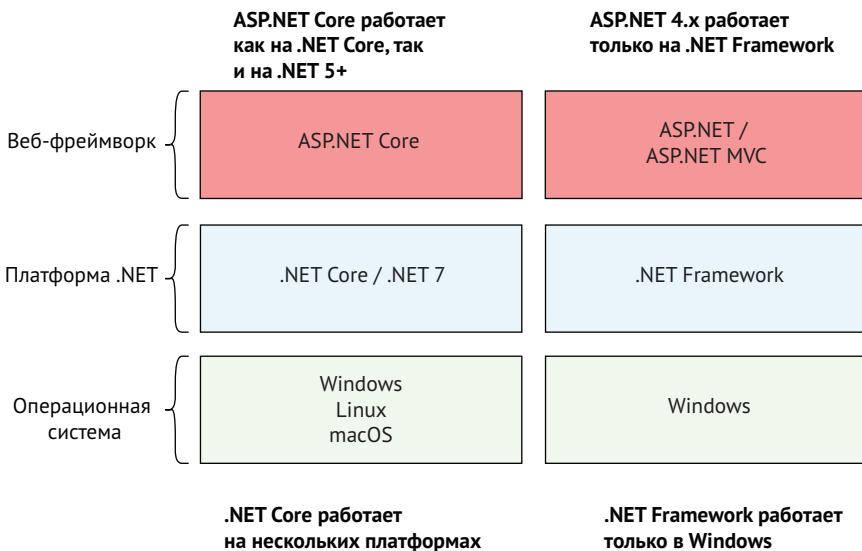


Рис. 2.2. Связь между ASP.NET Core, ASP.NET, .NET Core/.NET 5+ и .NET Framework. ASP.NET Core работает на .NET Core и .NET 5+, поэтому может запускаться на разных ОС. И наоборот, ASP.NET работает только на .NET Framework, поэтому он привязан к ОС Windows

ПРИМЕЧАНИЕ Если вы хотите узнать больше об экосистеме .NET, то можете прочитать две публикации в моем блоге: «Understanding the .NET ecosystem: The evolution of .NET into .NET 7» (<http://mng.bz/AoOW>) и «Understanding the .NET ecosystem: The introduction of .NET Standard» (<http://mng.bz/ZqPZ>).

Преимущества и недостатки ASP.NET

ASP.NET Core – это новейшая версия популярного веб-фреймворка ASP.NET от корпорации Microsoft, появившаяся в июне 2016 года. В предыдущих версиях ASP.NET было много дополнительных обновлений, в которых основное внимание уделялось высокой производительности разработчиков и приоритет отдавался обратной совместимости. В ASP.NET Core нет обратной совместимости со старой версией, в новой версии внесены значительные архитектурные изменения, пересматривающие способ проектирования и сборки веб-фреймворка.

ASP.NET Core во многом наследует ASP.NET, и многие функции, которые были перенесены в него, существовали и прежде, но ASP.NET Core – это новый фреймворк. В данном случае был переписан весь стек технологий, включая как веб-фреймворк, так и базовую платформу.

В основе изменений лежит философия, согласно которой ASP.NET должен держаться независимо по сравнению с другими современными фреймворками, однако разработчикам .NET по-прежнему должно быть здесь все знакомо.

Чтобы понять, почему корпорация Microsoft решила создать новый фреймворк, важно понимать преимущества и недостатки предыдущего веб-фреймворка.

Первая версия ASP.NET была выпущена в 2002 году как часть .NET Framework 1.0. Представленная им парадигма веб-форм ASP.NET существенно отличалась от традиционных окружений разработки скриптов классических ASP и PHP. Технология Web Forms позволила разработчикам быстро создавать веб-приложения с помощью графического конструктора и простой модели событий, отражающей методы создания настольных приложений.

Фреймворк ASP.NET позволил разработчикам быстро создавать новые приложения, но со временем экосистема веб-разработки изменилась. Стало очевидно, что у технологии Web Forms много проблем, которые особенно проявлялись при сборке более крупных приложений. Отсутствие возможности тестирования, сложная модель с отслеживанием состояния и ограниченное влияние на сгенерированный HTML (что затрудняло разработку на стороне клиента) заставили разработчиков присмотреться к другим вариантам.

В ответ корпорация Microsoft выпустила первую версию ASP.NET MVC в 2009 году на базе концепции «модель–представление–контроллер», распространенного паттерна, используемого в других фреймворках, таких как Ruby on Rails, Django и Java Spring. Этот фреймворк позволил отделять элементы пользовательского интерфейса от логики приложения, упростил тестирование и предоставил более жесткий контроль над процессом генерирования HTML.

ASP.NET MVC прошел еще четыре этапа с момента первого выпуска, но все они были построены на одном и том же базовом фреймворке, предоставляемом файлом System.Web.dll. Эта библиотека является частью .NET Framework, поэтому она предустановлена во всех версиях Windows. Она содержит весь основной код, используемый ASP.NET при создании веб-приложения.

У этой зависимости есть свои преимущества и недостатки. С одной стороны, фреймворк ASP.NET – это надежная, проверенная в боях платформа, которая отлично подходит для создания веб-приложений для Windows. Он предоставляет широкий спектр возможностей, которые использовались в течение многих лет, и хорошо известен практически всем веб-разработчикам, работающим с этой ОС.

С другой стороны, такая зависимость является ограничивающей – изменения в базовом файле System.Web.dll имеют далеко идущие последствия, в результате чего наблюдается замедление скорости развертывания. Это ограничивает степень развития ASP.NET, в результате чего новые выпуски появляются только раз в несколько лет. Также существует явная связь с веб-хостом Windows, Internet Information Service (IIS), что исключает его использование на платформах, отличных от Windows.

Совсем недавно Microsoft заявила, что .NET Framework больше не будет развиваться. Ее не будут удалять или заменять, но и никаких новых функциональных возможностей она не получит. Следовательно, ASP.NET на базе System.Web.dll также не будет получать новые возможности или обновления.

В последние годы многие веб-разработчики начали обращать внимание на кросс-платформенные веб-фреймворки, которые могут работать в Windows, а также в Linux и macOS. Microsoft почувствовала, что пришло время создать фреймворк, который больше не привязан к Windows. Так на свет появился ASP.NET Core.

С помощью .NET 7 можно создавать консольные приложения, работающие на разных ОС. Microsoft создала ASP.NET Core как дополнительный уровень поверх консольных приложений, чтобы преобразование в веб-приложение включало добавление и составление библиотек, как показано на рис. 2.3.

Вы пишете консольное приложение .NET 7, которое запускает экземпляр веб-сервера ASP.NET Core

По умолчанию Microsoft предоставляет кросс-платформенный веб-сервер под названием Kestrel

Логика вашего веб-приложения управлется Kestrel. Вы будете использовать различные библиотеки для активации таких функций, как журналирование и генерирование HTML-кода, по мере необходимости

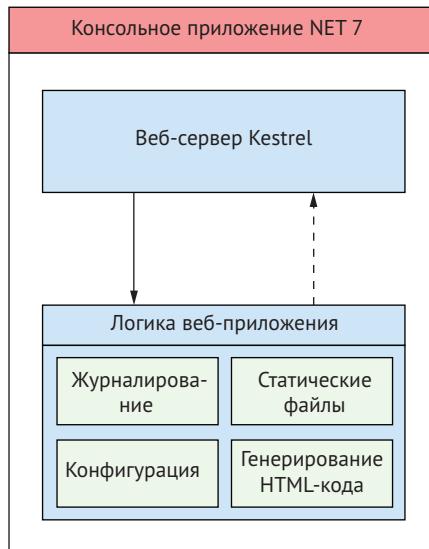


Рис. 2.3. Модель приложения ASP.NET Core. Платформа .NET 7 предоставляет базовую модель консольного приложения для запуска приложений командной строки. При добавлении библиотеки веб-сервера эта модель преобразуется в веб-приложение ASP.NET Core. Используя различные библиотеки, можно добавить другие возможности, например конфигурирование и журналирование

Когда вы добавляете веб-сервер ASP.NET Core в свое приложение .NET 7, консольное приложение может работать как веб-приложение. ASP.NET Core содержит огромное количество API, но вам редко понадобятся все доступные там возможности. Некоторые из них встроены и будут появляться практически в каждом создаваемом вами приложении, например в приложениях для чтения конфигурационных файлов или журналирования. Другие функциональные возможности предоставляются отдельными библиотеками и построены на основе этих базовых возможностей для обеспечения функциональности конкретного приложения, например сторонний вход в систему через Facebook или Google.

Большинство библиотек и API, которые вы будете использовать в ASP.NET Core, доступны на GitHub в репозиториях организации Microsoft .NET на странице <https://github.com/dotnet/aspnetcore>. Там вы можете найти основные API, включая API аутентификации и журналирования, а также множество периферийных библиотек, например сторонние библиотеки аутентификации.

Все приложения ASP.NET Core имеют одинаковую базовую конфигурацию, но в целом это гибкий фреймворк, что дает вам возможность создавать свои собственные соглашения о коде. Эти распространенные API, основанные на них библиотеки расширений и соглашения о проектировании, которые они продвигают, – все это объединено под несколько расплывчатым термином ASP.NET Core.

2.3 Парадигмы ASP.NET Core

В главе 1 вы узнали, что ASP.NET Core – это универсальный веб-фреймворк, который можно использовать для создания самых разных приложений. Как вы помните из раздела 1.2, основными парадигмами являются:

- *минимальные API* – простые API для протокола HTTP, которые могут потребляться мобильными приложениями или одностраничными браузерными приложениями (SPA);
- *веб-API* – альтернативный подход к созданию API для протокола HTTP, который добавляет больше структуры и возможностей по сравнению с минимальными API;
- *gRPC-API* – используются для создания эффективных API для обмена данными между серверами с использованием протокола gRPC в двоичном формате;
- *Razor Pages* – используется для создания многостраничных приложений с отрисовкой на стороне сервера;
- *контроллеры MVC* – аналог Razor Pages; используется для серверных приложений, но без страничной парадигмы;
- *Blazor WebAssembly* – фреймворк для создания одностраничных приложений на основе браузера, использующий стандарт WebAssembly. Аналогичен таким JavaScript-фреймворкам, как Angular, React и Vue;
- *Blazor Server* – используется для создания приложений с отслеживанием состояния с отображением на стороне сервера, которые отправляют события пользовательского интерфейса и обновления страниц через WebSockets, чтобы дать ощущение одностраничного приложения на стороне клиента, обеспечивая при этом легкость разработки приложения с отрисовкой на стороне сервера.

Все эти парадигмы используют базовые функциональные возможности ASP.NET Core и предоставляют дополнительные функции. Каждая парадигма подходит для своего стиля веб-приложения или API, поэтому некоторые из них могут подойти лучше других, в зависимости от создаваемого вами приложения.

Традиционные веб-приложения на основе страниц с отрисовкой на стороне сервера являются основой разработки ASP.NET как в предыдущей версии ASP.NET, так и теперь в ASP.NET Core. Парадигмы *Razor Pages* и *контроллеры MVC* предоставляют два несколько разных стиля для создания приложений такого типа, но имеют много общих концепций, как будет показано во второй части. Эти парадигмы могут быть полезны для создания многофункциональных, динамичных веб-сайтов, будь то сайты электронной коммерции, системы управления контентом (CMS) или большие многоуровневые приложения. Например, проект CMS Orchard Core¹ с открытым исходным кодом (рис. 2.4) и проект Cloudscribe² CMS созданы с использованием ASP.NET Core.

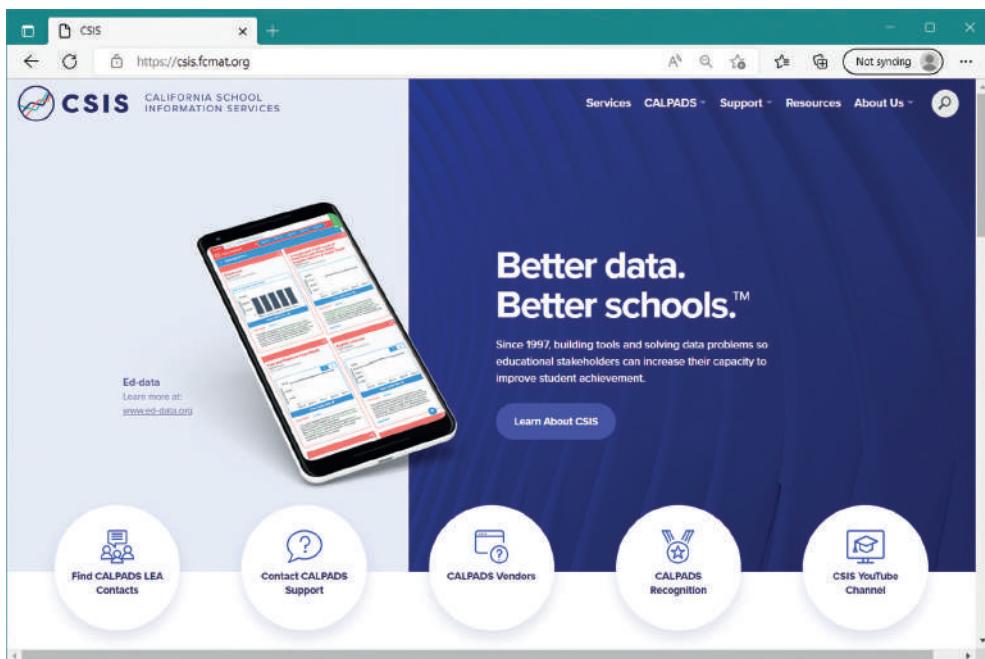


Рис. 2.4. Сайт CSIS (<https://csis.fcmat.org>) создан с использованием Orchard Core и ASP.NET Core

Помимо приложений с отрисовкой на стороне сервера, ASP.NET Core идеально подходит для создания сервера REST API для обработки HTTP-запросов. Независимо от того, создаете ли вы мобильное приложение, одностраничное JavaScript-приложение с использованием Angular, React, Vue или какого-либо другого клиентского фреймворка, легко создать приложение ASP.NET Core, которое будет выступать в качестве серверного API, используя такие парадигмы, как минимальный API и веб-API, встроенные в ASP.NET Core. Вы узнаете о минимальных API в первой части и о веб-API в главе 20.

¹ Orchard Core (<https://orchardcore.net>). Исходный код можно найти на странице <https://github.com/OrchardCMS/OrchardCore>.

² Проект Cloudscribe (<https://www.cloudscribe.com>). Исходный код можно найти на странице <https://github.com/cloudscribe>.

ОПРЕДЕЛЕНИЕ Аббревиатура *REST* означает *representational state transfer* (передача репрезентативного состояния). REST-совместимые приложения обычно используют легковесные HTTP-вызовы без сохранения состояния и применяются для чтения, публикации или передачи данных при их создании либо обновлении, а также для удаления данных.

ASP.NET Core не ограничивается созданием REST-совместимых служб. Легко создать для своего приложения веб-службу или службу в стиле удаленного вызова процедур (RPC), используя, например, gRPC, как показано на рис. 2.5. В простейшем случае ваше приложение может предоставить только одну конечную точку для обработки запросов! ASP.NET Core идеально подходит для создания простых служб благодаря кросс-платформенной поддержке и легковесному дизайну.

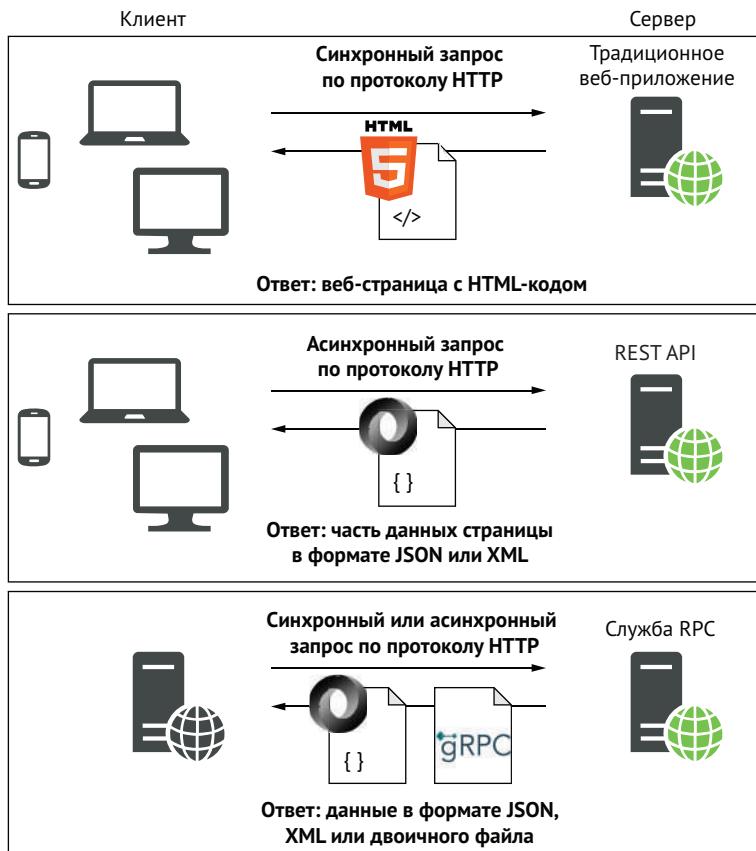


Рис. 2.5. ASP.NET Core может выступать в качестве серверного приложения для различных клиентов: он может обслуживать HTML-страницы для традиционных веб-приложений, действовать как REST API для клиентских одностраничных приложений или как специальная RPC-служба для клиентских приложений

ОПРЕДЕЛЕНИЕ gRPC – это современная высокопроизводительная платформа RPC с открытым исходным кодом. Более подробную информацию можно найти на странице <https://grpc.io>.

Помимо веб-приложений с отрисовкой на стороне сервера, API и конечных точек gRPC, ASP.NET Core включает в себя фреймворк Blazor, который можно использовать для создания двух совершенно разных стилей приложений. Приложения Blazor WebAssembly (WASM) запускаются непосредственно в браузере так же, как традиционные JavaScript-фреймворки для создания одностраничных приложений, например Angular и React. Ваш код .NET компилируется в WebAssembly (<https://webassembly.org>) или выполняется в среде выполнения .NET, скомпилированной для WASM, а браузер загружает и запускает его так же, как и приложение JavaScript. Таким образом, вы можете создавать интерактивные клиентские приложения, используя C# и все уже известные вам API и библиотеки .NET.

Напротив, приложения Blazor Server выполняются на сервере. Каждый щелчок мыши или событие клавиатуры отправляется на сервер через WebSockets. Затем сервер вычисляет изменения, которые следует внести в пользовательский интерфейс, и отправляет необходимые изменения обратно клиенту, который обновляет страницу в браузере. В результате получается приложение с сохранением состояния, которое работает на стороне сервера, но может использоваться для создания интерактивных одностраничных приложений. Главный недостаток Blazor Server заключается в том, что для него требуется постоянное подключение к интернету.

ПРИМЕЧАНИЕ В этой книге я сосредоточусь на создании традиционных веб-приложений на основе страниц с отрисовкой на стороне сервера и REST-совместимых веб-API. Я также покажу, как создавать сервисы, работающие в фоновом режиме, в главе 34. Для получения дополнительной информации о Blazor рекомендую книгу Криса Сэйнти «*Blazor в действии*» (ДМК Пресс, 2023).

Благодаря возможности применять все эти парадигмы можно использовать ASP.NET Core для создания самых разных приложений, но все же стоит подумать, подходит ли ASP.NET Core для вашего конкретного приложения. На это решение, скорее всего, повлияет и ваш опыт работы с .NET, и приложение, которое вы хотите создать.

2.4 Когда следует выбирать ASP.NET Core

В этом разделе я опишу некоторые моменты, которые следует учитывать при принятии решения, касающиеся использования ASP.NET Core и .NET 7 вместо устаревшего фреймворка ASP.NET. В большинстве случаев будет принято решение использовать ASP.NET Core, но следует учитывать ряд важных предостережений.

При выборе платформы нужно учитывать множество факторов, не все из которых являются техническими. Одним из них является

уровень поддержки, которую вы можете ожидать от его создателей. Для некоторых компаний ограниченная поддержка может быть одним из основных препятствий на пути внедрения ПО с открытым исходным кодом. К счастью, Microsoft обязалась обеспечить полную поддержку версий .NET и ASP.NET Core с долгосрочной поддержкой (LTS) в течение как минимум трех лет с момента их выпуска. А поскольку вся разработка происходит открыто, иногда вы можете получать ответы на свои вопросы от общего сообщества, а также напрямую от Microsoft.

ПРИМЕЧАНИЕ С официальной политикой поддержки Microsoft можете ознакомиться на странице <http://mng.bz/RxXP>.

При принятии решения о том, использовать ли ASP.NET Core, следует учитывать два основных аспекта: являетесь ли вы разработчиком .NET и создаете новое приложение или хотите преобразовать существующее.

2.4.1 Если вы новичок в разработке .NET

Если вы новичок в разработке .NET, то выбрали отличное время, чтобы стать частью сообщества разработчиков .NET. Многие проблемы роста, связанные с новым фреймворком, были решены, в результате получился стабильный высокопроизводительный кросплатформенный фреймворк для разработки приложений.

Основным языком разработки .NET и ASP.NET Core в частности является C#. У этого языка огромное количество последователей, и не зря! Будучи объектно ориентированным языком на основе C, он кажется знакомым тем, кто привык к C, Java и многим другим языкам. Кроме того, он имеет множество мощных функциональных возможностей, например Language Integrated Query (LINQ), замыкания и конструкции асинхронного программирования. Язык C# также доступен в открытом доступе на GitHub, как и компилятор C# от Microsoft под кодовым названием Roslyn (<https://github.com/dotnet/roslyn>).

ПРИМЕЧАНИЕ В этой книге я использую C# и расскажу о некоторых новых возможностях, которые он предоставляет, но не буду обучать этому языку с нуля. Если вы хотите изучить C#, рекомендую 4-е издание книги Джона Скита «C# для профессионалов: тонкости программирования» (Вильямс, 2019) и Code Like a Pro «in C#» Йорта Роденбурга (Manning, 2021).

Одним из крупных преимуществ ASP.NET Core и .NET 7 по сравнению с .NET Framework является тот факт, что они позволяют разрабатывать и запускать приложения на любой платформе. С помощью .NET 7 можно создавать и запускать одно и то же приложение на Mac, Windows и Linux и даже развертывать его в облаке с помощью крошечных контейнеров.

Напоминание о контейнеризации

Традиционно веб-приложения развертывались непосредственно на сервере, а в последнее время – в виртуальной машине. Виртуальные машины позволяют устанавливать операционные системы на уровне виртуального оборудования, абстрагируясь от реального аппаратного обеспечения. У этого подхода есть несколько преимуществ по сравнению с прямой установкой, например простота обслуживания, развертывания и восстановления. К сожалению, виртуальные машины тяжелы как с точки зрения размера файла, так и с точки зрения использования ресурсов.

Вот тут-то и пригодятся контейнеры. Контейнеры намного легковеснее и не имеют больших накладных расходов, как виртуальные машины. Они состоят из нескольких уровней и не требуют загрузки новой операционной системы при запуске. Это означает, что они быстро запускаются и отлично подходят для быстрого развертывания. Контейнеры (в частности, Docker) быстро становятся популярной платформой для сборки больших масштабируемых систем.

Контейнеры не были особенно привлекательным вариантом для приложений ASP.NET, но с появлением ASP.NET Core, .NET 7 и Docker для Windows все изменилось. Легковесное приложение ASP.NET Core, работающее на кросс-платформенном фреймворке .NET 7, идеально подходит для «тонкого» развертывания контейнеров. Дополнительную информацию о параметрах развертывания можно получить в главе 27.

Помимо того что они могут работать на всех платформах, одним из преимуществ .NET является возможность выполнять написание кода и компиляцию только один раз. Ваше приложение компилируется в код на промежуточном языке (IL), не зависящем от платформы. Если в целевой системе установлена среда выполнения .NET 7, то можно запускать скомпилированный код с любой платформы. Можно, например, вести разработку на компьютере, где установлена ОС Mac или Windows, и выполнять развертывание *абсолютно тех же файлов* на промышленных машинах с Linux. Такая возможность наконец-то была реализована с помощью ASP.NET Core и .NET 7.

СОВЕТ Можно пойти еще дальше и упаковать среду выполнения .NET в свое приложение в так называемом автономном развертывании (self-contained deployment – SCD). Таким образом, вы можете развернуть кроссплатформенную систему, и на целевой машине даже не потребуется установка .NET. При использовании SCD сгенерированные файлы развертывания настраиваются для целевой машины, поэтому в этом случае вам больше не придется развертывать одни и те же файлы повсюду.

Многие из доступных сегодня веб-платформ используют схожие устоявшиеся *паттерны проектирования*, и ASP.NET Core не исключение. Например, Ruby on Rails известен использованием паттерна MVC; Node.js известен тем, как он обрабатывает запросы, используя

небольшие дискретные модули (называемые *конвейером*), а внедрение зависимостей доступно в самых разных фреймворках. Если эти методы вам знакомы, вам будет легко перенести их в ASP.NET Core; если они вам в новинку, то можете рассчитывать на использование лучших отраслевых практик!

ПРИМЕЧАНИЕ Паттерны проектирования – это решения распространенных проблем проектирования ПО. Вы познакомитесь с конвейером в главе 4, с внедрением зависимостей – в главах 8 и 9, а с MVC – в главе 19.

Независимо от того, являетесь ли вы новичком в веб-разработке в целом или только в случае с .NET, ASP.NET Core предоставляет богатый набор возможностей, с помощью которых вы можете создавать приложения, не перегружая при этом концепциями, как это делал устаревший фреймворк ASP.NET. С другой стороны, если вы знакомы с .NET, стоит задуматься, не пора ли сейчас обратить свой взор на ASP.NET Core.

2.4.2 *Если вы .NET-разработчик, создающий новое приложение*

Если вы уже являетесь разработчиком .NET Framework, то, вероятно, слышали о .NET Core и ASP.NET Core, но, возможно, опасались приступить к работе слишком рано или не хотели столкнуться с неизбежными проблемами из серии «версия 1». Есть хорошая новость: теперь ASP.NET Core и .NET – это зрелые и стабильные платформы, и настало время рассмотреть возможность использования .NET 7 для своих новых приложений.

Будучи разработчиком .NET, если вы не используете какие-либо специфичные для Windows конструкции, например реестр, то возможность создания и развертывания приложений на разных платформах открывает возможность более дешевого хостинга Linux в облаке или для разработки в macOS без дополнительных затрат без необходимости в виртуальной машине.

.NET является кроссплатформенным фреймворком, однако начиная с версии .NET 5 мы можем использовать специфические функции каждой конкретной платформы. Например, для Windows мы можем использовать службы реестров и каталогов с помощью пакетов обеспечения совместимости, которые делают доступными системные API. Стоит помнить, что при использовании специфических функций конкретной платформы приложения, их использующие, должны запускаться и выполняться в том же окружении или учитывать потенциально недостающие API.

ПОДСКАЗКА Пакет обеспечения совместимости Windows предназначен для облегчения переноса кода из .NET Framework для .NET Core/.NET 5+. См. <http://mng.bz/2DeX>.

Модель хостинга для предыдущего фреймворка ASP.NET была относительно сложной. Она использовала Windows-сервис IIS для предоставления веб-хоста. В кросс-платформенном окружении такой поход невозможен, поэтому была принята альтернативная модель хостинга, отделяющая веб-приложения от основного хоста. Эта возможность привела к разработке Kestrel – быстрого кросс-платформенного HTTP-сервера, на котором может работать ASP.NET Core. Вместо предыдущего варианта, при котором IIS вызывает определенные точки приложения, приложения ASP.NET Core представляют собой консольные приложения, которые самостоятельно размещают веб-сервер и обрабатывают запросы напрямую, как показано на рис. 2.6. Эта модель хостинга концептуально намного проще и позволяет тестировать и отлаживать приложения из командной строки. Хотя в данном случае не обязательно избавляться от IIS (или его аналога) в промышленном окружении.

ASP.NET Core и обратные прокси-серверы

Можно предоставлять доступ к приложениям ASP.NET Core напрямую из интернета, чтобы Kestrel получал запросы непосредственно из Сети. Такой подход полностью поддерживается. Однако чаще используется обратный прокси-сервер (*reverse proxy-server*) между сетью и вашим приложением. В Windows обратным прокси-сервером обычно является IIS, а в Linux или macOS это может быть NGINX, HAProxy или Apache. Еще для проектов ASP.NET Core можно использовать библиотеку YARP (<https://microsoft.github.io/reverse-proxy>), на основе которой вы также можете создать обратный прокси.

Обратный прокси-сервер – это программное обеспечение, отвечающее за получение запросов и их пересылку на соответствующий веб-сервер. Обратный прокси-сервер доступен непосредственно из сети Интернет, тогда как базовый веб-сервер доступен только для прокси. Такая настройка имеет несколько преимуществ, в первую очередь это безопасность и производительность веб-серверов.

Вы можете подумать, что наличие обратного прокси-сервера и веб-сервера несколько излишне. Почему бы не использовать что-то одно? Что ж, одно из преимуществ – возможность отделить ваше приложение от операционной системы сервера. Один и тот же веб-сервер ASP.NET Core, Kestrel, может быть кросс-платформенным и использоваться за различными прокси-серверами, не накладывая каких-либо ограничений на конкретную реализацию. Если вы написали новый веб-сервер, то можете использовать его вместо Kestrel, не изменяя ничего другого в своем приложении.

Еще одним преимуществом обратного прокси-сервера является то, что он может быть защищен от потенциальных угроз из общедоступного интернета. Часто прокси-серверы отвечают за дополнительные аспекты, такие как перезапуск неработающего процесса. Kestrel может оставаться простым HTTP-сервером, не беспокоясь об этих дополнительных функциях, если используется за обратным прокси-сервером. Считайте это простым разделением обязанностей: Kestrel занимается генерированием ответов по протоколу HTTP, а обратный прокси-сервер занимается обработкой подключения к интернету.

ПРИМЕЧАНИЕ По умолчанию при работе в Windows ASP.NET Core запускается *внутри* IIS, как показано на рис. 2.6, что может обеспечить лучшую производительность по сравнению с версией с обратным прокси-сервером. Прежде всего это касается развертывания и не меняет способ создания приложений ASP.NET Core.

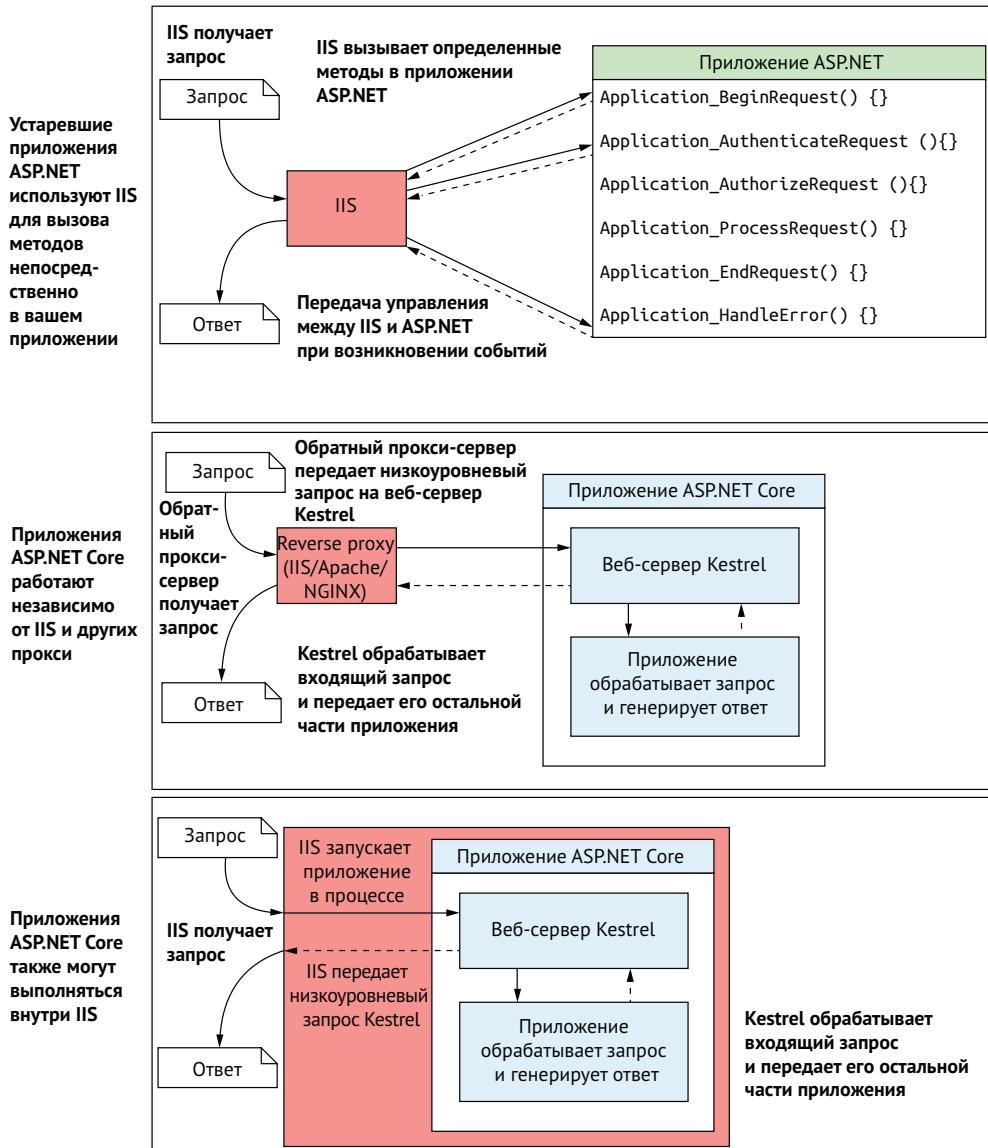


Рис. 2.6 Разница между моделями размещения в ASP.NET (вверху) и ASP.NET Core (внизу). В предыдущей версии ASP.NET IIS тесно связан с приложением. Модель размещения в ASP.NET Core проще; IIS передает запрос на локальный веб-сервер в приложении ASP.NET Core и получает ответ, но не имеет более детального представления о приложении

Изменение модели хостинга для использования встроенного веб-сервера HTTP создало еще одну возможность. В прошлом производительность была в некоторой степени проблемой для приложений ASP.NET. Конечно, можно создавать высокопроизводительные приложения – Stack Overflow (<https://stackoverflow.com>) является тому подтверждением, – но производительность не является приоритетом самого веб-фреймворка, поэтому в конечном итоге это может стать препятствием.

Чтобы быть конкурентоспособным среди кросс-платформенных решений, команда ASP.NET сосредоточилась на том, чтобы сделать HTTP-сервер Kestrel как можно более быстрым. TechEmpower (www.techempower.com/benchmark) уже несколько лет проводит тесты для целого ряда веб-фреймворков для разных языков. В 20-м раунде тестов в виде plaintext TechEmpower объявила, что ASP.NET Core с Kestrel вошел в десятку самых быстрых из более чем 400 протестированных фреймворков¹!

Веб-серверы: проблемы с терминологией

Один из сложных аспектов веб-программирования – это набор часто противоречащих друг другу терминов. Например, если вы использовали IIS, то могли называть его веб-сервером или, возможно, веб-хостом. И наоборот, если вы когда-либо создавали приложение с использованием Node.js, то могли назвать веб-сервером само приложение. Также веб-сервером можно назвать физический компьютер, на котором работает ваше приложение! Точно так же вы, возможно, создавали приложение для интернета и называли его веб-сайтом или веб-приложением, вероятно, несколько произвольно, исходя из наличия изменяемого контента.

В этой книге, когда я говорю «веб-сервер» в контексте ASP.NET Core, я имею в виду HTTP-сервер, который работает как часть вашего приложения ASP.NET Core. По умолчанию это веб-сервер Kestrel, но это не является обязательным требованием. При желании можно было бы написать собственный веб-сервер и использовать его вместо Kestrel.

Веб-сервер отвечает за получение HTTP-запросов и генерирование ответов. В предыдущей версии ASP.NET эту роль выполнял IIS, но в ASP.NET Core веб-сервер – это Kestrel.

В этой книге я буду использовать термин «веб-приложение» только для описания приложений ASP.NET Core, независимо от того, содержат ли они лишь статический контент или являются полностью динамичными. В любом случае это приложения, доступ к которым осуществляется по Сети, поэтому этот термин кажется наиболее подходящим.

Многие улучшения производительности, внесенные в Kestrel, исходили не от самих членов команды ASP.NET, а от участников проекта с от-

¹ Как и всегда в веб-разработке, технологии находятся в постоянном движении, поэтому со временем эти критерии будут меняться. Хотя ASP.NET Core, возможно, и не удержит свое место в топ-10, вы можете быть уверены, что производительность является одним из ключевых моментов команды ASP.NET Core.

крытым исходным кодом на GitHub (<https://github.com/dotnet/aspnetcore>). Открытая разработка означает, что обычно вы видите, что исправления и новая функциональность внедряются в промышленную эксплуатацию быстрее, чем в предыдущей версии ASP.NET, которая зависела от .NET Framework и Windows и поэтому имела длительные циклы выпуска.

Напротив, версия .NET 5 и выше, а следовательно, и ASP.NET Core предполагают регулярные небольшие изменения. Старшие версии будут выпускаться с предсказуемой периодичностью: новая версия будет выходить ежегодно, а новая версия с долгосрочной поддержкой (LTS) – каждые два года (<http://mng.bz/1qrg>). Кроме того, исправления ошибок и незначительные обновления можно выпускать по мере необходимости. Дополнительная функциональность предоставляется в виде пакетов NuGet, независимо от базовой платформы .NET 5+.

ПРИМЕЧАНИЕ NuGet – это менеджер пакетов для .NET, позволяющий импортировать библиотеки в ваши проекты. Он аналогичен Ruby Gems и npm в JavaScript или Maven в Java.

ASP.NET Core спроектирован из слабо связанных модулей, что помогает реализовать подобный подход к выпускам. Эта модульность позволяет использовать в отношении зависимостей принцип «плачу только за нужное», когда вы начинаете с «пустого» приложения и добавляете только те дополнительные библиотеки, которые вам требуются, в отличие от подхода «все сразу», использовавшегося в прежних приложениях ASP.NET. Сейчас даже не обязательно применять MVC! Но не волнуйтесь, это не означает, что в ASP.NET Core мало функциональных возможностей; просто вам нужно подключить их. Некоторые ключевые улучшения включают в себя:

- «конвейер» с компонентами, отвечающими за обработку запроса (middleware pipeline) для определения поведения вашего приложения;
- встроенную поддержку внедрения зависимостей;
- объединенную инфраструктуру UI (MVC) и API (Web API);
- расширяемую систему конфигурации;
- стандартизированную расширяемую систему журналирования;
- по умолчанию используется асинхронное программирование для встроенной масштабируемости на облачных платформах.

Каждая из этих функций была доступна в предыдущей версии ASP.NET, но требовала значительного объема дополнительной работы для настройки. С ASP.NET Core все они уже готовы и ждут подключения!

Microsoft полностью поддерживает ASP.NET Core, поэтому если вы хотите создать новую программную систему, нет серьезных причин не использовать его. Самое большое препятствие, с которым вы, вероятно, столкнетесь, – это желание применить модели программирования, которые больше не поддерживаются в ASP.NET Core, такие как Web Forms или сервер WCF, о чём я расскажу в следующем разделе.

Надеюсь, этот раздел пробудил в вас желание использовать ASP.NET Core для создания новых приложений. Но если вы уже работаете с ASP.NET и рассматриваете возможность переноса существующего приложения ASP.NET в ASP.NET Core, то это уже совсем другой вопрос.

2.4.3 Перенос существующего ASP.NET-приложения на ASP.NET Core

В отличие от новых приложений, существующее приложение, по-видимому, уже предоставляет ценность, поэтому переход на ASP.NET Core в конечном итоге должен принести ощутимо большую пользу, чтобы начать значительно переписывать приложение при миграции с ASP.NET на ASP.NET Core. Преимущества внедрения ASP.NET Core во многом такие же, как и у новых приложений: кросс-платформенное развертывание, модульные функции и акцент на производительность. Несколько их достаточно, во многом будет зависеть от особенностей вашего приложения, но есть характеристики, которые являются явными индикаторами не в пользу перехода:

- ваше приложение использует технологию Web Forms;
- ваше приложение создано с использованием WCF;
- ваше приложение большое, со множеством «продвинутых» функций MVC.

Если вы используете Web Forms, не рекомендуется переходить на ASP.NET Core. Эта технология неразрывно связана с System.Web.dll и как таковая, скорее всего, никогда не будет доступна в ASP.NET Core. Такой переход будет фактически включать в себя переписывание приложения с нуля. Это не только изменение фреймворков, но и изменение парадигм проектирования.

Однако не все потеряно. Сервер Blazor предоставляет компонентное приложение с отслеживанием состояния, похожее на модель приложения веб-форм. Возможно, вы сможете постепенно переносить свое приложение с Web Forms постранично на серверное приложение ASP.NET Core Blazor¹. В качестве альтернативы можно постепенно внедрять концепции веб-API в свое приложение с Web Forms, уменьшая зависимость от устаревших конструкций, например View-State, с целью в конечном итоге перейти к веб-API ASP.NET Core.

Фреймворк Windows Communication Foundation (WCF) лишь частично поддерживается в ASP.NET Core. Можно создавать клиентские службы WCF, используя библиотеки, предоставляемые ASP.NET Core (<https://github.com/dotnet/wcf>), а также создавать серверные службы WCF, используя поддерживаемый Microsoft проект CoreWCF², управляемый

¹ Сообщество предпринимает усилия по созданию Blazor-версий распространенных компонентов WebForms (<http://mng.bz/PzPP>). Также см. электронную книгу по Web Forms для разработчиков Blazor на странице <http://mng.bz/lgDv>.

² Вы можете найти библиотеки CoreWCF на странице <https://github.com/corewcf/corewcf>, а подробную информацию об обновлении службы WCF до .NET 5 и выше – на странице <http://mng.bz/mVg2>

сообществом. Эти библиотеки не поддерживают все API, доступные в .NET Framework WCF (распределенные транзакции и некоторые форматы безопасности сообщений, например), поэтому если вам непременно нужно поддерживать WCF, то лучше избегать использования ASP.NET Core на данный момент.

СОВЕТ Если вам нравится программирование в стиле RPC, основанное на контрактах, но у вас нет жестких требований к WCF, рассмотрите возможность использования gRPC. gRPC – это современный фреймворк для вызова удаленных процедур со множеством концепций, аналогичных WCF, и он поддерживается ASP.NET Core из коробки (<http://mng.bz/wv9Q>).

Если у вас сложное приложение и оно широко использует предыдущие точки расширения MVC, или веб-API, или обработчики сообщений, то его перенос в ASP.NET Core может оказаться более трудным. ASP.NET Core имеет множество функциональных возможностей, аналогичных предыдущей версии ASP.NET MVC, но базовая архитектура у него иная. Некоторые ранее существовавшие возможности не имеют прямых заменителей, поэтому их придется переосмыслить. Чем крупнее приложение, тем труднее будет перейти на ASP.NET Core. Microsoft предполагает, что перенос приложения из ASP.NET MVC в ASP.NET Core – это по крайней мере такого же порядка изменение, как перенос приложения из ASP.NET Web Forms в ASP.NET MVC. Если это вас не пугает, тогда все в порядке!

Если приложение используется редко и не является частью основного бизнеса или не требует значительного развития в ближайшем будущем, то я настоятельно рекомендую вам *не* пытаться переходить на ASP.NET Core. Microsoft будет поддерживать платформу .NET Framework в обозримом будущем (от нее зависит сама Windows!), и отдача от переноса этих «второстепенных» приложений вряд ли стоит затраченных усилий.

Итак, когда же следует переносить приложение на ASP.NET Core? Как я уже упоминал, наиболее подходящая возможность – это небольшие новые проекты с нуля, а не существующие приложения. Тем не менее если существующее приложение невелико или потребует значительного развития в будущем, перенос может быть хорошим вариантом.

При переносе приложения всегда лучше работать небольшими этапами, если это возможно, а не пытаться сделать все сразу. К счастью, Microsoft предоставляет инструменты для этой цели. Набор адаптеров System.Web, обратный прокси-сервер на основе .NET под названием YARP (Yet Another Reverse Proxy; <http://mng.bz/qr92>) и инструменты, встроенные в Visual Studio, могут помочь вам реализовать паттерн Strangler Fig (<http://mng.bz/rW6J>). Этот инструментарий позволяет переносить приложение по одной странице/API за раз, снижая риск, связанный с переносом приложения ASP.NET на ASP.NET Core.

В этой главе мы рассмотрели исторический контекст ASP.NET Core, а также некоторые преимущества его внедрения. В главе 3 мы создадим наше первое приложение на основе шаблона и запустим его. Мы разберем все основные компоненты, из которых оно состоит, и посмотрим, как они работают вместе при отображении веб-страницы.

Резюме

- Веб-фреймворки позволяют легко создавать динамические веб-приложения;
- ASP.NET Core – это веб-фреймворк, созданный с учетом современных практик архитектуры ПО и модульности;
- ASP.NET Core работает на кросс-платформенной платформе .NET 7. Вы можете получить доступ к специфичным для Windows функциям, например реестру Windows, с помощью пакета обеспечения совместимости Windows;
- .NET 5, .NET 6 и .NET 7 – следующие версии .NET Core после .NET Core 3.1;
- ASP.NET Core лучше всего использовать для новых проектов;
- устаревшие технологии, среди которых сервер WCF и Web Forms, нельзя использовать напрямую с ASP.NET Core, но у них есть аналоги и вспомогательные библиотеки, которые могут помочь при переносе приложений ASP.NET на ASP.NET Core;
- можно постепенно преобразовать существующее приложение ASP.NET в ASP.NET Core, используя шаблон Strangler Fig и инструменты и библиотеки, предоставляемые Microsoft;
- приложения ASP.NET Core часто защищаются от интернета с помощью обратного прокси-сервера, который пересыпает запросы приложению.

Наше первое приложение

3

В этой главе:

- создание первого веб-приложения с использованием ASP.NET Core;
- запуск приложения;
- разбор компонентов приложения.

После прочтения предыдущих глав у вас должно было сложиться представление о том, как работают приложения ASP.NET Core и когда их следует использовать. Кроме того, вы должны были настроить среду разработки, которую можно использовать, чтобы приступить к созданию приложения.

СОВЕТ См. приложение A, где содержатся инструкции по установке набора средств разработки .NET 7 и выбору редактора и IDE.

В этой главе мы займемся созданием нашего первого веб-приложения. Вы на практике попробуете понять, как все работает, а в последующих главах я покажу, как проводить настройку и сборку ваших собственных приложений. По мере прохождения этой главы у вас появится представление о компонентах, из которых состоит приложение ASP.NET Core, и об общем процессе его построения. Большинство приложений, которые вы разрабатываете, будут создаваться с использованием шаблона, поэтому рекомендуется как можно скорее ознакомиться с его настройкой.

ОПРЕДЕЛЕНИЕ Шаблон предоставляет базовый код, необходимый для создания приложения. Шаблон можно использовать в качестве отправной точки для создания новых приложений.

Вначале я покажу, как создать простое приложение ASP.NET Core с помощью одного из шаблонов Visual Studio. Если вы используете другие инструменты, например интерфейс командной строки .NET, вам будут доступны аналогичные шаблоны. В этой главе я использую Visual Studio 2022, ASP.NET Core 7 и .NET 7, но также дам советы по работе с интерфейсом командной строки .NET.

СОВЕТ Код приложения для этой главы можно посмотреть в репозитории GitHub на странице <http://mng.bz/5wj1>.

После того как мы создадим приложение, я покажу вам, как восстановить все необходимые зависимости, скомпилировать приложение и запустить его, чтобы увидеть результат. Приложение будет простым и будет содержать основу приложения ASP.NET Core, которое содержит фразу «Hello World!».

После запуска приложения наш следующий шаг – понять, что происходит! Мы совершим путешествие по приложению ASP.NET Core, поочередно рассматривая каждый файл в шаблоне. Вы почувствуете, как устроено приложение ASP.NET Core, и увидите, как выглядит код C# для минимально возможного приложения.

И наконец, вы увидите, как расширить ваше приложение для обработки запросов к статическим файлам, а также как создать простой API, который возвращает данные в стандартном формате JSON.

Пусть вас не беспокоит, если какие-то части проекта покажутся вам запутанными или сложными; вы подробно изучите каждый раздел по ходу чтения книги. К концу этой главы вы будете иметь общее представление о том, как взаимодействуют компоненты приложения ASP.NET Core с момента первого запуска приложения до момента генерации ответа. Однако, прежде чем начать, мы рассмотрим, как приложения ASP.NET Core обрабатывают запросы.

3.1 Краткий обзор приложения ASP.NET Core

В первой главе я описал, как браузер выполняет HTTP-запрос к серверу и получает ответ, который он использует для отображения HTML-кода на странице. ASP.NET Core позволяет динамически генерировать этот код в зависимости от содержимого запроса, чтобы вы могли, например, отображать разные данные в зависимости от текущего пользователя.

Допустим, вы хотите создать веб-приложение, где будет показана информация о вашей компании. Для этого можно создать простое приложение ASP.NET Core и затем добавить в него динамические функции. На рис. 3.1 показано, как приложение будет обрабатывать запрос на отображение страницы.

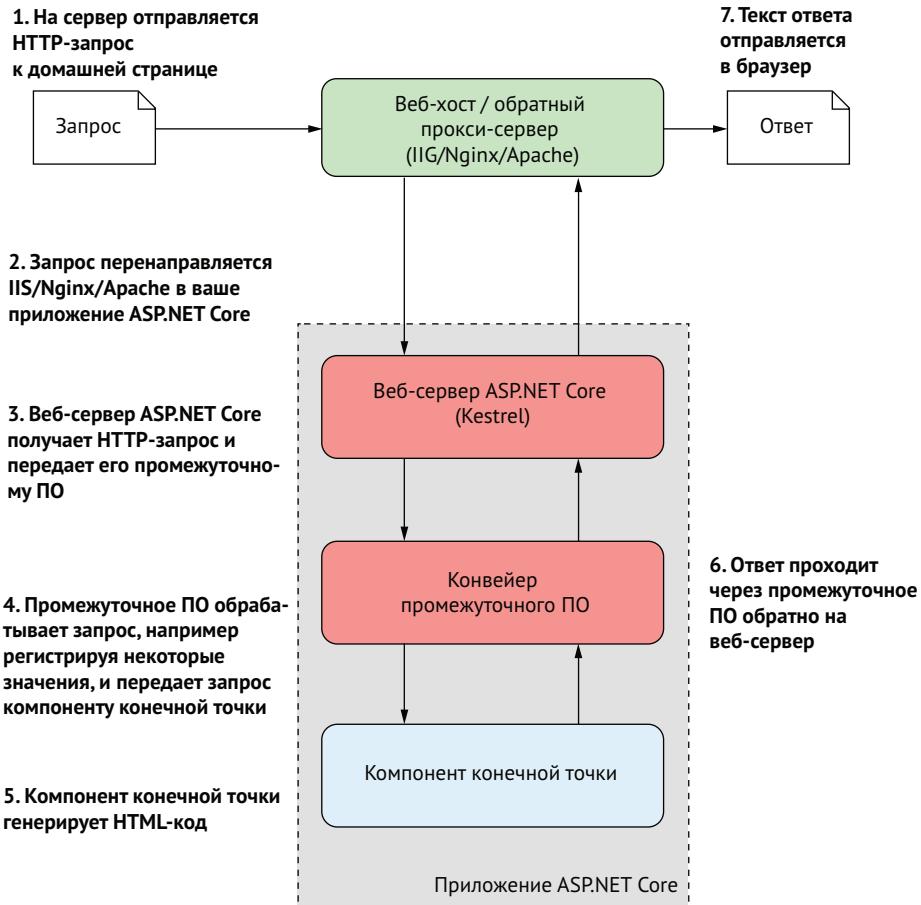


Рис. 3.1 Обзор приложения ASP.NET Core. Приложение получает входящий HTTP-запрос от браузера. Каждый запрос передается в конвейер промежуточного программного обеспечения, который потенциально изменяет его, а затем передает компоненту конечной точки в конце конвейера для формирования ответа. Ответ возвращается через промежуточное ПО на сервер и, наконец, в браузер

Большая часть этой диаграммы должна быть вам знакома по рис. 1.3 из главы 1; запрос и ответ, а также веб-сервер ASP.NET Core – все на месте. Но вы заметите, что я добавил обратный прокси-сервер, чтобы показать распространенный паттерн развертывания приложений ASP.NET Core. Я также расширил само приложение ASP.NET Core, чтобы показать конвейер промежуточного ПО и компонент конечной точки – основную настраиваемую часть вашего приложения, которая генерирует ответ на запрос.

Первым адресатом, кому обратный прокси-сервер пересыпает запрос, является веб-сервер с ASP.NET Core. По умолчанию это кросс-платформенный сервер Kestrel. Kestrel принимает низкоуровневый сетевой запрос и использует его для создания объекта `HttpContext`, который может применяться остальной частью приложения.

Объект HttpContext

Объект `HttpContext`, создаваемый веб-сервером ASP.NET Core, используется приложением как своего рода ящик для хранения данных запроса. Все, что относится к этому конкретному запросу и последующему ответу, может быть связано с ним и сохранено в нем. Это могут быть свойства запроса, сервисы, связанные с запросом, загруженные данные или возникшие ошибки. Веб-сервер заполняет начальный объект `HttpContext` данными исходного HTTP-запроса и другими деталями конфигурации и передает его остальной части приложения.

ПРИМЕЧАНИЕ Kestrel – не единственный HTTP-сервер, доступный в ASP.NET Core, но он кросс-платформенный и самый производительный. При запуске в процессе в *Internet Information Services* (IIS) используется другой веб-сервер, IIS HTTP Server. Основная альтернатива – HTTP.sys – работает только в Windows и не может использоваться с IIS¹.

Kestrel отвечает за получение данных запроса и построение представления запроса на языке C#, но не пытается сгенерировать ответ напрямую. Для этого Kestrel передает объект `HttpContext` конвейеру промежуточного ПО, который есть в каждом приложении ASP.NET Core. Это серия компонентов, которые обрабатывают входящий запрос для выполнения распространенных операций, таких как ведение журнала, обработка исключений или обслуживание статических файлов.

ПРИМЕЧАНИЕ Подробно о конвейере промежуточного ПО вы узнаете в главе 4.

В конце конвейера находится компонент конечной точки. Он отвечает за вызов кода, который генерирует окончательный ответ. В большинстве приложений это будет конечная точка MVC, Razor Pages или минимального API.

Большинство приложений ASP.NET Core следуют этой базовой архитектуре, и пример, приведенный в данной главе, ничем от них не отличается. Сначала вы узнаете, как создать и запустить приложение, а затем мы рассмотрим соответствие кода схеме, изображенной на рис. 3.1. Итак, не тратя времени попусту, приступим к созданию приложения!

3.2 Создаем наше первое приложение ASP.NET Core

В этом разделе мы создадим приложение с минимальным API, которое возвращает в ответ фразу «Hello World!» при вызове HTTP API. Это самое простое приложение ASP.NET Core, которое вы можете

¹ Если вы хотите узнать больше о Kestrel, HTTP-сервере IIS и HTTP.sys, в этой документации описаны различия между ними: <http://mng.bz/6DgD>.

создать, но оно демонстрирует многие фундаментальные концепции создания и запуска приложений с помощью .NET.

Приложение ASP.NET Core можно создавать разными способами, это зависит от инструментов и операционной системы, которые вы используете. Каждый набор инструментов будет иметь несколько разные шаблоны, однако у них много общего. Пример, используемый в этой главе, основан на шаблоне Visual Studio 2022, но вы легко можете использовать шаблоны из интерфейса командной строки .NET или Visual Studio для Mac.

ПРИМЕЧАНИЕ Напоминаю, что на протяжении всей книги я использую Visual Studio 2022 и ASP.NET Core с .NET 7.

Подготовка и запуск приложения обычно включают в себя четыре основных шага, которые мы рассмотрим в данной главе:

- 1 создание (generate) – для начала создаем базовое приложение на основе шаблона;
- 2 восстановление (restore) – восстанавливаем все пакеты и зависимости в локальную папку проекта с помощью NuGet;
- 3 сборка (build) – компилируем приложение и генерируем все необходимые ресурсы;
- 4 запуск (run) – запускаем скомпилированное приложение.

Visual Studio и интерфейс командной строки .NET содержат множество шаблонов ASP.NET Core для создания различных типов приложений. Например:

- *приложение с минимальным API* – приложения HTTP API, возвращающие данные в формате JSON, которые могут использоваться одностраничными (SPA) и мобильными приложениями. Обычно они используются вместе с клиентскими приложениями, такими как Angular и React.js, или мобильными приложениями;
- *веб-приложение Razor Pages* – приложения Razor Pages генерируют HTML-код на сервере и предназначены для просмотра пользователями непосредственно в веб-браузере;
- *приложение MVC* – эти приложения похожи на приложения Razor Pages в том смысле, что они генерируют HTML-код на сервере и предназначены для просмотра пользователями непосредственно в веб-браузере. Вместо Razor Pages они используют традиционные контроллеры MVC;
- *приложение веб-API* – приложения веб-API схожи с минимальными API-приложениями, так как также используются одностраничными (SPA) и мобильными приложениями. Эти приложения предоставляют дополнительную функциональность по сравнению с минимальными API за счет некоторой производительности и удобства.

В этой книге мы рассмотрим каждый из этих типов приложений, но в первой части сосредоточимся на минимальных API, поэтому в разделе 3.2.1 начнем с рассмотрения самого простого приложения ASP.NET Core, которое можно создать.

3.2.1 Использование шаблона

В этом разделе мы будем использовать шаблон для создания нашего первого приложения в ASP.NET Core. Использование шаблона может помочь вам быстро начать работу с приложением, осуществляя автоматическую настройку многих основных частей. И Visual Studio, и интерфейс командной строки .NET поставляются с определенным набором стандартных шаблонов для создания веб-приложений, консольных приложений и библиотек классов.

ПОДСКАЗКА В .NET *проект* – это единица развертывания, которая будет компилироваться, например, в файл .dll или исполняемый файл. Каждое отдельное приложение – это отдельный проект. В одном *решении* можно осуществлять сборку и разработку сразу нескольких проектов.

Чтобы создать свое первое веб-приложение, откройте Visual Studio и выполните следующие действия:

- 1 выберите **Create a New Project** (Создать новый проект) на экране-заставке или щелкните **File > New > Project** (Файл > Создать > Проект) на главном экране Visual Studio;
- 2 в списке шаблонов выберите **ASP.NET Core Empty**; выберите шаблон языка C#, как показано на рис. 3.2; а затем выберите **Next** (Далее);

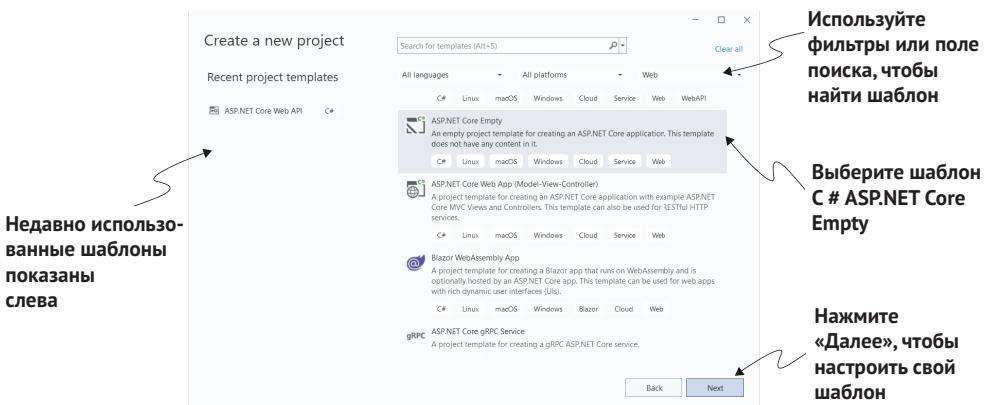


Рис. 3.2 Диалоговое окно нового проекта. Выберите шаблон веб-приложения ASP.NET Core из списка справа. Когда вы создадите новый проект, то сможете выбрать то, что вам нужно, из списка недавних шаблонов слева

- 3 на следующем экране введите имя проекта, расположение и имя решения и нажмите **Create** (Создать), как показано на рис. 3.3. Например, используйте *WebApplication1* в качестве имени проекта и решения;

Убедитесь, что выбран шаблон ASP.NET Core Empty

Введите имя и местоположение для вашего проекта и решения

Нажмите «Создать», чтобы перейти к дополнительным параметрам

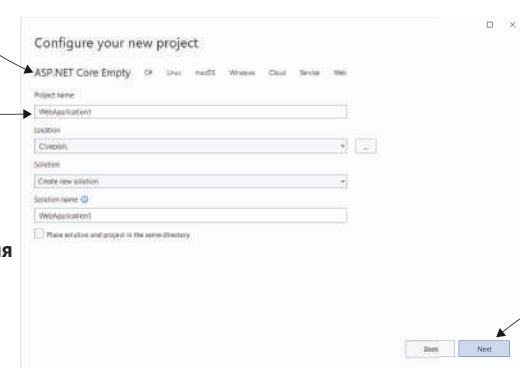


Рис. 3.3 Диалоговое окно **Configure Your New Project**. Введите имя проекта, расположение и имя решения и нажмите **Create**

- 4 на следующем экране (рис. 3.4) выполните следующие действия:
 - a выберите .NET 7.0. Если этот параметр недоступен, убедитесь, что у вас установлена .NET 7. Подробную информацию о настройке среды см. в приложении А;
 - b убедитесь, что установлен флажок **Configure for HTTPS** (Настроить для HTTPS);
 - c убедитесь, что флажок **Enable Docker Support** (Включить поддержку Docker) не установлен;
 - d убедитесь, что флажок **Do not use top-level statements** (Не использовать инструкции верхнего уровня) не установлен (я объясню инструкции верхнего уровня в разделе 3.6);
 - e выберите **Create** (Создать);

Убедитесь, что выбрана .NET 7.0

Убедитесь, что параметр «HTTPS» отмечен, а параметр «Включить поддержку Docker» нет

Нажмите «Создать», чтобы сгенерировать приложение из выбранного шаблона

Убедитесь, что флажок «Не использовать операторы верхнего уровня» снят

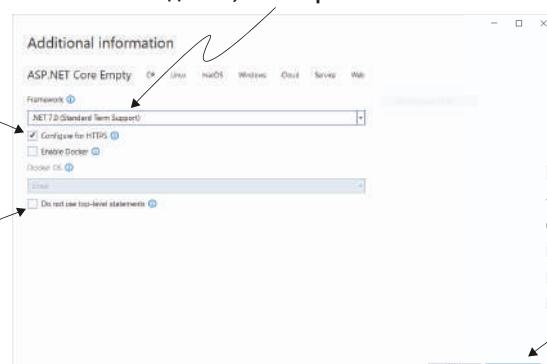


Рис. 3.4 Диалоговое окно «Дополнительная информация» следует за диалоговым окном «Настройка нового проекта» и позволяет настроить шаблон, который будет генерировать наше приложение. В этом стартовом проекте мы создадим пустое приложение .NET 7, использующее операторы верхнего уровня

- 5 подождите, пока Visual Studio создаст приложение из шаблона. Когда Visual Studio завершит работу, перед вами откроется вводная страница об ASP.NET Core, и вы сможете увидеть, что Visual Studio создала и добавила несколько файлов в ваш проект, как показано на рис. 3.5.

**При первом
создании
проекта
отображается
вводная
страница**

**Обозрева-
тель
решений
показыва-
ет файлы
в вашем
проекте**

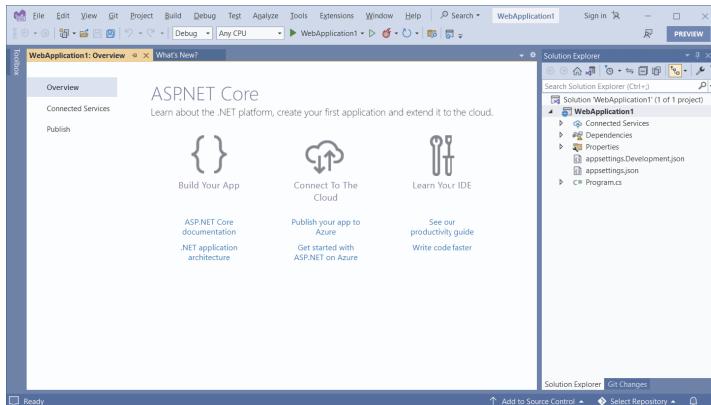


Рис. 3.5 Visual Studio после создания нового приложения ASP.NET Core из шаблона. Обозреватель решений показывает ваш только что созданный проект. На вводной странице есть полезные ссылки для ознакомления с ASP.NET Core

Если вы не используете Visual Studio, то можно создать аналогичное приложение с помощью интерфейса командной строки .NET. Создайте папку для своего нового проекта. Откройте командную строку PowerShell или cmd в папке (в Windows) или сеанс терминала (в Linux или macOS) и выполните команды из этого листинга:

Листинг 3.1. Создание нового приложения минимального API с помощью интерфейса командной строки .NET

```
dotnet new sln -n WebApplication1 ← | Создаем файл решения с именем  
dotnet new web -o WebApplication1   | WebApplication1 в текущей папке  
dotnet sln add WebApplication1      | Создаем пустой проект  
→ Добавляем новый проект в файл решения  
    во вложенной папке WebApplication1
```

ПРИМЕЧАНИЕ Visual Studio использует концепцию решения для работы с несколькими проектами. Пример решения состоит из одного проекта, который указан в .sln-файле. Если вы используете шаблон CLI для создания проекта, то у вас не будет .sln-файла, если вы не создадите его явно, используя дополнительные шаблоны интерфейса командной строки .NET (листинг 3.1).

Независимо от того, используете вы Visual Studio или интерфейс командной строки .NET, теперь у вас есть основные файлы, необходимые для создания и запуска вашего первого приложения ASP.NET Core.

3.2.2 Сборка приложения

На данный момент у нас есть основная часть кода, необходимого для запуска приложения, но осталось еще два шага. Во-первых, необходимо убедиться, что все зависимости, используемые нашим проектом, копируются в локальный каталог, а во-вторых, нужно скомпилировать приложение, чтобы можно было его запустить. Первый шаг не является строго обязательным, поскольку и Visual Studio, и интерфейс командной строки .NET автоматически восстанавливают пакеты при первом создании проекта, но будет полезно знать, как здесь все работает. В более ранних версиях, до 2.0, нужно было вручную восстанавливать пакеты с помощью команды `dotnet restore`. Можно скомпилировать приложение, выбрав **Build > Build Solution** (Сборка > Собрать решение), используя сочетание клавиш `Ctrl+Shift+B`, или выполнить команду `dotnet build` из командной строки. При выполнении сборки из Visual Studio ход сборки отображается в окне вывода, и если все в порядке, то ваше приложение будет скомпилировано и готово к запуску. Консольную команду `dotnet build` также можно выполнить из консоли диспетчера пакетов в Visual Studio.

ПОДСКАЗКА Visual Studio и инструменты интерфейса командной строки .NET автоматически выполняют сборку вашего приложения при его запуске, если обнаружат, что файл был изменен, поэтому обычно не нужно явно выполнять этот шаг самостоятельно.

Пакеты NuGet и интерфейс командной строки .NET

Одним из основополагающих компонентов кросс-платформенной разработки на .NET 5.0 является интерфейс командной строки (CLI) .NET. Он предоставляет несколько базовых команд для создания, сборки и запуска приложений .NET 5.0. Visual Studio фактически вызывает их автоматически, но их также можно вызывать напрямую из командной строки, если вы используете другой редактор. Наиболее распространенные команды, используемые во время разработки:

- `dotnet restore;`
- `dotnet build;`
- `dotnet run.`

Каждую из них следует запускать внутри папки проекта, и она будет выполнена только для этого проекта. Если не указано иное, это относится ко всем командам .NET CLI.

Большинство приложений ASP.NET Core зависят от внешних библиотек, управление которыми осуществляется с помощью диспетчера пакетов NuGet. Эти зависимости перечислены в проекте, но файлы самих библиотек не включены. Прежде чем вы сможете собрать и запустить свое приложение, необходимо убедиться, что в папке вашего проекта присутствуют локальные копии каждой зависимости. Первая команда, `dotnet restore`, обеспечивает копирование зависимостей NuGet вашего приложения в папку проекта.

Зависимости проектов ASP.NET Core перечислены в файле проекта .csproj. Это XML-файл, в котором каждая зависимость перечисляется как узел PackageReference. При выполнении команды dotnet restore она использует этот файл, чтобы определить, какие пакеты NuGet нужно загружать и копировать в папку проекта. Любые перечисленные зависимости доступны для использования в вашем приложении.

Процесс восстановления пакетов обычно происходит неявно при сборке или запуске приложения, но иногда может быть полезно запустить его явно, например в конвейерах сборки с непрерывной интеграцией.

Команда dotnet restore извлекает и восстанавливает все указанные пакеты NuGet

Команда dotnet build по умолчанию выполняет восстановление и компиляцию

Команда dotnet run по умолчанию восстанавливает, собирает и затем запускает приложение



Можно пропустить предыдущие шаги с помощью флагов -no-restore и -no-build

Команда dotnet build неявно запускает команду dotnet restore. Аналогично команда dotnet run запускает команды dotnet build и dotnet restore. Если вы не хотите автоматически выполнять предыдущие шаги, то можно использовать флаги --no-restore и --no-build, например: dotnet build --no-restore.

Скомпилировать приложение можно, используя команду dotnet build. Приложение будет проверено на наличие ошибок, и если проблем нет, в результате получится приложение, которое можно запустить с помощью команды dotnet run.

Каждая команда содержит ряд параметров, которые могут изменять ее поведение. Чтобы увидеть полный список доступных команд, выполните dotnet -help

Чтобы увидеть параметры, доступные для конкретной команды, например new, выполните

```
dotnet new -help
```

3.3 Запуск веб-приложения

Теперь можно запустить наше первое приложение, и сделать это можно несколькими способами. В Visual Studio можно щелкнуть по зеленой стрелке на панели инструментов рядом с IIS Express или нажать клавишу F5. Visual Studio автоматически откроет для вас окно веб-браузера с соответствующим URL-адресом, и через секунду или две вы увидите ответ Hello world!, как показано на рис. 3.6.

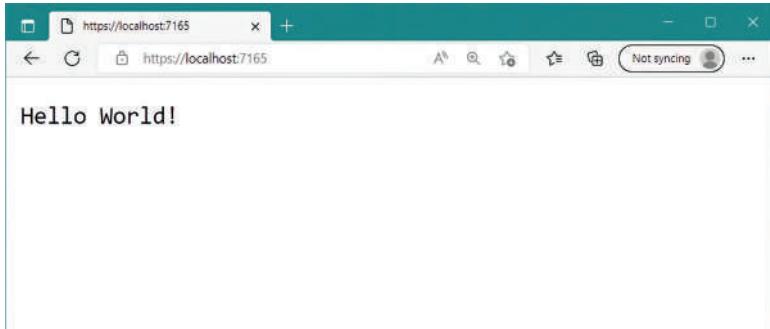


Рис. 3.6 Выходные данные нашего нового приложения ASP.NET Core. Шаблон выбирает случайный порт для использования в качестве URL-адреса приложения, который будет автоматически открыт в браузере при запуске из Visual Studio

Кроме того, можно запустить приложение из командной строки с помощью инструментов интерфейса командной строки .NET, используя команду `dotnet run`, и открыть URL-адрес в веб-браузере вручную, используя адрес, указанный в командной строке.

В зависимости от того, создавали ли вы свое приложение с помощью Visual Studio, вы можете увидеть URL-адрес `http://` или `https://`.

СОВЕТ При первом запуске приложения из Visual Studio вам будет предложено установить сертификат разработчика. Это нужно для того, чтобы ваш браузер не отображал предупреждения о недействительном сертификате HTTPS¹. Подробнее о сертификатах HTTPS см. главу 28.

В нашем учебном примере у приложения имеется единственная конечная точка, которая возвращает ответ в виде обычного текста, когда вы зараживаете путь `/`, как показано на рис. 3.6. С этим простым приложением больше ничего нельзя сделать, поэтому взглянем на код!

3.4 Разбираемся с шаблоном проекта

Создание приложения на основе шаблона имеет свои плюсы и минусы, особенно если вы делаете это в первый раз. С одной стороны, вы можете быстро запустить приложение, практически никакого участия с вашей стороны при этом не требуется. С другой стороны, количество файлов иногда может быть просто огромным, и вы будете ломать голову, пытаясь понять, с чего начать. Базовый шаблон веб-приложения не содержит много файлов и папок, как показано на рис. 3.7, я пробегусь по основным, чтобы вы сориентировались.

¹ Вы можете установить сертификат разработки в Windows и macOS. Шаги по настройке доверенных сертификатов в Linux ищите в инструкциях к вашему дистрибутиву. Не все браузеры (например, Mozilla Firefox) используют хранилище сертификатов, поэтому следуйте инструкциям браузера по добавлению сертификата в список доверенных. Если у вас все еще есть проблемы, см. советы по устранению неполадок – <http://mng.bz/o1pr>.

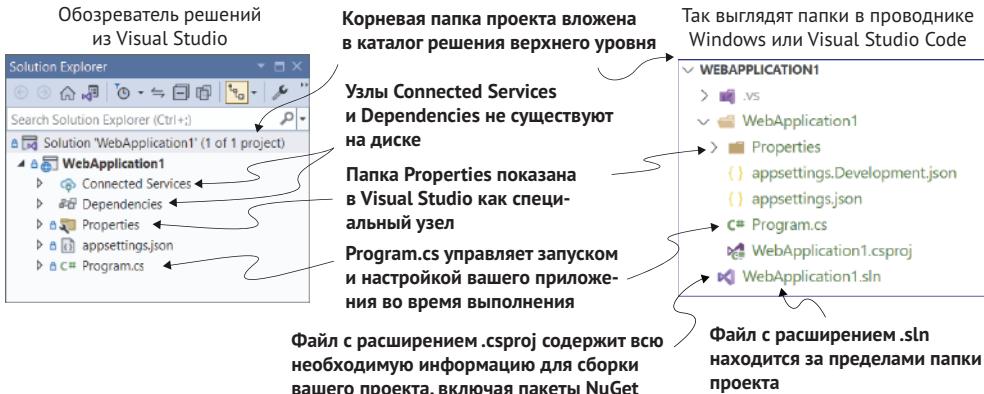


Рис. 3.7 Обозреватель решений и папка на диске для нового приложения ASP.NET Core. Обозреватель решений также отображает узлы Connected Services и Dependencies, в которых перечислены NuGet и другие зависимости, хотя самих папок на диске нет

Первое, на что следует обратить внимание, – это то, что основной проект, WebApplication1, вложен в корневую папку и само решение носит то же имя – WebApplication1. В корневой папке вы также найдете файл решения (.sln) для использования в Visual Studio, хотя он скрыт в окне обозревателя решений Visual Studio.

В папке решения вы найдете папку проекта, в которой, в свою очередь, содержится самый важный файл – WebApplication1.csproj. В этом файле описано, как собрать проект, и перечислены все необходимые дополнительные пакеты NuGet. Visual Studio не отображает файл .csproj явно, но его можно отредактировать, если дважды щелкнуть имя проекта в обозревателе решений или щелкнуть правой кнопкой мыши и выбрать **Properties** (Свойства) в контекстном меню. Мы подробнее рассмотрим этот файл проекта в следующем разделе.

В папке нашего проекта есть вложенная папка Properties, содержащая один файл: launchSettings.json. Этот файл управляет тем, как Visual Studio будет запускать и отлаживать приложение. Visual Studio отображает файл как специальный узел в обозревателе решений (не в алфавитном порядке) в верхней части проекта. У вас есть еще два специальных узла в проекте, **Dependencies** (Зависимости) и **Connected Services** (Подключенные службы), но у них нет соответствующих папок на диске. Вместо этого они показывают коллекцию всех зависимостей, например пакеты NuGet и удаленные службы, от которых зависит проект.

В корне папки вашего проекта вы найдете два файла JSON: appsettings.json и appsettings.Development.json. Эти файлы содержат параметры конфигурации, которые используются во время выполнения для управления поведением вашего приложения.

Наконец, Visual Studio отображает в папке проекта один файл C#: Program.cs. В разделе 3.6 вы увидите, как этот файл настраивает и запускает приложение.

3.5 Файл проекта .csproj: объявление зависимостей

Файл .csproj представляет собой файл проекта .NET-приложения и содержит данные, необходимые инструментам .NET для сборки проекта. Он определяет тип создаваемого проекта (веб-приложение, консольное приложение или библиотека), целевую платформу (.NET Core 3.1, .NET 7 и т. д.), а также то, от каких пакетов NuGet зависит проект.

Файл проекта был основой приложений .NET, но в ASP.NET Core он претерпел некоторые изменения, чтобы его было проще читать и редактировать. Эти изменения включают в себя:

- *отсутствие глобально уникальных идентификаторов* – раньше эти идентификаторы использовались для самых разных вещей, но теперь они почти не встречаются в файле проекта;
- *неявное включение файла* – прежде каждый файл в проекте должен был быть указан в файле .csproj, чтобы его можно было включить в сборку. Теперь файлы включаются в проект автоматически;
- *отсутствие путей к файлам .dll пакета NuGet* – раньше приходилось указывать путь к файлам .dll, содержащимся в пакетах NuGet в .csproj, а также перечислять зависимости в файле packages.config. Теперь можно ссылаться на пакет NuGet напрямую в файле .csproj и путь на диске указывать не нужно.

Все эти изменения делают файл проекта намного более компактным, чем тот, который мы привыкли видеть в предыдущих проектах .NET. В следующем листинге показан весь файл .csproj нашего приложения, используемого в качестве примера.

Листинг 3.2 Файл проекта .csproj, показывающий набор средств разработки, целевой фреймворк и ссылки

```
<Project Sdk="Microsoft.NET.Sdk.Web"> <!-- Атрибут SDK указывает тип проекта, который вы создаете -->
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework> <!-- TargetFramework – это платформа, на которой вы будете работать, в данном случае .NET 7 -->
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings> <!-- Включает функцию C# 10 «неявные операторы using» -->
  </PropertyGroup>
</Project>
```

Включает функцию C# 8 «ссылочные типы, допускающие значение NULL»

В случае с простыми приложениями вам, вероятно, не потребуется сильно менять файл проекта. Атрибут Sdk в элементе Project включает настройки по умолчанию, которые описывают, как собрать проект, тогда как элемент TargetFramework описывает фреймворк, на котором будет запускаться приложение. Для проектов .NET Core 6.0 это будет значение net6.0; если вы используете .NET 7, то это будет net7.0. Вы также можете включать и отключать различные функциональные возможности ком-

пилота, например ссылочные типы, допускающие значение NULL, в C# 8 или неявные директивы using в C# 10¹.

COBET Благодаря новому стилю csproj пользователи Visual Studio могут дважды щелкнуть по проекту в обозревателе решений, чтобы отредактировать файл .csproj, не выгружая проект.

Наиболее частые изменения, которые вы будете вносить в файл проекта, – это добавление дополнительных пакетов NuGet с помощью элемента PackageReference. По умолчанию в приложении нет ссылок на пакеты NuGet.

Использование пакетов NuGet в проекте

Несмотря на то что все приложения в чем-то уникальны, у них есть общие черты. Например, большинству приложений требуется доступ к базе данных или обработка JSON и XML. Вместо того чтобы заново писать этот код в каждом проекте, следует использовать существующие готовые библиотеки.

NuGet – это диспетчер пакетов библиотек для .NET, где библиотеки упакованы в пакеты NuGet и выложены на сайте <https://nuget.org>. Их можно использовать в своем проекте, указав уникальное имя пакета в файле .csproj. Таким образом пространство имен и классы пакета станут доступны в ваших файлах кода. Вы можете публиковать (и размещать) пакеты NuGet в репозиториях, отличных от тех, что находятся на сайте <https://nuget.org>, – см. <https://learn.microsoft.com/en-us/nuget/> для получения подробной информации.

Вы можете добавить ссылку на NuGet в свой проект, выполнив команду dotnet add package<packagename> из папки проекта. После этого в файл проекта будет добавлен узел, и пакет NuGet будет восстановлен. Например, чтобы установить популярную библиотеку Newtonsoft.Json, нужно выполнить команду

```
dotnet add package Newtonsoft.Json
```

После этого в файл проекта будет добавлена ссылка на последнюю версию библиотеки, как показано ниже, а пространство имен Newtonsoft.Json станет доступным в файлах исходного кода.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
  </ItemGroup>
</Project>
```

¹ О новых возможностях C#, включенных в .NET 7 и C# 11, можно прочитать на странице <http://mng.bz/nWMg>.

Если вы используете Visual Studio, то можно управлять пакетами с помощью диспетчера пакетов NuGet, щелкнув правой кнопкой мыши на имени решения или проекта и выбрав пункт Manage NuGet Packages.

Интересно отметить, что официально согласованного произношения слова NuGet не существует. Можно спокойно использовать популярные варианты типа «нугет» или «наггет», а если вы любитель гламура, тогда – «ну-джей»!

Упрощенный формат файла проекта намного легче редактировать вручную, чем предыдущие версии, и это замечательно, если вы занимаетесь кросс-платформенной разработкой. Но если вы применяете Visual Studio, то это, скорее всего, не ваш путь. Вы по-прежнему можете использовать графический интерфейс, чтобы добавлять ссылки на проекты, исключать файлы, управлять пакетами NuGet и т. д. Visual Studio сам обновит файл проекта, как всегда это делал.

СОВЕТ Для получения дополнительных сведений об изменениях в формате csproj см. документацию на странице <http://mng.bz/vnZL>.

Файл проекта определяет все, что необходимо Visual Studio и интерфейсу командной строки .NET для сборки вашего приложения. Все, кроме кода! В следующем разделе мы рассмотрим точку входа для нашего приложения ASP.NET Core – класс Program.cs.

3.6 Файл Program.cs: определение приложения

Все приложения ASP.NET Core запускаются так же, как и приложения .NET Console. Начиная с .NET 6 это обычно означает программу, написанную с использованием *инструкций верхнего уровня*, в которой код запуска приложения записывается непосредственно в файле, а не в функции static void Main.

Инструкции верхнего уровня

До C# 9 каждая программа .NET должна была включать функцию static void Main (она также могла возвращать int, Task или Task<int>), обычно объявляемую в классе Program. Эта функция, которая должна существовать, определяет точку входа для вашей программы. Этот код запускается при запуске приложения, как в данном примере:

```
using System;
namespace MyApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

С помощью инструкций верхнего уровня можно записать тело этого метода непосредственно в файл, и компилятор сгенерирует для вас метод Main.

В сочетании с такими возможностями C# 10, как неявные директивы using, это значительно упрощает код точки входа приложения:

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

При использовании явной функции Main можно получить доступ к аргументам командной строки, предоставленным при запуске приложения, с использованием параметра args. В инструкциях верхнего уровня переменная args также доступна в виде string[], даже если не объявлена явно. Можно вывести каждый предоставленный аргумент, используя следующий код:

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

В .NET 7 все шаблоны по умолчанию используют инструкции верхнего уровня, а я использую их на протяжении всей книги. Большинство шаблонов включают возможность применения явной функции Main, если вы предпочитаете (используя параметр --use-program-main, если вы работаете с интерфейсом командной строки). Для получения дополнительной информации об инструкциях верхнего уровня и их ограничениях см. <http://mng.bz/4DZa>. Если позже вы решите сменить подход, вы всегда сможете добавить или удалить функцию Main вручную по мере необходимости.

В приложениях .NET 7 ASP.NET Core инструкции верхнего уровня создают и запускают экземпляр WebApplication, как продемонстрировано в следующем листинге, где показан файл Program.cs по умолчанию. WebApplication – это ядро нашего приложения ASP.NET Core, содержащее конфигурацию приложения и сервер Kestrel, который прослушивает запросы и отправляет ответы.

Листинг 3.3. Файл Program.cs по умолчанию, который настраивает и запускает веб-приложение

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args); ←
WebApplication app = builder.Build(); ←
    ↗ Создаем
    ↗ WebApplicationBuilder с по-
    ↗ мощью метода CreateBuilder

    ↗ app.MapGet("/", () => "Hello World!"); ←
        ↗ Создаем и возвращаем
        ↗ экземпляр WebApplication
        ↗ из WebApplicationBuilder

    ↗ app.Run(); ←
        ↗ Запускаем веб-приложение,
        ↗ чтобы начать прослушивать за-
        ↗ просы и генерировать ответы

    ↗ Определяем конечную точку
    ↗ нашего приложения, которая
    ↗ возвращает фразу Hello World!,
    ↗ когда вызывается путь «/»
```

Эти четыре строки содержат весь код инициализации, необходимый для создания веб-сервера и начала прослушивания запросов. В нем используется `WebApplicationBuilder`, созданный вызовом `CreateBuilder`, чтобы определить, как настроено веб-приложение, прежде чем создавать экземпляр `WebApplication`, используя вызов метода `Build()`.

ПРИМЕЧАНИЕ Паттерн «Строитель» для конфигурирования сложного объекта широко используется в ASP.NET Core. Это полезный метод, позволяющий пользователям настраивать объект, откладывая его создание до завершения всех настроек, и один из паттернов, описанных в книге «Банды четырех» – «Паттерны проектирования: элементы объектно ориентированного программного обеспечения многократного использования» Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса (Addison Wesley, 1994).

В этом простом приложении мы не вносим никаких изменений в `WebApplicationBuilder` перед вызовом `Build()`, но по умолчанию `WebApplicationBuilder` настраивает много разных вещей, в том числе:

- *конфигурирование* – ваше приложение загружает значения из файлов JSON и переменных окружения, которые можно использовать для управления поведением приложения во время выполнения, например для загрузки строк подключения для базы данных. Подробнее о системе конфигурации вы узнаете в главе 10;
- *журналирование* – ASP.NET Core включает расширяемую систему журналирования для наблюдения и отладки. Я подробно расскажу о системе журналирования в главе 26;
- *сервисы* – любые классы, от которых зависит ваше приложение для обеспечения функциональности (как те, что используются фреймворком, так и конкретные классы для вашего приложения), должны быть зарегистрированы, чтобы их можно было правильно создать во время выполнения. `WebApplicationBuilder` настраивает минимальный набор сервисов, необходимых для приложения ASP.NET Core. В главах 8 и 9 подробно рассматривается конфигурация сервисов;
- *размещение* – по умолчанию ASP.NET Core использует веб-сервер Kestrel для обработки запросов.

После настройки `WebApplicationBuilder` мы вызываем метод `Build()` для создания экземпляра `WebApplication`. В этом экземпляре мы определяем, как наше приложение обрабатывает запросы и отвечает на них, используя два строительных блока:

- *промежуточное ПО* – эти небольшие компоненты выполняются последовательно, когда приложение получает HTTP-запрос. Они могут выполнять целый ряд задач, например журналирование, идентификация текущего пользователя для запроса, обслуживание статических файлов и обработка ошибок. Мы подробно рассмотрим конвейер промежуточного ПО в главе 4;
- *конечные точки* – определяют, как должен генерироваться ответ на конкретный запрос URL-адреса в приложении.

Для приложения из листинга 3.3 мы не добавляли никакого промежуточного ПО, а определили одну конечную точку с помощью вызова `MapGet`:

```
app.MapGet("/", () => "Hello World!");
```

Мы используем функцию `MapGet`, чтобы определить, как обрабатывать запрос, использующий *HTTP-команду GET*. Существуют и другие функции `Map*` для других HTTP-команд, например `MapPost`.

ОПРЕДЕЛЕНИЕ Каждый HTTP-запрос включает в себя *метод*, указывающий «тип» запроса. При просмотре сайта по умолчанию используется метод `GET`, который *извлекает* ресурс с сервера, чтобы его можно было просмотреть. Второй наиболее распространенный метод – `POST`, который используется для отправки данных на сервер, например при заполнении формы.

Первый аргумент, передаваемый `MapGet`, определяет, на какой URL-адрес следует ответить, а второй аргумент определяет, как сгенерировать ответ в виде делегата, возвращающего строку. В данном простом случае аргументы говорят: «При выполнении делается запрос к пути `/` с использованием метода `GET`, ответьте текстовым значением `Hello World!`».

ОПРЕДЕЛЕНИЕ Путь – это оставшаяся часть URL-адреса запроса после удаления домена. Для запроса к www.example.org/account/manage путь – это `/account/manage`.

Пока вы конфигурируете `WebApplication` и `WebApplicationBuilder`, приложение не обрабатывает HTTP-запросы. Только после вызова `Run()` HTTP-сервер начинает прослушивать запросы. На данном этапе наше приложение полностью работоспособно и может ответить на первый запрос от удаленного браузера.

ПРИМЕЧАНИЕ Классы `WebApplication` и `WebApplicationBuilder` впервые появились в .NET 6. Код инициализации в предыдущих версиях ASP.NET Core был более подробным, но давал больше контроля над поведением приложения. Обычно конфигурация разделялась на два класса – `Program` и `Startup` – и использовались разные типы конфигурации – `IHostBuilder` и `IHost`, у которых меньше значений по умолчанию, чем у `WebApplication`. В главе 30 некоторые из этих различий описаны более подробно и показано, как настроить приложение, используя универсальный `IHost` вместо `WebApplication`.

До сих пор в этой главе мы рассматривали самое простое базовое приложение ASP.NET, которое можно создать: приложение минимального API Hello World. В оставшейся части этой главы мы будем использовать его, чтобы представить некоторые фундаментальные концепции ASP.NET Core.

3.7 Добавляем функциональность в приложение

Настройка приложения, которую мы видели в `Program.cs`, состоит всего из четырех строк кода, но все равно показывает общую *структуру*

типичной точки входа приложения ASP.NET Core, которая обычно состоит из шести этапов:

- 1 создание экземпляра `WebApplicationBuilder`;
- 2 регистрация необходимых служб и конфигурации с помощью `WebApplicationBuilder`;
- 3 вызов метода `Build()` в экземпляре компоновщика, чтобы создать экземпляр `WebApplication`;
- 4 добавление промежуточного ПО в веб-приложение, чтобы создать конвейер;
- 5 сопоставление конечных точек приложения;
- 6 вызов метода `Run()` в веб-приложении, чтобы запустить сервер и обработать запросы.

Базовое приложение минимального API, показанное ранее в листинге 3.3, было достаточно простым, поэтому ему не требовалось этапы 2 и 4, но в остальном оно следовало этой последовательности в файле `Program.cs`. Следующий листинг расширяет приложение по умолчанию, добавляя дополнительные функциональные возможности, и при этом здесь используются все шесть этапов.

Листинг 3.4. Файл Program.cs для более сложного примера минимального API

```
using Microsoft.AspNetCore.HttpLogging;
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

Можно настроить функциональные возможности, добавив или настроив сервисы приложения

builder.Services.AddHttpLogging(opts =>
    opts.LoggingFields = HttpLoggingFields.RequestProperties);
Гарантирует, что журналы, добавленные компонентом журналирования HTTP-запросов, видны в выходных данных журналов

builder.Logging.AddFilter(
    "Microsoft.AspNetCore.HttpLogging", LogLevel.Information);

Можно добавить промежуточное ПО условно, в зависимости от среды выполнения

WebApplication app = builder.Build();
if (app.Environment.IsDevelopment()) <-->
{
    app.UseHttpLogging(); <-->
Компонент журналирования HTTP-запросов регистрирует каждый запрос к приложению в выходных данных журнала
}

app.MapGet("/", () => "Hello World!");
app.MapGet("/person", () => new Person("Andrew", "Lock")); <-->
Создаем новую конечную точку, которая возвращает объект C#, сериализованный как JSON

app.Run();

public record Person(string FirstName, string LastName);
```

Создаем тип записи

Приложение из листинга 3.4 настраивает две новые функции:

- при работе в окружении разработки сведения о каждом запросе регистрируются с помощью `HttpLoggingMiddleware`¹;
- создает новую конечную точку в `/person`, которая создает экземпляр записи C# с именем `Person` и сериализует ее в ответе как JSON.

Когда вы запускаете приложение и отправляете запросы через веб-браузер, вы видите подробную информацию о запросе, отображаемую в консоли, как показано на рис. 3.8. Если вы вызовете конечную точку `/person`, то увидите JSON-представление записи `Person`, которую вы создали в конечной точке.

ПРИМЕЧАНИЕ Вы можете просмотреть приложение только на том компьютере, на котором оно запущено в данный момент; наше приложение еще не доступно в интернете. В главе 27 вы узнаете, как опубликовать и развернуть свое приложение.

Конфигурирование служб, журналирования, промежуточного ПО и конечных точек имеет основополагающее значение для создания приложений ASP.NET Core, поэтому оставшаяся часть раздела 3.7 познакомит вас с каждой из этих концепций, чтобы у вас сформировалось представление о том, как они используются. Я не буду подробно их объяснять (этому у нас посвящена остальная часть книги!), но вы должны учитывать, как они вытекают друг из друга и какой вклад вносят в конфигурацию приложения в целом.

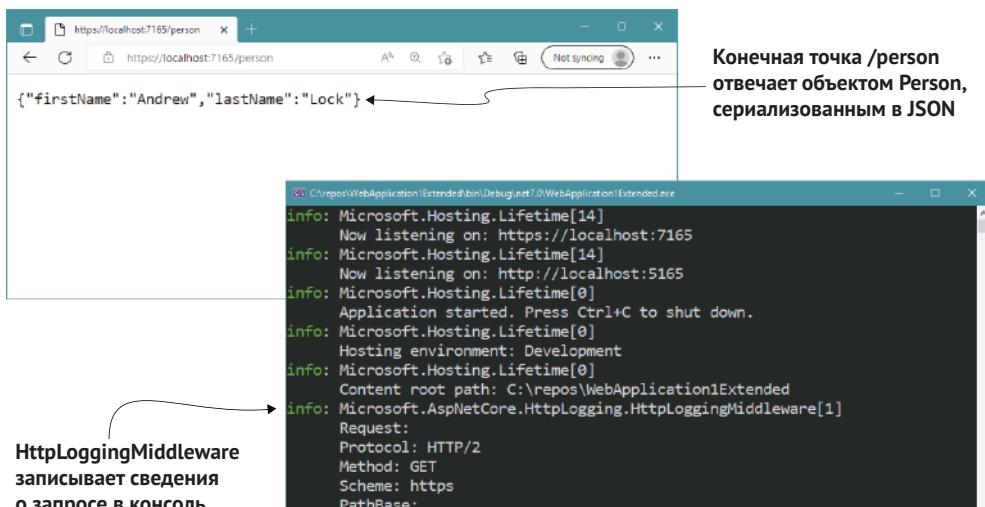


Рис. 3.8 Вызов конечной точки `/person` возвращает сериализованную в формате JSON версию экземпляра записи `Person`. Подробная информация о каждом запросе записывается в консоль с помощью `HttpLoggingMiddleware`

¹ Подробнее о журналировании HTTP-запросов можно прочитать в документации на странице <http://mng.bz/QPmw>.

3.7.1 Добавление и настройка сервисов

ASP.NET Core использует небольшие модульные компоненты для каждой отдельной функциональной возможности, что позволяет этим возможностям развиваться отдельно, при этом будучи слабо связанными с другими возможностями. Обычно это считается хорошей практикой проектирования. Обратная сторона данного подхода состоит в том, что он возлагает ответственность за правильное создание функциональной возможности на ее потребителя. В рамках нашего приложения эти модульные компоненты представлены в виде одного или нескольких *сервисов*, используемых приложением.

ОПРЕДЕЛЕНИЕ В контексте ASP.NET Core под словом *сервис* подразумевается любой класс, предоставляющий функциональные возможности приложению. Это могут быть классы, доступ к которым предоставляется библиотекой, или код, который вы написали для своего приложения.

Например, в приложении для онлайн-коммерции у вас может быть сервис `TaxCalculator`, который рассчитывает налог, причитающийся с определенного продукта, с учетом местоположения пользователя. Или же у вас может быть сервис `ShippingCostService`, рассчитывающий стоимость доставки к местоположению пользователя. Третий сервис, `OrderTotalCalculatorService`, может использовать оба этих сервиса для расчета общей стоимости, которую пользователь должен заплатить за заказ. Каждый сервис предоставляет небольшую часть независимых функциональных возможностей, но их можно объединить, чтобы создать законченное приложение. Этот принцип известен как *принцип единственной ответственности*.

ОПРЕДЕЛЕНИЕ *Принцип единственной ответственности* гласит, что каждый класс должен отвечать только за одну часть функциональности – его следует изменять лишь в случае изменения этой необходимой функциональности. Это один из пяти основных принципов объектно ориентированного проектирования, изложенных Робертом Мартином в книге «Быстрая разработка программ. Принципы, примеры, практика».

Сервису `OrderTotalCalculatorService` требуется доступ к экземпляру сервисов `ShippingCostService` и `TaxCalculator`. Использовать ключевое слово `new` и создавать экземпляр сервиса всякий раз, когда он вам понадобится, – примитивный подход к этой проблеме. К сожалению, это тесно привязывает ваш код к конкретной реализации, которую вы используете, и может полностью свести на нет все достоинства модульного подхода к проектированию. В некоторых случаях это может нарушить принцип единственной ответственности, заставляя вас выполнять код инициализации в дополнение к использованию созданного вами сервиса.

Одно из решений этой проблемы – переложить ее на кого-то другого. При написании сервиса можно объявить свои зависимости и позволить

другому классу разрешить их за вас. Тогда ваш сервис может сосредоточиться на функциональности, для которой он был разработан, вместо того чтобы пытаться понять, как собрать свои зависимости. Данный метод называется *внедрением зависимостей*, или *принципом инверсии управления* (IoC). Это хорошо известный и широко используемый паттерн проектирования. Обычно зависимости приложения регистрируются в «контейнере», который затем можно использовать для создания любого сервиса, что справедливо как для ваших собственных сервисов приложения, так и для сервисов фреймворка, используемых ASP.NET Core. Вы должны зарегистрировать сервис, прежде чем его можно будет использовать в приложении.

ПРИМЕЧАНИЕ Принцип инверсии зависимостей и IoC-контейнер, используемый в ASP.NET Core, будут подробно описаны в главах 8 и 9.

В приложении ASP.NET Core эта регистрация выполняется путем использования свойства `Services` класса `WebApplicationBuilder`. Всякий раз, когда вы используете новую функцию ASP.NET Core в своем приложении, вам нужно возвращаться в `Program.cs` и добавлять необходимые сервисы. Эта задача не всегда так сложна, как кажется: обычно для настройки приложений требуется всего одна-две строки кода.

В листинге 3.4 мы настроили дополнительный сервис для компонента журналирования HTTP-запросов, используя следующий код:

```
builder.Services.AddHttpLogging(opts =>
    opts.LoggingFields = HttpLoggingFields.RequestProperties);
```

Вызов метода `AddHttpLogging()` добавляет необходимые сервисы для компонента журналирования HTTP-запросов в IoC-контейнер и настраивает параметры, используемые промежуточным ПО для отображения. `AddHttpLogging` не отображается непосредственно в свойстве `Services`; это метод расширения, который обеспечивает удобный способ инкапсуляции всего необходимого кода, чтобы настроить журналирование HTTP-запросов. Этот паттерн инкапсуляции настройки методов расширения распространен в ASP.NET Core.

Помимо регистрации сервисов, связанных с фреймворком, свойство `Services` позволяет регистрировать любые пользовательские сервисы, имеющиеся в вашем приложении, например `TaxCalculator`, который мы обсуждали ранее. Свойство `Services` – это `IServiceCollection`, представляющий собой список всех известных сервисов, которые потребуется использовать вашему приложению. Добавляя новый сервис, вы гарантируете, что всякий раз, когда класс объявляет зависимость от вашего сервиса, IoC-контейнер будет знать, как его предоставить.

Помимо конфигурирования сервисов, `WebApplicationBuilder` позволяет настроить дополнительную сквозную функциональность, например журналирование. В листинге 3.4 я показал, как добавить фильтр журналирования, чтобы гарантировать, что журналы, созданные `HttpLoggingMiddleware`, будут записываться в консоль:

```
builder.Logging.AddFilter(  
    "Microsoft.AspNetCore.HttpLogging", LogLevel.Information);
```

Данная строка кода гарантирует, что журналы уровня серьезности Information или выше, созданные в пространстве имен Microsoft.AspNetCore.HttpLogging, будут включены в вывод журнала.

ПРИМЕЧАНИЕ Для удобства я показываю настройку фильтров журналов в коде, но это не является идиоматическим подходом к настройке фильтров в ASP.NET Core. Обычно мы управляем отображением уровней, добавляя значения в файл appsettings.json, как показано в исходном коде к этой главе. Подробнее о журналировании и фильтрации журналов вы узнаете в главе 26.

После вызова метода `Build()` для экземпляра `WebApplicationBuilder` вы больше не сможете регистрировать сервисы или изменять конфигурацию журналирования; сервисы, определенные для экземпляра `WebApplication`, незыблемы. Следующий шаг – определить, как наше приложение отвечает на HTTP-запросы.

3.7.2 Определение способа обработки запросов с помощью промежуточного ПО и конечных точек

После регистрации сервисов в IoC-контейнере в `WebApplicationBuilder` и выполнения дальнейших настроек мы создаем экземпляр `WebApplication`, с которым можно делать три основные вещи:

- добавлять промежуточное ПО в конвейер;
- сопоставлять конечные точки, которые генерируют ответ на запрос;
- запускать приложение, вызвав метод `Run()`.

Как я описывал ранее, промежуточное ПО состоит из небольших компонентов, которые выполняются последовательно, когда приложение получает HTTP-запрос. Они могут выполнять множество функций, среди которых журналирование, идентификация текущего пользователя для запроса, обслуживание статических файлов и обработка ошибок. Промежуточное ПО обычно добавляется в `WebApplication` путем вызова методов расширения `Use*`. В листинге 3.4 я показал пример условного добавления `HttpLogging-Middleware` в конвейер промежуточного ПО путем вызова метода `UseHttpLogging()`:

```
if (app.Environment.IsDevelopment())  
{  
    app.UseHttpLogging();  
}
```

В этом примере мы добавили в конвейер только одну часть промежуточного ПО, но когда вы добавляете несколько частей, важен порядок вызовов методов `Use*`: порядок, в котором они добавляются в конструктор, – это порядок, в котором они будут выполняться в финальном конвейере. Промежуточное ПО может использовать только объекты, созданные

предыдущим промежуточным ПО в конвейере; оно не может получить доступ к объектам, созданным более поздним промежуточным ПО.

ВНИМАНИЕ Важно учитывать порядок промежуточного ПО при добавлении его в конвейер, поскольку оно может использовать только объекты, созданные ранее в конвейере.

Также следует отметить, что в листинге 3.4 используется свойство `WebApplication.Environment` (экземпляр `IWebHostEnvironment`), чтобы обеспечить другое поведение в окружении разработки. `HttpLoggingMiddleware` добавляется в конвейер только во время разработки; в промышленном окружении (или, скорее, когда для параметра `EnvironmentName` не задано значение `Development`) `HttpLoggingMiddleware` не будет добавлен.

ПРИМЕЧАНИЕ Вы узнаете об окружениях размещения и о том, как изменить текущее окружение, в главе 10.

`WebApplicationBuilder` создает объект `IWebHostEnvironment` и задает его в свойстве `Environment`. `IWebHostEnvironment` предоставляет несколько свойств, связанных с окружением:

- `ContentRootPath` – расположение рабочего каталога приложения, обычно это папка, в которой выполняется приложение;
- `WebRootPath` – расположение папки `wwwroot`, содержащей статические файлы;
- `EnvironmentName` – является текущее окружение окружением разработки или промышленным окружением.

К моменту создания экземпляра `WebApplication` значение для `IWebHostEnvironment` уже задано. Значение для `EnvironmentName` обычно задается извне с помощью переменной окружения при запуске приложения.

В листинге 3.4 в конвейер добавлена только одна часть промежуточного ПО, но `WebApplication` автоматически добавляет дополнительное промежуточное ПО, включая две наиболее важные и существенные части этого ПО в конвейере: компонент маршрутизации и компонент конечной точки. Компонент маршрутизации добавляется автоматически в начало конвейера перед добавлением любого дополнительного промежуточного ПО в файл `Program.cs` (т. е. перед `HttpLoggingMiddleware`). Компонент конечной точки добавляется в конец конвейера, после того как все остальное промежуточное ПО добавляется в `Program.cs`.

ПРИМЕЧАНИЕ По умолчанию `WebApplication` добавляет в конвейер еще несколько частей промежуточного ПО. Например, он автоматически добавляет компонент обработки ошибок, когда вы работаете в окружении разработки. Мы подробно обсудим некоторые из этих автоматически добавляемых компонентов в главе 4.

Вместе эта пара отвечает за интерпретацию запроса, чтобы определить, какую конечную точку вызывать, за чтение параметров за-

проса и за генерацию окончательного ответа. Для каждого запроса компонент *маршрутизации* использует URL-адрес запроса, чтобы определить, какую конечную точку вызвать. Затем остальная часть конвейера промежуточного ПО выполняется до тех пор, пока запрос не достигнет компонента конечной точки, после чего этот компонент выполняет конечную точку для генерации окончательного ответа.

Компоненты маршрутизации и конечных точек работают в tandemе, используя набор конечных точек, определенный для вашего приложения. В листинге 3.4 мы определили две конечные точки:

```
app.MapGet("/", () => "Hello World!");  
app.MapGet("/person", () => new Person("Andrew", "Lock"));
```

Вы уже видели конечную точку «Hello World!». При отправке запроса методом GET в / компонент маршрутизации выбирает конечную точку «Hello World!». Запрос идет дальше по конвейеру промежуточного ПО, пока не достигнет компонента конечной точки, который выполняет лямбда-выражение и возвращает строковое значение в теле ответа.

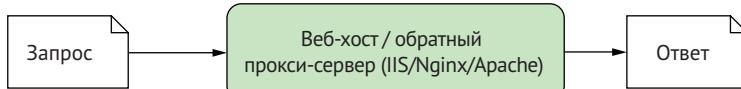
Другая конечная точка определяет лямбда-выражение для выполнения запросов методом GET к пути /person, но возвращает запись C# вместо строки. При возвращении объекта C# из конечной точки минимального API объект автоматически сериализуется в JSON и возвращается в теле ответа, как было показано на рис. 3.8. В главе 6 вы узнаете, как настроить этот ответ, а также возвращать другие типы ответов.

И вот оно. Мы завершили обзор своего первого приложения ASP.NET Core!

Прежде чем двигаться дальше, в последний раз посмотрим, как наше приложение обрабатывает запрос. На рис. 3.9 показан запрос к пути /person, обрабатываемый приложением, которое используется в качестве примера. Здесь вы уже все видели, поэтому процесс обработки запроса должен быть вам знаком. На рисунке показано, как запрос проходит через конвейер промежуточного ПО перед обработкой компонентом конечной точки. Конечная точка выполняет лямбда-метод и генерирует ответ в формате JSON, который передается обратно через промежуточное ПО на веб-сервер ASP.NET Core перед отправкой в браузер пользователя.

Это было довольно насыщенное путешествие, но теперь у вас есть неплохое представление о том, как настраивается все приложение и как оно обрабатывает запросы, используя минимальные API. В главе 4 мы более подробно рассмотрим конвейер промежуточного ПО, который есть во всех приложениях ASP.NET Core. Вы узнаете, как он устроен и как его можно использовать для расширения функциональности приложения и создания простых HTTP-сервисов.

1. Выполняется HTTP-запрос к URL/person



2. Запрос пересыпается IIS/Nginx/Apache в ASP.NET Core

3. ASP.NET Core (Kestrel) получает HTTP-запрос и передает его промежуточному ПО

4. Путь запроса /person маршрутизируется на конечную точку минимального API, поэтому запрос проходит через конвейер промежуточного ПО без изменений

5. Конечная точка минимального API обрабатывает запрос, возвращая экземпляр Person

8. Ответ HTTP, содержащий данные в формате JSON, отправляется в браузер

7. Ответ в формате JSON передается обратно через каждый компонент промежуточного ПО на веб-сервер ASP.NET Core

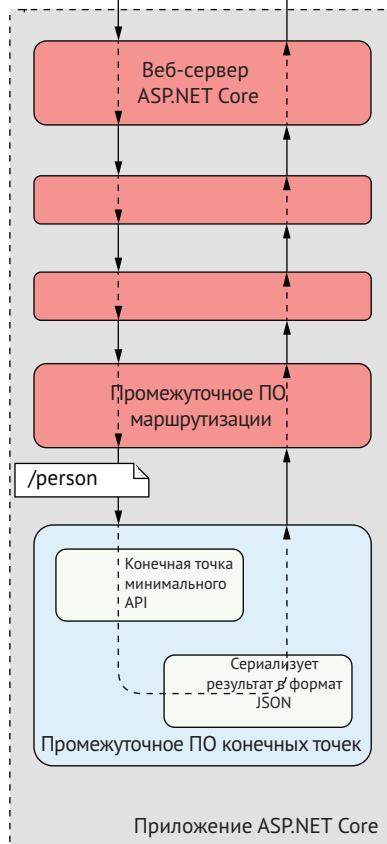


Рис. 3.9 Обзор запроса к URL-адресу /person для расширенного приложения минимального API ASP.NET Core. Компонент маршрутизации направляет запрос к правильному лямбда-методу. Конечная точка генерирует ответ в формате JSON, выполняя метод, и передает ответ обратно через конвейер промежуточного ПО в браузер

Резюме

- Файл .csproj содержит подробную информацию о том, как собрать проект, в том числе от каких пакетов NuGet он зависит. Visual Studio и .NET CLI используют этот файл для создания приложения;
- при восстановлении пакетов NuGet для приложения ASP.NET Core скачиваются все зависимости вашего проекта, чтобы приложение можно было собрать и запустить;
- файл Program.cs – это место, где вы определяете код, который запускается при запуске приложения. Можно создать `WebApplicationBuilder`, используя `WebApplication.CreateBuilder()` и вызывая методы построителя для создания приложения;
- все сервисы, как сервисы фреймворка, так и собственные сервисы приложений, должны быть зарегистрированы в `WebApplicationBuilder` с помощью свойства `Services`, чтобы к ним можно было получить доступ позже в приложении;
- после настройки сервисов мы вызываем метод `Build()` экземпляра `WebApplicationBuilder`, чтобы создать экземпляр `WebApplication`. Мы используем `WebApplication` для настройки конвейера промежуточного ПО приложения, регистрации конечных точек и запуска сервера, слушающего запросы;
- промежуточное ПО определяет, как приложение отвечает на запросы. Порядок регистрации промежуточного ПО определяет окончательный порядок конвейера промежуточного ПО для приложения;
- экземпляр `WebApplication` автоматически добавляет `RoutingMiddleware` в начало конвейера промежуточного ПО, а `EndpointMiddleware` – в качестве последнего компонента в конвейере;
- конечные точки определяют, как должен генерироваться ответ на данный запрос, и обычно привязаны к пути запроса. В случае с минимальными API для генерации ответа используется простая функция;
- можно запустить веб-сервер и начать принимать HTTP-запросы, вызвав метод `Run` экземпляра `WebApplication`.



Обработка ошибок с помощью конвейера промежуточного ПО

В этой главе:

- что такое промежуточное ПО;
- обслуживание статических файлов с использованием промежуточного ПО;
- добавление функциональности с помощью промежуточного ПО;
- объединение промежуточного ПО для формирования конвейера;
- обработка исключений и ошибок с помощью промежуточного ПО.

В предыдущей главе вы подробно ознакомились с полным приложением ASP.NET Core, чтобы увидеть, как объединяются компоненты для создания веб-приложения.

В этой главе мы сосредоточимся на одном небольшом подразделе: конвейере промежуточного программного обеспечения (*middleware pipeline*).

В ASP.NET Core *промежуточное ПО* – это классы или функции C#, которые обрабатывают HTTP-запрос или ответ. Они выстроены в цепочку, чтобы выходные данные одного компонента действовали как входные данные для следующего компонента, формируя конвейер.

Конвейер промежуточного ПО – одна из наиболее важных частей конфигурации для определения того, как ваше приложение ведет себя и реагирует на запросы. Понимание того, как создавать и скомпоновывать промежуточное ПО, является ключом к добавлению функциональности в ваши приложения.

В этой главе вы узнаете, что такое промежуточное ПО и как использовать его для создания конвейера. Вы увидите, как связать несколько компонентов промежуточного ПО воедино, где каждый компонент добавляет отдельную функциональность. Примеры, приведенные в данной главе, ограничиваются использованием существующих компонентов промежуточного ПО, показывая, как правильно расположить их для своего приложения. В главе 31 вы узнаете, как создавать собственные компоненты промежуточного ПО и включать их в конвейер.

Мы начнем с рассмотрения концепции промежуточного ПО, всего того, чего можно достичь с его помощью, и того, как компонент промежуточного ПО часто сопоставляется со «сквозной задачей». Это функции приложения, работающие на разных уровнях. Журналирование, обработка ошибок и безопасность – это классические сквозные задачи, необходимые множеству различных частей приложения. Поскольку все запросы проходят через конвейер промежуточного ПО, то это предпочтительное место для настройки и обработки данной функциональности.

В разделе 4.2 я объясню, как скомпоновать отдельные компоненты промежуточного ПО в конвейер. Мы начнем с малого, с веб-приложения, которое отображает только страницу приветствия. Далее вы узнаете, как создать простой сервер статических файлов, который возвращает запрашиваемые файлы из папки на диске.

Затем мы перейдем к более сложному конвейеру, содержащему несколько компонентов. Мы рассмотрим важность упорядочивания, и вы увидите, как обрабатываются запросы, когда конвейер содержит несколько компонентов.

В разделе 4.3 вы узнаете, как использовать промежуточное ПО для решения важного аспекта любого приложения: обработки ошибок. Ошибки – это реальность любого приложения, поэтому важно учитывать их при его создании. Помимо обеспечения того, чтобы ваше приложение не прерывалось при выбросе исключения или возникновении ошибки, важно, чтобы пользователи приложения были информированы о том, что пошло не так, в удобной для них форме.

Можно обрабатывать ошибки несколькими способами, но поскольку речь идет об одной из классических сквозных задач, промежуточное ПО хорошо подходит для обеспечения необходимой функциональности. В разделе 4.3 я покажу, как можно обрабатывать исключения и ошибки с помощью промежуточного ПО, предоставляемого компанией Microsoft. В частности, вы узнаете о трех различных компонентах:

- `DeveloperExceptionPageMiddleware` – обеспечивает быструю обратную связь об ошибках при создании приложения;
- `ExceptionHandlerMiddleware` – предоставляет удобную для пользователя общую страницу ошибок в промышленном окружении.

В этой главе вы не увидите, как создавать собственное промежуточное ПО, – но поймете, что можно многое достичь, используя компоненты, предоставляемые как часть ASP.NET Core. Разобравшись с тем, что такое конвейер промежуточного ПО и как он работает, легче понять, когда и почему требуется специальное промежуточное ПО. Приступим!

4.1 Что такое промежуточное ПО

Словосочетание *промежуточное ПО* используется в различных контекстах при разработке программного обеспечения и ИТ, однако оно не особо информативно – так что же такое промежуточное ПО?

В ASP.NET Core промежуточное ПО – это классы C#¹, которые могут обрабатывать HTTP-запрос или ответ. Оно может:

- обработать входящий HTTP-запрос путем создания HTTP-ответа;
- обработать входящий HTTP-запрос, изменить его и передать другой части промежуточного ПО;
- обработать исходящий HTTP-ответ, изменить его и передать либо другой части промежуточного ПО, либо веб-серверу ASP.NET Core.

Промежуточное ПО можно использовать в своих приложениях самыми разными способами. Например, компонент журналирования может отмечать, когда поступил запрос, и затем передавать его другому компоненту. Между тем компонент для изменения размера изображения может обнаружить входящий запрос на изображение с указанным размером, сгенерировать запрошенное изображение и отправить его обратно пользователю, не передавая его дальше.

Самый важный компонент промежуточного ПО в большинстве приложений ASP.NET Core – это класс `EndpointMiddleware`. Обычно он генерирует все наши HTML-страницы и ответы в формате JSON, которым посвящена большая часть этой книги. Как и промежуточное ПО для изменения размера изображения, оно обычно получает запрос, генерирует ответ и затем отправляет его обратно пользователю, как показано на рис. 4.1.

ОПРЕДЕЛЕНИЕ Схема, при которой один компонент может вызывать другой, который, в свою очередь, может вызывать следующий, и так далее, называется *конвейером*. Можно рассматривать каждую часть промежуточного программного обеспечения как секцию канала – когда вы соединяете все секции, запрос перетекает из одной части в другую.

¹ С технической точки зрения промежуточное ПО должно быть функцией, как вы увидите в главе 31, но очень часто оно реализуется в виде класса C# с помощью единственного метода.

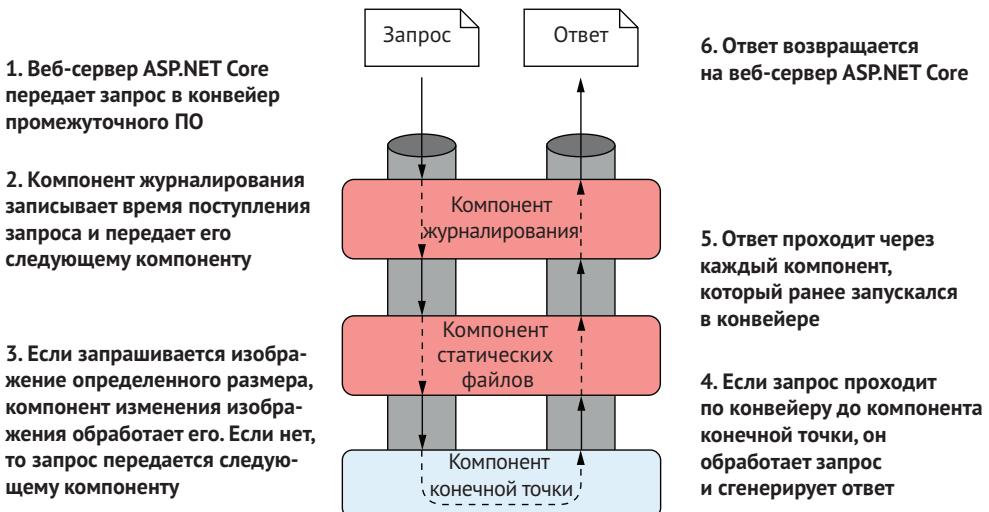


Рис. 4.1 Пример конвейера промежуточного ПО. Каждый компонент обрабатывает запрос и передает его следующему компоненту в конвейере. После того как ответ будет сгенерирован, он передается дальше по конвейеру. Достигнув веб-сервера ASP.NET Core, ответ отправляется в браузер пользователя

Один из наиболее распространенных вариантов использования промежуточного ПО – решение сквозных задач приложения. Эти аспекты приложения должны выполняться для каждого запроса, независимо от конкретного пути в запросе или запрашиваемого ресурса. Сюда входят:

- журналирование каждого запроса;
- добавление стандартных заголовков безопасности в ответ;
- связывание запроса с соответствующим пользователем;
- установка языка текущего запроса.

В каждом из этих примеров компонент получал запрос, изменял его, а затем передавал его следующему компоненту в конвейере. Последующий компонент мог бы использовать детали, добавленные более ранним компонентом, для обработки запроса. Например, на рис. 4.2 компонент аутентификации связывает запрос с пользователем. Он использует эту деталь, чтобы проверить, есть ли у пользователя полномочия на выполнение данного конкретного запроса к приложению.

Если у пользователя есть полномочия, компонент авторизации передаст запрос компоненту конечной точки, чтобы он мог сгенерировать ответ. Если у пользователя нет полномочий, то компонент авторизации может замкнуть конвейер, генерируя ответ напрямую. Он возвращает ответ предыдущему компоненту до того, как компонент конечной точки увидит запрос. Такой сценарий является примером паттерна проектирования *Цепочка обязанностей*.

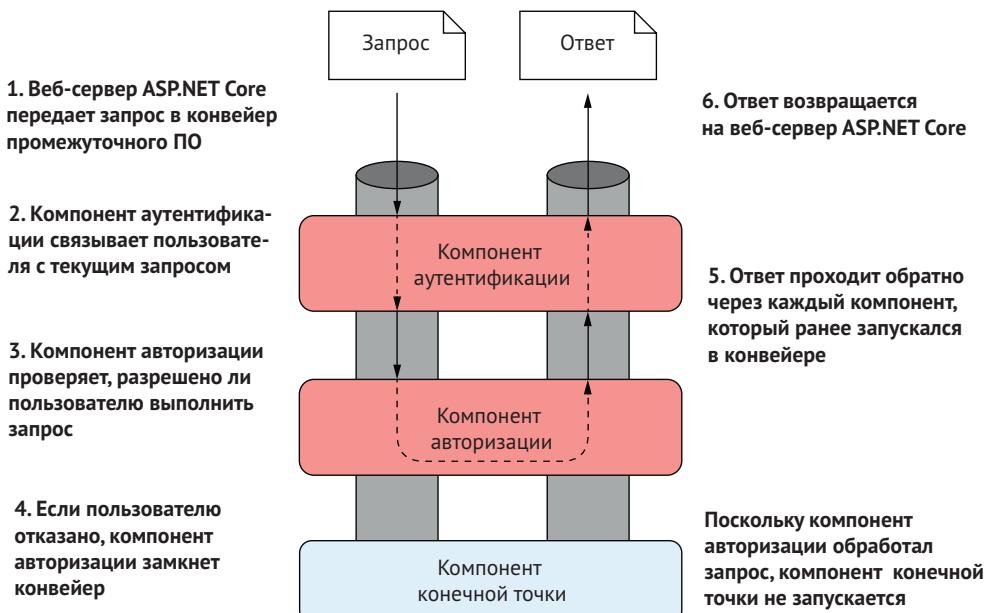


Рис. 4.2 Пример компонента, изменяющего запрос для дальнейшего использования в конвейере. Компонент также может замкнуть конвейер, возвращая ответ до того, как запрос достигнет более позднего компонента

ОПРЕДЕЛЕНИЕ Когда компонент замыкает конвейер и возвращает ответ, это называется *терминальным компонентом*.

Ключевым моментом, на который следует обратить внимание, выступает тот факт, что конвейер является *дву направленным*. Запрос проходит через конвейер в одном направлении, пока какая-то часть не сгенерирует ответ, после чего ответ проходит обратно по конвейеру, проходя через каждый компонент во *второй раз*, пока не вернется к первой части. В конце первый или последний компонент передаст ответ обратно веб-серверу ASP.NET Core.

Объект HttpContext

Я упоминал объект `HttpContext` в главе 3, и здесь он также присутствует за кулисами. Веб-сервер ASP.NET Core создает его для каждого запроса, а приложение ASP.NET Core использует его как своего рода ящик для хранения одного-единственного запроса.

Все, что относится к этому конкретному запросу и последующему ответу, может быть связано с ним и храниться в нем. Это могут быть свойства запроса, сервисы, связанные с запросом, загруженные данные или возникшие ошибки. Веб-сервер заполняет исходный объект `HttpContext` деталями исходного HTTP-запроса и другими деталями конфигурации и передает их остальной части приложения.

Все промежуточное ПО имеет доступ к объекту `HttpContext` для запроса. Он может использоваться, например, чтобы определить, содержит

ли запрос какие-либо учетные данные пользователя, к какой странице запрос пытается получить доступ, и получить любые отправленные данные. Затем он может использовать эти сведения, чтобы определить, как обрабатывать запрос. Как только приложение завершит обработку запроса, оно обновит объект `HttpContext` соответствующим ответом и вернет его через конвейер промежуточного программного обеспечения на веб-сервер. Затем веб-сервер ASP.NET Core преобразует представление в низкоуровневый HTTP-ответ и отправит его на обратный прокси-сервер, который пересыпает его в браузер пользователя.

Как вы видели в главе 3, конвейер промежуточного ПО определяется в коде как часть начальной конфигурации приложения в файле `Program.cs`. Можно настроить конвейер в соответствии со своими потребностями – простым приложением может потребоваться только короткий конвейер, тогда как для больших приложений с различными функциями может использоваться гораздо больше компонентов. Промежуточное ПО – это фундаментальный источник поведения приложения. В конечном итоге конвейер несет ответственность за ответы на все получаемые HTTP-запросы.

Запросы передаются в конвейер промежуточного ПО в виде объектов `HttpContext`. Как вы уже видели в главе 3, веб-сервер ASP.NET Core создает объект `HttpContext` из входящего запроса, который проходит вверх и вниз по конвейеру. Если вы используете существующее промежуточное ПО для создания конвейера, то с этой деталью вы редко будете сталкиваться. Но, как вы увидите в последнем разделе этой главы, его присутствие за кулисами дает возможность получить дополнительный контроль над конвейером.

Также можно рассматривать конвейер промежуточного ПО как серию вложенных компонентов, похожих на традиционную русскую матрешку, как показано на рис. 4.3.

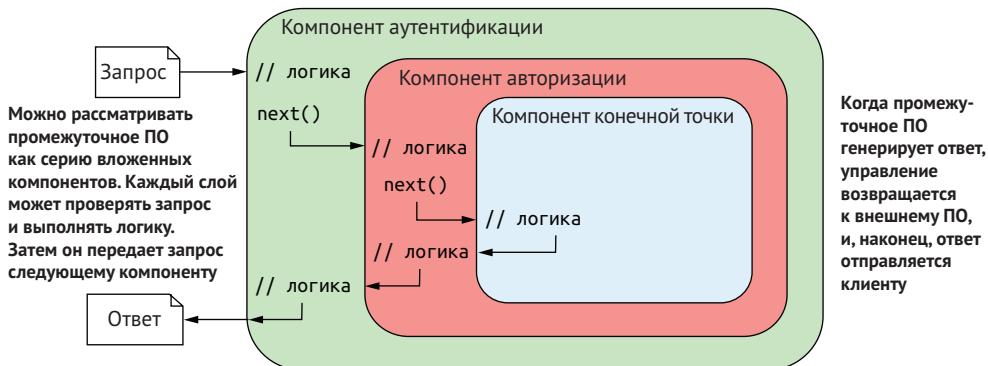


Рис. 4.3 Промежуточное ПО можно также рассматривать как серию вложенных компонентов, в которых запрос отправляется глубже в промежуточное ПО, а из него выходит ответ. Каждый компонент может выполнять логику перед передачей ответа следующему компоненту и после создания ответа на обратном пути из стека

Запрос проходит «через» конвейер, направляясь в стек промежуточного ПО, пока не будет возвращен ответ. Затем ответ возвращается через промежуточное ПО, проходя через компоненты в порядке, обратном запросу.

Промежуточное ПО, или модули HTTP, или обработчики HTTP

В предыдущей версии ASP.NET не использовалась концепция конвейера промежуточного ПО. Вместо этого у вас были модули и обработчики HTTP.

Обработчик *HTTP* – это процесс, который запускается в ответ на запрос и генерирует ответ. Например, обработчик страницы ASP.NET выполняется в ответ на запросы страниц .aspx. В качестве альтернативы можно написать собственный обработчик, который будет возвращать изображения с измененным размером при запросе изображения.

Модули *HTTP* решают общие проблемы приложений, такие как безопасность, журналирование или управление сессиями. Они выполняются в ответ на события жизненного цикла, через которые проходит запрос, когда он получен сервером. Примеры событий включают `BeginRequest`, `AcquireRequestState` и `PostAcquireRequestState`.

Этот подход работает, но иногда сложно понять, какие модули в каких точках будут выполняться. Реализация модуля требует относительно подробного понимания состояния запроса в каждом отдельном событии жизненного цикла.

Конвейер промежуточного ПО значительно упрощает работу с приложением. Конвейер полностью определен в коде, в котором указывается, какие компоненты должны выполняться и в каком порядке. За кулисами конвейер представляет собой цепочку вызовов методов, где каждая функция вызывает следующую в конвейере.

Это почти все, что нужно, чтобы изложить концепцию промежуточного ПО. В следующем разделе я расскажу, как комбинировать компоненты промежуточного ПО для создания приложения и как использовать его для отделения сквозных задач приложения друг от друга.

4.2 Объединение компонентов в конвейер

По сути, у каждого компонента промежуточного ПО есть одна основная задача. Речь идет об обработке только одного из аспектов запроса. Компонент журналирования занимается лишь журналированием запроса, компонент аутентификации – только идентификацией текущего пользователя, а компонент статических файлов занимается только возвратом статических файлов.

Каждая из этих проблем очень целенаправлена, что делает сами компоненты небольшими и понятными, что также придает вашему приложению дополнительную гибкость; добавление компонента статических файлов не означает, что вас заставляют изменять размер

изображения или выполнять аутентификацию. Каждая из этих функций является дополнительным компонентом.

Чтобы создать законченное приложение, несколько компонентов объединяются в конвейер, как показано в предыдущем разделе. Каждый компонент имеет доступ к исходному запросу, а также к любым изменениям объекта `HttpContext`, внесенным предыдущими компонентами в конвейере.

Как только ответ будет сгенерирован, каждый компонент может проверить и/или изменить его, когда он возвращается по конвейеру, прежде чем он будет отправлен пользователю. Это позволяет создавать сложные модели поведения приложений из небольших целенаправленных компонентов.

В оставшейся части данного раздела вы увидите, как создать конвейер промежуточного ПО путем объединения небольших компонентов. Используя стандартные компоненты, вы научитесь создавать страницу приветствия и обслуживать статические файлы из папки на диске. Наконец, мы рассмотрим более сложный конвейер, например в приложении минимального API с компонентами маршрутизации и конечной точки.

4.2.1 Простой сценарий конвейера 1: страница приветствия

В случае со своим первым приложением и первым конвейером промежуточного ПО вы узнаете, как создать приложение, состоящее из страницы приветствия. Это может быть полезно при первой настройке приложения, чтобы гарантировать, что оно обрабатывает запросы без ошибок.

СОВЕТ Помните, код приложения для этой книги можно просмотреть в репозитории GitHub на странице <http://mng.bz/Y1qN>.

В предыдущих главах я упоминал, что фреймворк ASP.NET Core состоит из множества небольших отдельных библиотек. Обычно компонент добавляют, ссылаясь на пакет в файле проекта `.csproj` приложения и настраивая промежуточное ПО в файле `Program.cs`. Компания Microsoft поставляет множество стандартных компонентов с ASP.NET Core на ваш выбор. Вы также можете использовать сторонние компоненты из NuGet и GitHub или создать собственные. Список встроенного промежуточного ПО можно найти на странице <http://mng.bz/Gyxq>.

ПРИМЕЧАНИЕ Я расскажу о создании собственных компонентов в главе 31.

В этом разделе вы увидите, как создать один из простейших конвейеров промежуточного ПО, состоящий только из компонента `WelcomePageMiddleware`. `WelcomePageMiddleware` предназначен для быстрого предоставления образца страницы при первой разработке приложения, как видно на рис. 4.4. Вы вряд ли стали бы использовать его в промышленном окружении, так как здесь нельзя настроить вывод, но это единственный автономный компонент, который можно использовать, чтобы гарантировать правильную работу приложения.

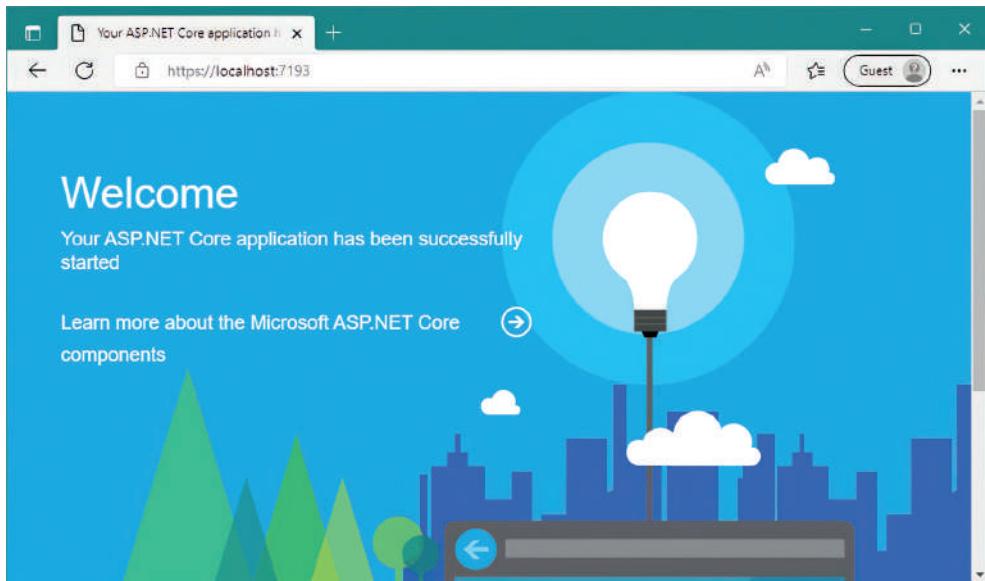


Рис. 4.4 Ответ в виде страницы приветствия. Каждый запрос к приложению на любом пути будет возвращаться в виде такой же страницы приветствия

COBET `WelcomePageMiddleware` входит в состав базового фреймворка ASP.NET Core, поэтому не нужно добавлять ссылку на какие-либо дополнительные пакеты NuGet.

Несмотря на то что это простое приложение, точно такой же процесс, который вы видели раньше, происходит, когда приложение получает HTTP-запрос, как показано на рис. 4.5.

Запрос передается на веб-сервер ASP.NET Core, который создает представление запроса и передает его в конвейер. Поскольку это первый (и единственный!) компонент в конвейере, `WelcomePageMiddleware` получает запрос и должен решить, как его обработать. Промежуточное ПО генерирует ответ в виде HTML-кода, независимо от того, какой запрос получает. Этот ответ возвращается на веб-сервер ASP.NET Core, который пересыпает его пользователю для отображения в его браузере.

Как и во всех приложениях ASP.NET Core, конвейер промежуточного ПО определяется в файле `Program.cs` путем вызова методов `Use*` экземпляра `WebApplication`. Чтобы создать свой первый конвейер, состоящий из одного компонента, нам понадобится всего лишь один вызов метода.

Приложение не требует каких-либо дополнительных настроек или служб, поэтому оно состоит из четырех строк, как показано в следующем листинге.

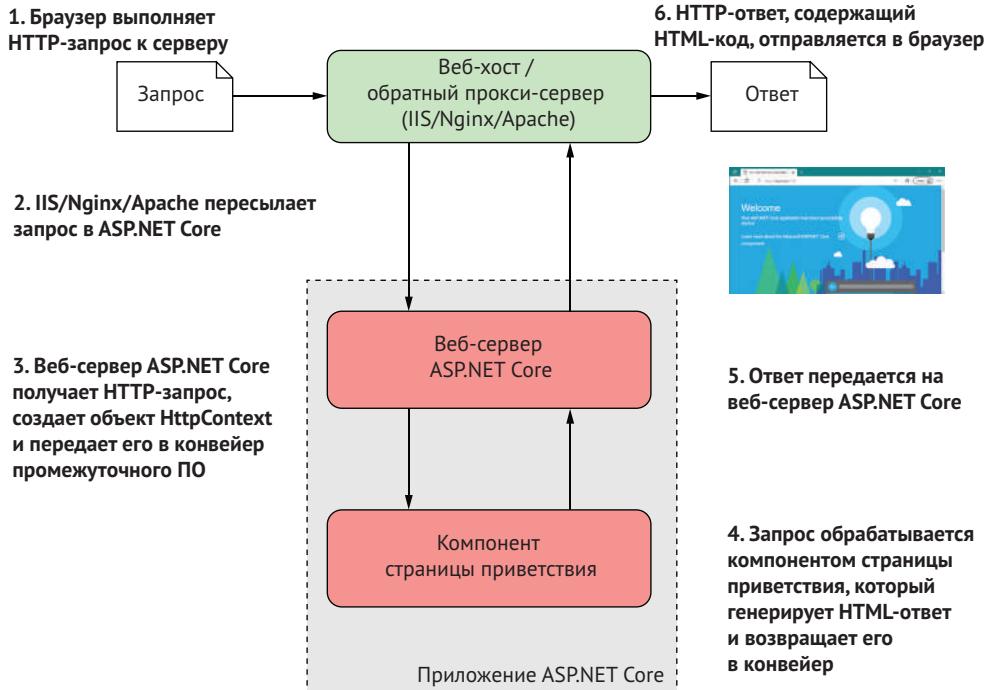


Рис. 4.5 Компонент страницы приветствия обрабатывает запрос. Запрос передается от обратного прокси-сервера в веб-сервер ASP.NET Core и далее в конвейер промежуточного ПО, который генерирует HTML-ответ

Листинг 4.1. Файл Program.cs для конвейера промежуточного ПО страницы приветствия

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args); <-- Использует конфигурацию WebApplication по умолчанию
WebApplication app = builder.Build();

app.UseWelcomePage(); <-- Единственный собственный компонент в конвейере
app.Run(); <-- Запускает приложение для обработки запросов
```

Конвейер промежуточного ПО в ASP.NET Core создается путем вызова методов `WebApplication` (который реализует `IApplicationBuilder`). `WebApplication` не определяет такие методы, как `UseWelcomePage`; это методы *расширения*.

Использование методов расширения позволяет добавлять функциональные возможности к классу `WebApplication`, сохраняя при этом их реализацию изолированной от него. Под капотом методы обычно вызывают еще один метод расширения для добавления компонента в конвейер.

Например, за кулисами метод `UseWelcomePage` добавляет `WelcomePageMiddleware` в конвейер, вызывая

```
UseMiddleware<WelcomePageMiddleware>();
```

Это соглашение о создании метода расширения для каждого компонента. Имя метода начинается со слова `Use`. Это делается для того, чтобы при добавлении компонента в приложение его проще было обнаружить¹.

ASP.NET Core включает в себя множество компонентов в качестве частей основного фреймворка, поэтому вы можете использовать IntelliSense в Visual Studio или другую интегрированную среду разработки для просмотра всех доступных компонентов, как показано на рис. 4.6.

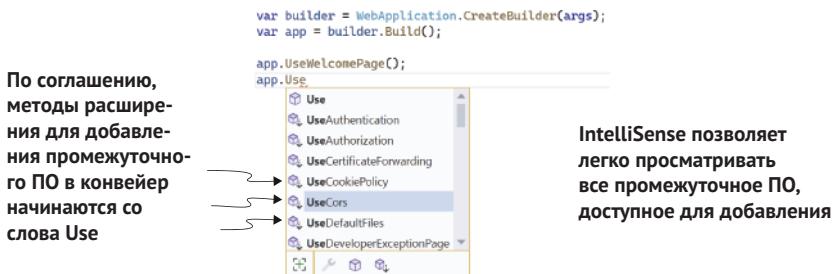


Рис. 4.6 IntelliSense упрощает просмотр всех компонентов, доступных для добавления в конвейер

Вызов метода `UseWelcomePage` добавляет `WelcomePageMiddleware` в качестве следующего компонента в конвейер. Хотя здесь используется только один компонент, важно помнить, что порядок, в котором вызывается `IApplicationBuilder` в методе `Configure`, определяет порядок, в котором компонент будет выполняться в конвейере.

ВНИМАНИЕ Всегда будьте осторожны при добавлении компонента в конвейер и учитывайте порядок, в котором он будет выполняться. Компонент может получить доступ только к данным, созданным компонентами, находящимися перед ним в конвейере.

Это самое простое из приложений, возвращающее один и тот же ответ независимо от того, по какому URL-адресу вы переходите, но оно показывает, насколько легко определить поведение вашего приложения с помощью компонента. Теперь мы сделаем все немного интереснее и будем возвращать разные ответы, когда вы отправляете запросы по разным путям.

4.2.2 Простой сценарий конвейера 2: обработка статических файлов

В этом разделе я покажу вам, как создать один из простейших конвейеров промежуточного ПО, который можно использовать для полноценного приложения: приложения со статическими файлами.

¹ Обратная сторона этого подхода состоит в том, что он может скрыть, какой именно компонент добавляется в конвейер. Если ответ неясен, я обычно ищу исходный код метода расширения на GitHub: <https://github.com/aspnet/aspnetcore>.

Большинство веб-приложений, в том числе и с динамическим содержимым, возвращают некоторые страницы с использованием статических файлов. Это могут быть изображения, таблицы стилей CSS и файлы JavaScript. Обычно такие файлы сохраняются на диск во время разработки и возвращаются при запросе из специальной папки проекта wwwroot как часть полного запроса HTML-страницы.

ОПРЕДЕЛЕНИЕ По умолчанию wwwroot является единственной папкой в приложении, из которой ASP.NET Core будет обслуживать файлы. Он не обслуживает файлы из других папок по соображениям безопасности. Папка wwwroot в проекте ASP.NET Core обычно развертывается в рабочем окружении, включая все содержащиеся в ней файлы и папки.

Для обслуживания статических файлов из папки wwwroot по запросу можно использовать `StaticFileMiddleware`, как показано на рис. 4.7. В этом примере изображение с именем moon.jpg находится в папке wwwroot. Когда вы запрашиваете файл, используя путь /moon.jpg, он загружается и возвращается в качестве ответа на запрос.

Если пользователь запрашивает файл, которого нет в папке wwwroot, например missing.jpg, компонент статических файлов не будет обслуживать файл. Вместо этого в браузер пользователя будет отправлен ответ с кодом ошибки HTTP 404, который по умолчанию выведет страницу с надписью «Файл не найден», как показано на рис. 4.8.

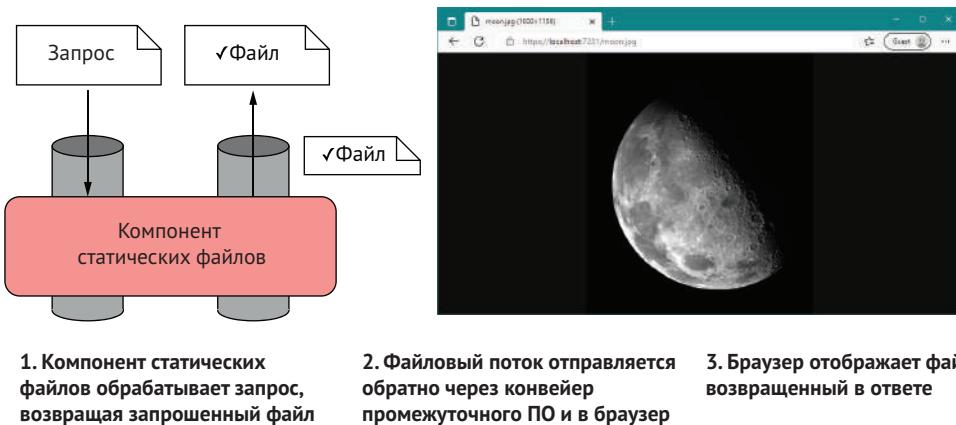
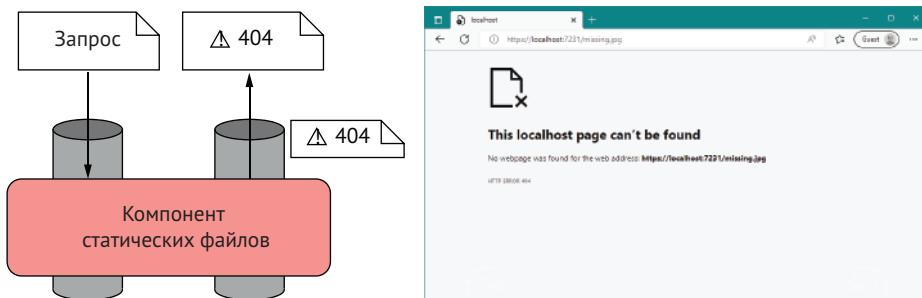


Рис. 4.7 Возврат статического файла изображения с помощью компонента для статических файлов

ПРИМЕЧАНИЕ Внешний вид этой страницы будет зависеть от вашего браузера. В некоторых браузерах, например Internet Explorer (IE), вы можете увидеть полностью пустую страницу.



1. Компонент статических файлов обрабатывает запрос, пытаясь вернуть запрошенный файл, но поскольку его не существует, он возвращает низкоуровневый ответ 404

2. Код ошибки 404 отправляется обратно через конвейер промежуточного ПО к пользователю

3. В браузере по умолчанию отображается страница «Файл не найден»

Рис. 4.8 В браузер возвращается страница с сообщением об ошибке, если файл не существует. Запрошенный файл не существует в папке wwwroot, поэтому приложение ASP.NET Core вернуло ответ с кодом 404. Затем браузер, в данном случае Microsoft Edge, покажет пользователю сообщение по умолчанию «Файл не найден»

Создать конвейер промежуточного ПО для этого приложения очень просто. Он состоит из единственного компонента `StaticFileMiddleware`, как показано в следующем листинге. Никакие сервисы не нужны, поэтому все, что требуется, – это настроить конвейер с помощью метода `UseStaticFiles`.

Листинг 4.2. Файл Program.cs для конвейера компонентов статических файлов

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.UseStaticFiles(); // Добавляет StaticFileMiddleware в конвейер

app.Run();
```

СОВЕТ Помните, что код приложения для этой книги можно просмотреть в репозитории GitHub на странице <http://mng.bz/Y1qN>.

Когда приложение получает запрос, веб-сервер ASP.NET Core обрабатывает его и передает в конвейер. `StaticFileMiddleware` получает запрос и определяет, сможет ли он его обработать. Если запрошенный файл существует, запрос обрабатывается, и в качестве ответа возвращается файл, как показано на рис. 4.9.

Если файл не существует, запрос фактически проходит через компонент статических файлов без изменений. Но подождите, мы добавили только один компонент, верно? Конечно, мы не сможем передать запрос следующему компоненту, если нет другого?

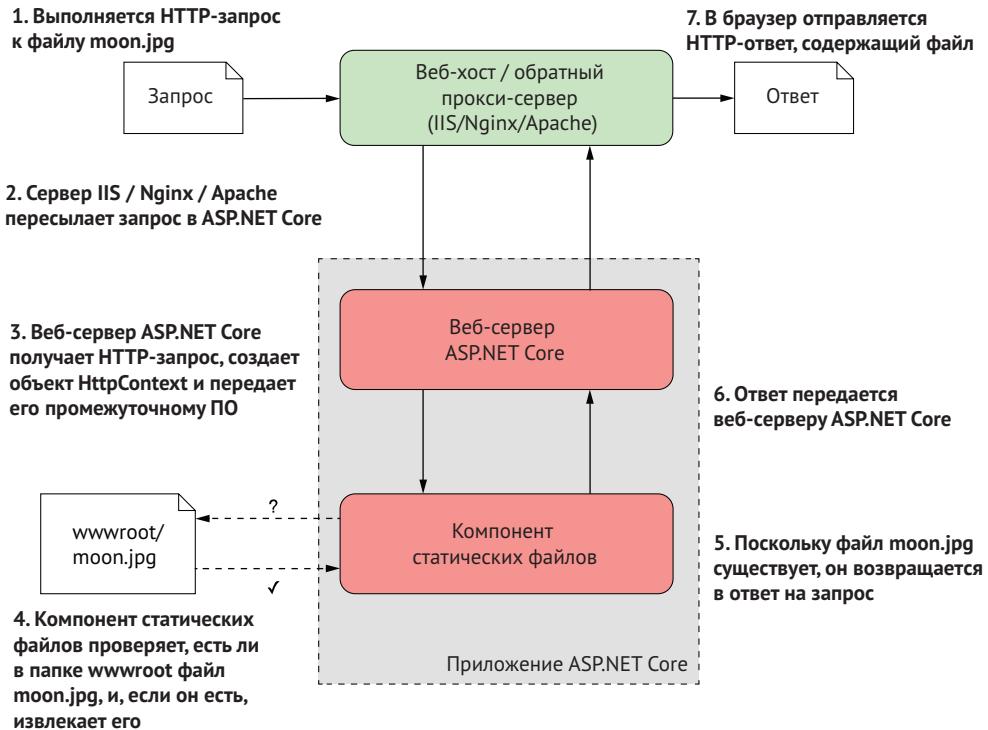


Рис. 4.9 `StaticFileMiddleware` обрабатывает запрос на получение файла. Компонент проверяет папку `wwwroot`, чтобы узнать, существует ли запрашиваемый файл `moon.jpg`. Файл существует, поэтому компонент извлекает его и возвращает в виде ответа веб-серверу и в конечном итоге браузеру

ASP.NET Core автоматически добавляет «фиктивный» компонент в конец конвейера, который при вызове всегда возвращает ответ с ошибкой 404.

СОВЕТ Если компонент не генерирует ответ на запрос, конвейер автоматически вернет браузеру простой ответ с ошибкой 404.

Коды состояния HTTP-ответа

Каждый HTTP-ответ содержит код состояния и, необязательно, поясняющую фразу, описывающую код состояния. Коды состояния ответа являются основополагающими для протокола HTTP и представляют собой стандартный способ обозначения основных результатов. Например, ответ с кодом 200 означает, что на запрос был успешно получен ответ, а ответ с кодом 404 означает, что запрошенный ресурс не может быть найден. Полный список стандартизованных кодов состояния можно увидеть на странице <https://www.rfc-editor.org/rfc/rfc9110#name-status-codes>.

Коды состояния ответа всегда состоят из трех цифр и сгруппированы в пять разных классов в зависимости от первой цифры:

- 1xx – информационные. Нечасто используются, дают общее подтверждение;
- 2xx – успешно. Запрос был успешно обработан;
- 3xx – перенаправление. Браузер должен перейти по предоставленной ссылке, например чтобы пользователь мог войти в систему;
- 4xx – ошибка клиента. Возникла проблема с запросом. Например, в запросе отправлены недопустимые данные или пользователь не авторизован для выполнения запроса;
- 5xx – ошибка сервера. На сервере возникла проблема, из-за которой не удалось выполнить запрос.

Эти коды состояния обычно определяют поведение браузера пользователя. Например, браузер автоматически обработает ответ 301 путем перенаправления на предоставленную новую ссылку и выполнения второго запроса. И все это без вмешательства пользователя.

Коды ошибок находятся в классах 4xx и 5xx. Среди распространенных кодов можно упомянуть ответ 404, когда файл не может быть найден, ошибку 400, когда клиент отправляет недопустимые данные (например, неверный адрес электронной почты), и ошибку 500, когда ошибка возникает на сервере. HTTP-ответы для кодов ошибок могут включать или не включать тело ответа, содержимое которого отображается, когда клиент получает ответ.

Получившееся базовое приложение ASP.NET Core позволяет понять поведение конвейера промежуточного ПО и, в частности, компонента статических файлов, но маловероятно, что ваши приложения будут такими простыми. Более вероятно, что статические файлы станут одной из частей вашего конвейера. В следующем разделе вы узнаете, как объединить несколько компонентов, когда мы будем рассматривать простое приложение с минимальным API.

4.2.3 Простой сценарий конвейера 3: приложение с минимальным API

К этому моменту вы должны четко представлять себе, что такое конвейер промежуточного ПО, а также понимать, что он определяет поведение вашего приложения. В этом разделе вы увидите, как объединить несколько стандартных компонентов для формирования конвейера.

Как и раньше, это делается в файле Program.cs путем добавления промежуточного ПО в объект `WebApplication`.

Мы начнем с создания базового конвейера, который можно найти в типичном шаблоне минимального API ASP.NET Core, а затем расширим его, добавив промежуточное ПО. Результат при переходе на домашнюю страницу приложения показан на рис. 4.10, что идентично примеру, приведенному в третьей главе.

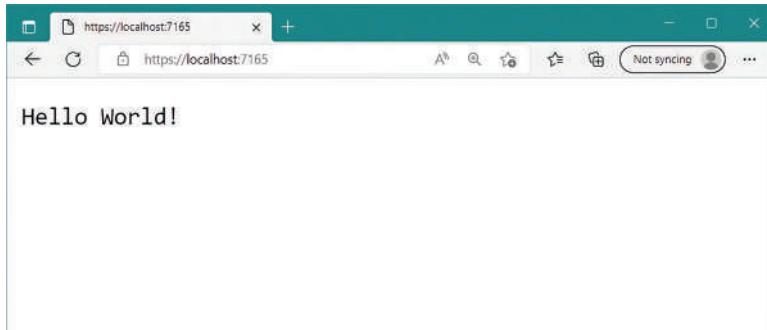


Рис. 4.10 Простое приложение с минимальным API. В приложении используются только четыре компонента: компонент маршрутизации для выбора конечной точки, которую нужно выполнить, компонент конечной точки для генерирования ответа от страницы Razor Page, компонент статических файлов для обслуживания файлов изображений и компонент обработчика исключений для перехвата любых ошибок

Для создания этого приложения требуется всего четыре компонента: компонент маршрутизации для выбора конечной точки минимального API, которую нужно выполнить, компонент конечной точки для генерирования ответа, компонент статических файлов для обслуживания любых файлов изображений из папки wwwroot и компонент обработчика исключений для обработки любых возможных ошибок. Несмотря на то что это по-прежнему пример с Hello World!, эта архитектура намного ближе к реалистичному примеру. В следующем листинге показан пример такого приложения.

Листинг 4.3 Базовый конвейер промежуточного ПО для приложения с минимальным API

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
app.UseDeveloperExceptionPage(); ← Этот вызов не является строго необходимым, поскольку он уже добавлен WebApplication по умолчанию
app.UseStaticFiles();
app.UseRouting(); ← Добавляет RoutingMiddleware в конвейер
app.MapGet("/", () => "Hello World!"); ← Определяет конечную точку для приложения
app.Run();
Добавляет StaticFileMiddleware в конвейер
```

Добавление компонента к объекту `WebApplication` для формирования конвейера должно быть вам знакомо, но в этом примере стоит отметить несколько моментов:

- компонент добавляется с помощью методов `Use*`;
- метод `MapGet` определяет конечную точку, а не компонент. Он определяет конечные точки, которые могут использовать компоненты маршрутизации и конечных точек;

- `WebApplication` автоматически добавляет в конвейер некий компонент, например `EndpointMiddleware`;
- порядок вызовов метода `Use*()` важен и определяет порядок конвейера промежуточного ПО.

Во-первых, все методы добавления компонентов начинаются со слова `Use`. Как я упоминал ранее, это связано с соглашением об использовании методов расширения, чтобы расширить функциональность `WebApplication`; если добавить к методам префикс `Use`, то их будет легче обнаружить. Во-вторых, важно понимать, что метод `MapGet` не добавляет компонент в конвейер; он определяет конечную точку приложения. Эти конечные точки используются компонентами маршрутизации и конечных точек. Подробнее о конечных точках и маршрутизации вы узнаете в главе 5.

СОВЕТ Можно определить конечные точки для своего приложения, используя метод `MapGet()` в любом месте файла `Program.cs` перед вызовом `app.Run()`, но вызовы обычно размещаются после определения конвейера.

В главе 3 я упоминал, что `WebApplication` автоматически добавляет промежуточное ПО в приложение. Вы можете увидеть этот процесс в действии в листинге 4.3, автоматически добавляя компонент `EndpointMiddleware` в конец конвейера. `WebApplication` также автоматически добавляет компонент страницы исключений разработчика в начало конвейера при запуске в окружении разработки. В результате можно опустить вызов метода `UseDeveloperExceptionPage()` из листинга 4.3, и по сути ваш конвейер будет таким же.

WebApplication и автоматически добавляемое промежуточное ПО

`WebApplication` и `WebApplicationBuilder` впервые появились в .NET 6, чтобы попытаться уменьшить количество шаблонного кода, необходимого для приложения `Hello World!`.

В рамках этой инициативы Microsoft решила, чтобы `WebApplication` автоматически добавлял в конвейер различные компоненты. Это решение облегчает ряд распространенных проблем при начале работы с упорядочиванием промежуточного ПО, гарантируя, что, например, метод `UseRouting()` всегда вызывается перед методом `UseAuthorization()`.

Конечно, у всего есть свои компромиссы, и для `WebApplication` компромисс заключается в том, что труднее понять, что именно находится в вашем конвейере, не обладая глубокими знаниями самого кода фреймворка.

К счастью, вам не нужно беспокоиться о компонентах, которые по большей части добавляет `WebApplication`. Если вы новичок в ASP.NET Core, то обычно вы можете согласиться с тем, что `WebApplication` добавит компонент только тогда, когда это необходимо и безопасно.

Тем не менее в некоторых случаях полезно точно знать, что находится в вашем конвейере, особенно если вы знакомы с ASP.NET Core. В .NET 7 `WebApplication` автоматически добавляет некоторые или все перечисленные ниже компоненты в начало конвейера:

- `HostFilteringMiddleware` – этот компонент связан с безопасностью. Подробнее о том, почему он полезен и как его настроить, можно узнать на странице <http://mng.bz/zXxa>;
- `ForwardedHeadersMiddleware` – этот компонент управляет обработкой пересылаемых заголовков. Подробнее об этом можно прочитать в главе 27;
- `DeveloperExceptionPageMiddleware` – как уже говорилось, этот компонент добавляется при запуске в окружении разработки;
- `RoutingMiddleware` – если вы добавляете в приложение какие-либо конечные точки, метод `UseRouting()` выполняется до добавления в приложение какого-либо собственного компонента;
- `AuthenticationMiddleware` – если вы настраиваете аутентификацию, этот компонент аутентифицирует пользователя для запроса. В главе 23 подробно обсуждается аутентификация;
- `AuthorizationMiddleware` – компонент авторизации запускается после аутентификации и определяет, разрешено ли пользователю выполнить конечную точку. Если у пользователя нет на это полномочий, то запрос прерывается. Подробно авторизация обсуждается в главе 24;
- `EndpointMiddleware` – этот компонент соединяется с `RoutingMiddleware` для выполнения конечной точки. В отличие от других компонентов, описанных здесь, `EndpointMiddleware` добавляется в конец конвейера после любого другого компонента, который настраивается в файле `Program.cs`.

В зависимости от конфигурации файла `Program.cs` `WebApplication` может не добавить все эти компоненты. Кроме того, если вы не хотите, чтобы часть этих компонентов находилась в начале вашего конвейера, обычно их местоположение можно переопределить. Например, в листинге 4.3 мы переопределяем автоматическое расположение `RoutingMiddleware`, явно вызывая метод `UseRouting()`, гарантируя, что маршрутизация происходит именно там, где нам это нужно.

Еще один важный момент в листинге 4.3 заключается в том, что порядок добавления компонента к объекту `WebApplication` совпадает с порядком, в котором компонент добавляется в конвейер. Порядок вызовов в листинге 4.3 создает конвейер, аналогичный тому, что показан на рис. 4.11.

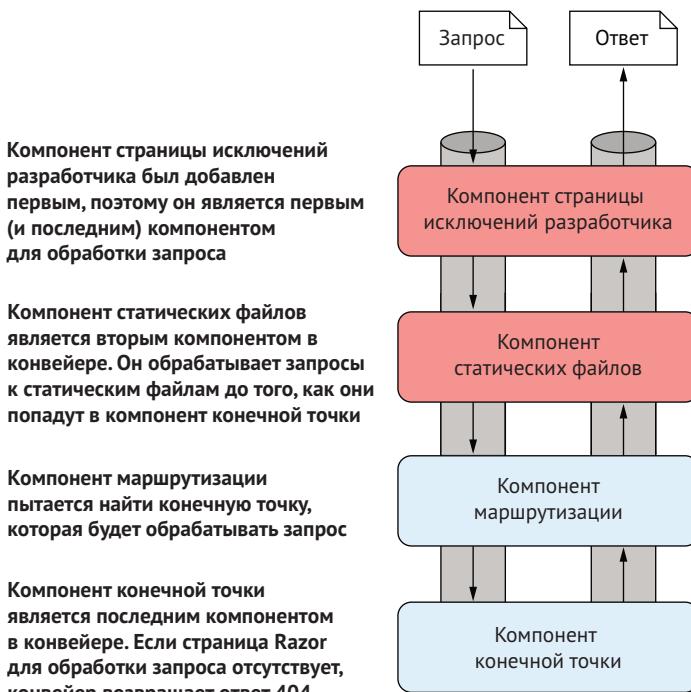


Рис. 4.11 Конвейер промежуточного ПО для приложения из листинга 4.3. Порядок добавления компонента в `WebApplication` определяет порядок компонентов в конвейере

Веб-сервер ASP.NET Core сначала передает входящий запрос компоненту страницы исключений разработчика. Этот компонент изначально игнорирует запрос; его цель – перехватывать любые исключения, создаваемые более поздним компонентом в конвейере, как вы увидите в разделе 4.3. Важно, чтобы этот компонент был размещен в начале конвейера, чтобы он мог перехватывать ошибки, возникающие в более поздних компонентах.

Компонент страницы исключений разработчика передает запрос компоненту статических файлов. Обработчик статического файла генерирует ответ, если запрос соответствует файлу; в противном случае он передает запрос компоненту маршрутизации. Компонент маршрутизации выбирает конечную точку минимального API на основе определенных конечных точек и URL-адреса запроса, а компонент конечной точки выполняет ее.

Если ни одна конечная точка не может обработать запрошенный URL-адрес, автоматический фиктивный компонент возвращает ответ 404.

В главе 3 я упоминал, что `WebApplication` автоматически добавляет `RoutingMiddleware` в начало конвейера. Поэтому вам может быть интересно, почему я явно добавил его в конвейер в листинге 4.3 с помощью метода `UseRouting()`.

Ответ, опять же, связан с порядком компонентов. Добавление явного вызова метода `UseRouting()` сообщает `WebApplication`, что не следует автоматически добавлять `RoutingMiddleware` перед компонентом, определенным в файле `Program.cs`. Это позволяет нам «переместить» `RoutingMiddleware`, чтобы поместить его *после* `StaticFileMiddleware`. Хотя в данном случае такой шаг не является строго необходимым, это хорошая практика. `StaticFileMiddleware` не использует маршрутизацию, поэтому желательно, чтобы этот компонент проверял, относится ли входящий запрос к статическому файлу; если да, то он может замкнуть конвейер и избежать ненужного вызова `RoutingMiddleware`.

ПРИМЕЧАНИЕ В версиях ASP.NET Core 1.x и 2.x компоненты маршрутизации и конечных точек были объединены в один компонент «MVC». Разделение ответственности за маршрутизацию от исполнения позволяет вставлять компоненты *между* компонентами маршрутизации и конечной точки. Подробнее о маршрутизации я расскажу в главах 6 и 14.

Влияние упорядочения наиболее очевидно можно увидеть, когда у вас есть два компонента, оба из которых прослушивают один и тот же путь. Например, компонент конечной точки в примере с конвейером в настоящее время отвечает на запрос к домашней странице приложения (с путем «/»), генерируя ответ в виде строки `Hello world!`, как показано на рис. 4.10. На рис. 4.12 показано, что произойдет, если вы повторно добавите компонент, который уже видели ранее, `WelcomePageMiddleware`, и настроите его таким образом, чтобы он также отвечал на путь «/».

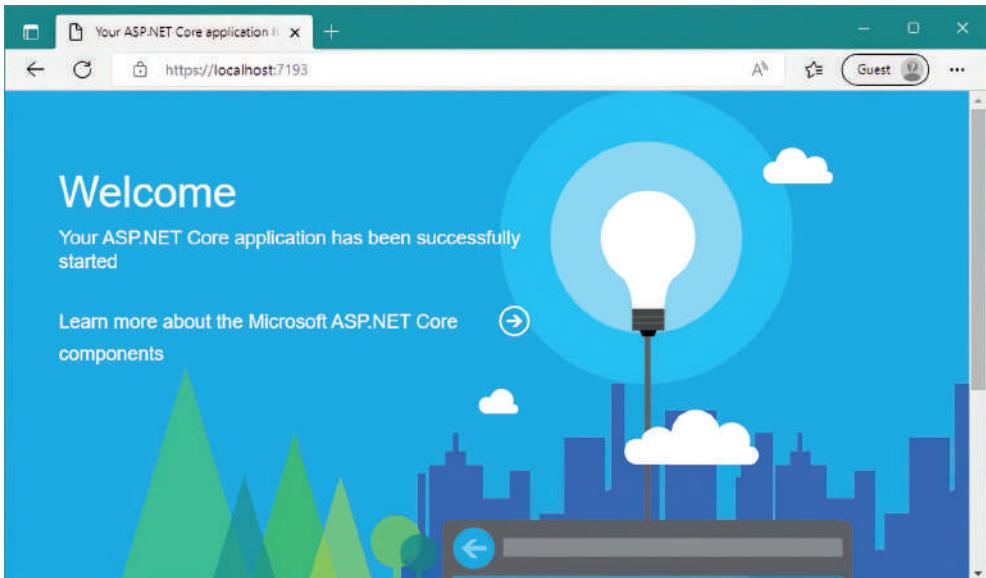


Рис 4.12 Страница приветствия, полученная в качестве ответа. Компонент страницы приветствия предшествует компоненту конечной точки, поэтому запрос на домашнюю страницу возвращает компонент страницы приветствия вместо ответа минимального API

Как вы видели в разделе 4.2.1, `WelcomePageMiddleware` предназначен для возврата фиксированного ответа в виде HTML-кода, поэтому вы не будете использовать его в промышленном окружении, однако он хорошо иллюстрирует суть. В следующем листинге он добавлен в начало конвейера и настроен для ответа только на путь "/".

Листинг 4.4. Добавление компонента `WelcomePageMiddleware` в конвейер

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.UseWelcomePage("/");
app.UseDeveloperExceptionPage();
app.UseStaticFiles();
app.UseRouting();
app.MapGet("/", () => "Hello World!");
app.Run();
```

`app.UseWelcomePage("/");` ← **WelcomePageMiddleware обрабатывает все запросы к пути «/» и возвращает образец ответа в формате HTML**

`app.MapGet("/", () => "Hello World!");` | **Запросы к «/» никогда не достигнут компонента конечной точки, поэтому эта конечная точка не будет вызвана**

Несмотря на то что компонент конечной точки *также* может обрабатывать путь "/", `WelcomePageMiddleware` расположен в конвейере перед ним, поэтому он возвращает ответ, когда получает запрос по пути "/", замыкая конвейер, как показано на рис. 4.13. Остальные компоненты конвейера не выполняются для этого запроса, поэтому ни у кого нет возможности сгенерировать ответ.

Поскольку `WebApplication` автоматически добавляет `EndpointMiddleware` в конец конвейера, `WelcomePageMiddleware` всегда будет впереди него, поэтому в этом примере он всегда генерирует ответ до того, как сможет выполниться конечная точка.

СОВЕТ Всегда нужно учитывать порядок компонентов при добавлении их в `WebApplication`. Компонент, добавленный в конвейер раньше, будет работать (и, возможно, вернет ответ), прежде чем это сделает компонент, добавленный позже.

Во всех показанных до сих пор примерах мы пытались обработать входящий запрос и сгенерировать ответ, но важно помнить, что конвейер компонентов является двунаправленным. Каждый компонент получает возможность обрабатывать как входящий запрос, так и исходящий ответ. Порядок наиболее важен для тех компонентов, которые создают или изменяют исходящий ответ.

В листинге 4.3 я включил `DeveloperExceptionMiddleware` в начало конвейера приложения, но, похоже, он ничего не делал. Компонент обработки ошибок обычно игнорирует входящий запрос, когда тот поступает в конвейер, и вместо этого проверяет исходящий ответ, изменяя его только при возникновении ошибки. В следующем разделе я подробно расскажу о типах компонентов для обработки ошибок, которые можно использовать с вашим приложением, и о том, когда это делать.

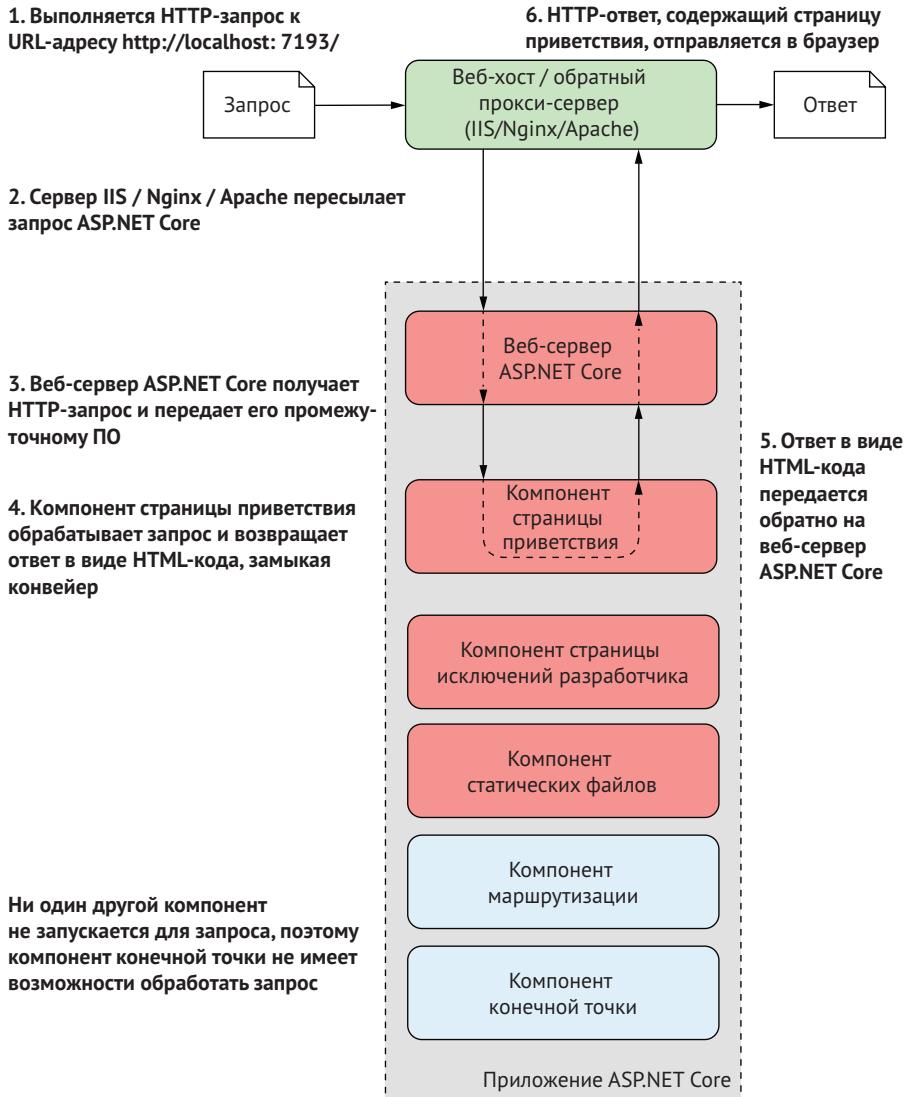


Рис. 4.13 Обзор приложения, обрабатывающего запрос к пути "/". Компонент со страницей приветствия идет первым в конвейере, поэтому он получает запрос раньше любого другого компонента. Он генерирует ответ в виде HTML-кода, замыкая конвейер. Остальные компоненты в данном примере не выполняются для конкретного запроса

4.3 Обработка ошибок с помощью промежуточного ПО

Ошибки – это реальность при разработке приложений. Даже если вы напишете идеальный код, как только вы выпустите и развернете свое приложение, пользователи найдут способ вывести его из строя, случайно или намеренно! Важно, чтобы ваше приложение корректно

обрабатывало эти ошибки, обеспечивая соответствующий ответ пользователю, и не давало сбоя.

Философия проектирования ASP.NET Core заключается в том, что любая функциональность является подключаемой. Итак, поскольку обработка ошибок – это функциональность, необходимо явно активировать ее в своем приложении. В приложении может возникать множество различных типов ошибок, и есть много разных способов их обработки, но в этом разделе я сосредоточусь на двух: исключениях и кодах состояния ошибок.

Исключения обычно возникают всякий раз, когда вы сталкиваетесь с непредвиденными обстоятельствами. Типичное (и очень неприятное) исключение, с которым вы, несомненно, сталкивались раньше, – это исключение `NullReferenceException`, возникающее, когда вы пытаетесь получить доступ к переменной, которая не была инициализирована¹.

Если исключение возникает в компоненте, оно распространяется вверх по конвейеру, как показано на рис. 4.14. Если конвейер не обрабатывает исключение, веб-сервер вернет пользователю код состояния 500.

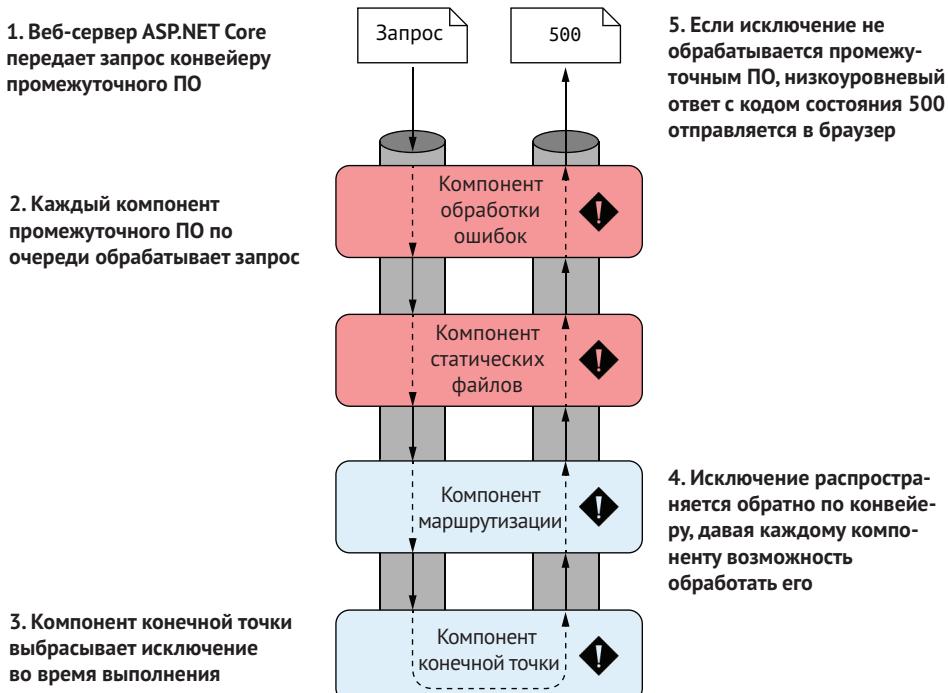


Рис. 4.14 Исключение в компоненте конечной точки распространяется по конвейеру. Если исключение не было перехвачено ранее в конвейере, то в браузер пользователя отправляется код состояния 500 «Ошибка сервера»

¹ В C# 8.0 представлены ссылочные типы, не допускающие значения `NULL`, которые позволяют более четко обрабатывать значения `NULL` и обещают окончательно избавить .NET от исключений `NullReferenceException`! Библиотеки фреймворка ASP.NET Core в .NET 7 полностью поддерживают ссылочные типы, допускающие значение `NULL`. Дополнительную информацию см. в документации: <http://mng.bz/7V0g>.

В некоторых ситуациях ошибка не вызывает исключения. Вместо этого компонент может генерировать код состояния ошибки. Один из таких случаев – когда запрашиваемый путь не обрабатывается. В этой ситуации конвейер вернет ошибку 404.

Для API, которые обычно используются приложениями (а не конечными пользователями), это, вероятно, хороший результат. Но для приложений, которые обычно генерируют HTML, например приложения Razor Pages, возврат ошибки 404 обычно приводит к тому, что пользователь видит универсальную страницу с ошибкой, как на рис. 4.8. Хотя такое поведение является «корректным», оно не обеспечивает удобного взаимодействия с пользователями приложения.

Компонент обработки ошибок пытается решить эти проблемы, изменяя ответ до того, как приложение вернет его пользователю. Как правило, компонент обработки ошибок либо возвращает сведения о произошедшей ошибке, либо возвращает пользователю обычную, но стилизованную HTML-страницу. Этот компонент всегда следует размещать в начале конвейера, чтобы он мог улавливать любые ошибки, генерированные в последующих компонентах, как показано на рис. 4.15. Вы узнаете, как справиться с этим вариантом использования, в главе 13, когда научитесь генерированию ответов с помощью Razor Pages.

В оставшейся части этого раздела мы рассмотрим два основных типа компонентов для обработки ошибок, которые доступны для использования в вашем приложении. Они доступны как часть базового фреймворка ASP.NET Core, поэтому не нужно ссылаться на какие-либо дополнительные пакеты NuGet, чтобы использовать их.

4.3.1 Просмотр исключений в окружении разработки: *DeveloperExceptionPage*

При разработке приложения обычно нужен доступ к как можно большему количеству информации, если где-то в приложении возникает ошибка. По этой причине компания Microsoft предоставляет компонент *DeveloperExceptionPageMiddleware*, который можно добавить в конвейер следующим образом:

```
app.UseDeveloperExceptionPage();
```

ПРИМЕЧАНИЕ Как было показано ранее, *WebApplication* автоматически добавляет этот компонент в конвейер, когда вы работаете в окружении разработки, поэтому не нужно добавлять его явно. Подробнее о вариантах настройки окружений вы узнаете в главе 10.

Когда исключение выбрасывается и распространяется по конвейеру к этому компоненту, оно будет перехвачено. Затем компонент генерирует удобную HTML-страницу, которую возвращает пользователю с кодом состояния 500, как показано на рис. 4.15. Эта страница содержит различные сведения о запросе и исключении, включая трасси-

ровку стека исключений, исходный код в строке, в которой произошло исключение, и детали запроса, например файлы cookie или заголовки, которые были отправлены.

Заголовок, указывающий на проблему

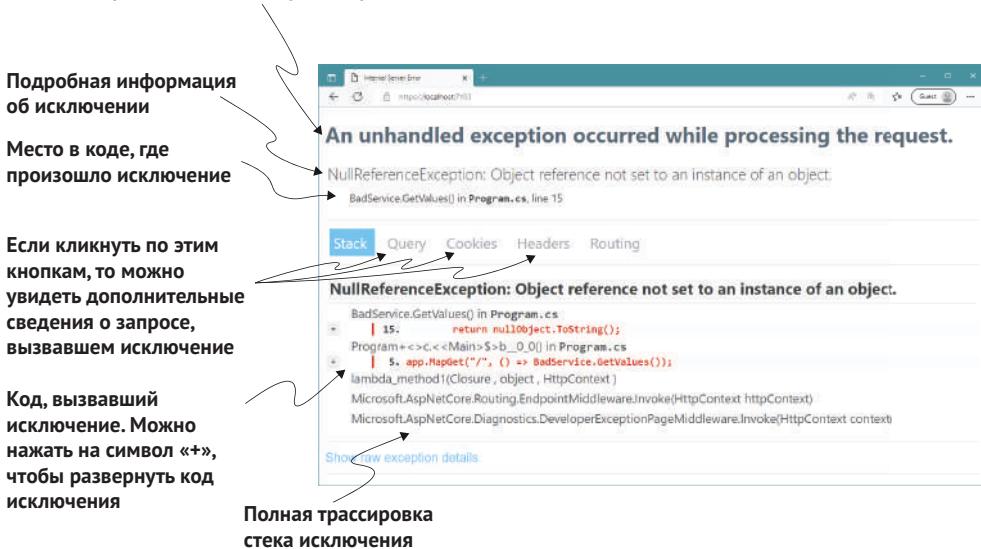


Рис. 4.15 На странице исключения разработчика показана подробная информация об исключении, когда оно возникает во время обработки запроса. Место в коде, вызвавшее исключение, строка исходного кода и трассировка стека отображаются по умолчанию. Вы также можете нажать кнопки **Query** (Запрос), **Cookies** (Файлы cookie), **Headers** (Заголовки) или **Routing** (Маршрутизация), чтобы получить дополнительные сведения о запросе, вызвавшем исключение

Доступность этих сведений при возникновении ошибки имеет неоценимое значение для отладки проблемы, но также представляет угрозу для безопасности при неправильном использовании. Никогда не возвращайте пользователям более подробную информацию о своем приложении, чем нужно, поэтому следует использовать DeveloperExceptionPage только при разработке приложения. Подсказка кроется в названии!

ВНИМАНИЕ Никогда не используйте страницу исключений разработчика при работе в промышленном окружении. Это представляет угрозу для безопасности, так как может предоставить открытый доступ к коду вашего приложения, что делает вас легкой мишенью для злоумышленников. По умолчанию WebApplication использует правильное поведение и добавляет промежуточное ПО только в окружении разработки.

Если страница исключений разработчика не подходит для использования в промышленном окружении, то что тогда использовать? К счастью, есть еще один компонент для обработки ошибок

общего назначения, который можно использовать в промышленном окружении. Вы уже видели его и работали с ним. Это `ExceptionHandlerMiddleware`.

4.3.2 Обработка исключений в промышленном окружении: `ExceptionHandlerMiddleware`

Страница исключений разработчика удобна при разработке приложений, но не следует использовать ее в промышленном окружении, поскольку это может привести к утечке информации о приложении и ей могут воспользоваться потенциальные злоумышленники. Тем не менее ошибки все равно нужно отлавливать; в противном случае пользователи будут видеть недружелюбные страницы с ошибками или пустые страницы, в зависимости от используемого браузера.

Эту проблему можно решить с помощью компонента `ExceptionHandlerMiddleware`. Если в приложении возникает ошибка, то пользователь увидит страницу с ошибкой, которая согласуется с остальной частью приложения, но предоставляет только необходимые сведения об ошибке. В случае с приложением с минимальным API этим ответом может быть JSON или обычный текст, как показано на рис. 4.16.

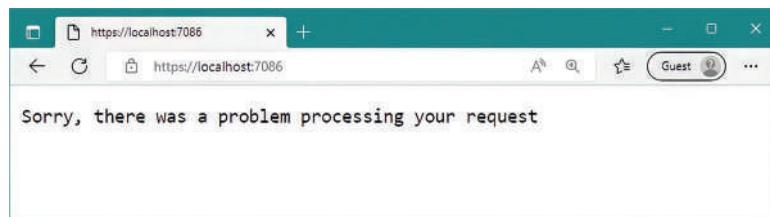


Рис. 4.16 Используя `ExceptionHandlerMiddleware`, можно вернуть обобщенное сообщение об ошибке при возникновении исключения, гарантируя, что никакие конфиденциальные сведения о приложении не «утекут» при работе в промышленном окружении

Для приложений Razor Pages можно создать собственный ответ об ошибке, как, например, тот, что показан на рис. 4.17. Вы сохраняете внешний вид приложения, используя один и тот же заголовок, отображая выполнившего вход пользователя и показывая соответствующее сообщение пользователю вместо подробных сведений об исключении.

Учитывая разные требования к обработчикам ошибок в окружении разработки и промышленном окружении, большинство приложений ASP.NET Core добавляют свой компонент обработчика ошибок условно, в зависимости от окружения размещения. `WebApplication` автоматически добавляет страницу исключений разработчика при запуске в окружении разработки, поэтому, когда вы не находитесь в этом окружении, обычно добавляется `ExceptionHandlerMiddleware`, как показано в следующем листинге.

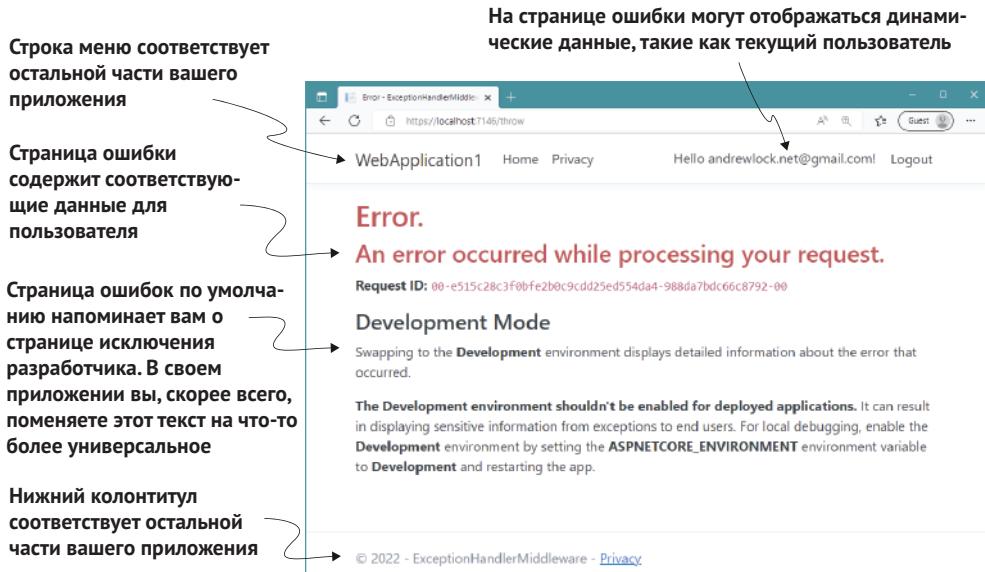


Рис. 4.17 Собственная страница ошибок, созданная ExceptionHandlerMiddleware. Такая страница может иметь тот же внешний вид, что и остальная часть приложения, за счет повторного использования таких элементов, как верхний и нижний колонтитулы. Что еще более важно, вы можете с легкостью контролировать сведения об ошибках, которые видят пользователи

Листинг 4.5. Добавление компонента обработчика исключений в промышленном окружении

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build(); ←
    В окружении разработки
    WebApplication автоматически
    добавляет компонент страницы
    исключений разработчика

if (!app.Environment.IsDevelopment()) ←
{
    app.UseExceptionHandler("/еггог"); ←
}

// Дополнительная настройка компонента;
app.MapGet("/еггог", () => "Sorry, an eggog occurred"); ←
    Настраивает другой конвейер,
    когда он не работает в окру-
    жении разработки

ExceptionHandlerMiddleware не будет пе- ←
редавать конфиденциальные данные при ←
запуске в промышленном окружении ←
    Эта конечная точка ошибки
    будет выполнена при обработке
    исключения
```

Помимо демонстрации того, как добавить компонент ExceptionHandlerMiddleware в свой конвейер, этот листинг показывает, что совершенно приемлемо настраивать различные конвейеры в зависимости от окружения при запуске приложения. Вы также можете изменить свой конвейер на основе других значений, например параметров, загруженных из конфигурации.

ПРИМЕЧАНИЕ Вы увидите, как использовать значения конфигурации для настройки конвейера промежуточного ПО, в главе 10.

При добавлении компонента `ExceptionHandlerMiddleware` в приложение обычно указывается путь к странице ошибки, которую увидит пользователь. В примере из листинга 4.5 мы использовали путь обработки ошибок `"/еггог"`:

```
app.UseExceptionHandler("/еггог");
```

`ExceptionHandlerMiddleware` вызовет этот путь после перехвата исключения, чтобы сгенерировать окончательный ответ. Возможность динамически генерировать ответ – ключевая особенность `ExceptionHandlerMiddleware`: это позволяет повторно запустить конвейер промежуточного ПО, чтобы сгенерировать ответ, отправляемый пользователю.

На рис. 4.18 показано, что происходит, когда `ExceptionHandlerMiddleware` обрабатывает исключение. Здесь показан поток событий, когда конечная точка минимального API для пути `"/"` генерирует исключение. Окончательный ответ возвращает код состояния ошибки, но также предоставляет строку ошибки, используя конечную точку `"/еггог"`.

Последовательность событий, когда где-то в конвейере (или в конечной точке) после `ExceptionHandlerMiddleware` возникает исключение, выглядит следующим образом:

- 1 часть промежуточного ПО выбрасывает исключение;
- 2 `ExceptionHandlerMiddleware` перехватывает его;
- 3 любой определенный к этому моменту частичный ответ удаляется;
- 4 `ExceptionHandlerMiddleware` перезаписывает путь запроса предоставленным путем обработки ошибок;
- 5 компонент отправляет запрос обратно по конвейеру, как если бы исходный запрос был для пути обработки ошибок;
- 6 конвейер генерирует новый ответ как обычно;
- 7 когда ответ возвращается в `ExceptionHandlerMiddleware`, он изменяет код состояния на ошибку 500 и продолжает передавать ответ по конвейеру на веб-сервер.

Одно из основных преимуществ, которое дает повторное выполнение конвейера для приложений Razor Pages, – это возможность интегрировать сообщения об ошибках в обычный макет сайта, как было показано ранее на рис. 4.17. Конечно, при возникновении ошибки можно вернуть фиксированный ответ, но у вас не будет строки меню с динамически генерируемыми ссылками или вы не сможете отобразить имя текущего пользователя в меню. Повторно запуская конвейер, можно убедиться, что все динамические области приложения правильно интегрированы, как если бы эта страница была стандартной страницей сайта.

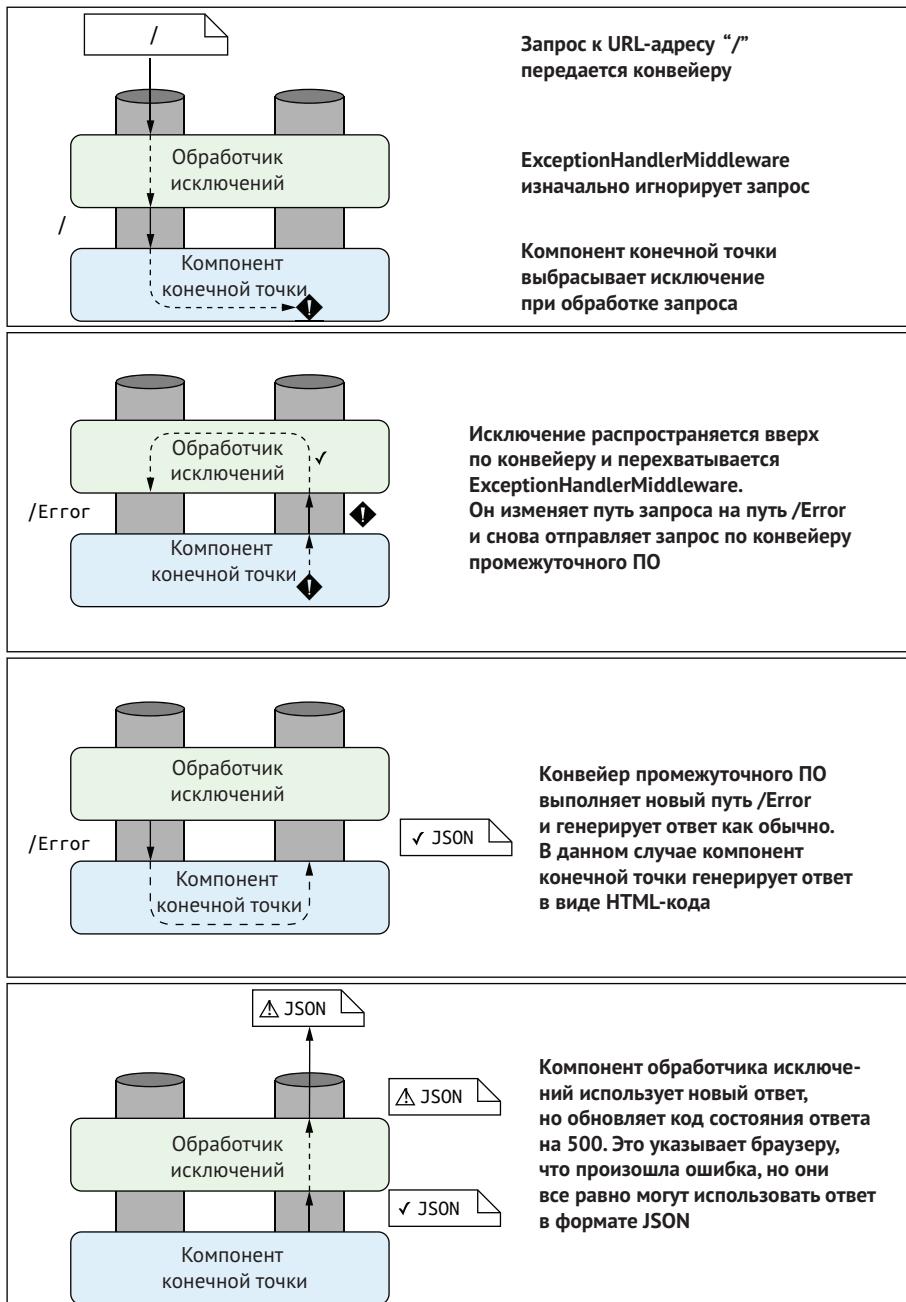


Рис. 4.18 ExceptionHandlerMiddleware обрабатывает исключение, чтобы сгенерировать ответ в формате JSON. Запрос к пути / создает исключение, которое обрабатывается компонентом. Конвейер повторно запускается с использованием пути /error для генерации ответа

ПРИМЕЧАНИЕ Все, что нужно сделать, – это добавить компонент `ExceptionMiddleware` в свое приложение и настроить валидный путь обработки ошибок, чтобы разрешить повторное выполнение конвейера, как показано на рис. 4.18. Компонент перехватит исключение и повторно выполнит переход по конвейеру. Последующий компонент будет рассматривать повторное выполнение как новый запрос, но предыдущий компонент в конвейере не узнает, что случилось что-либо необычное.

Повторное выполнение конвейера – отличный способ сохранить последовательность в своем веб-приложении для страниц с ошибками, но есть некоторые подводные камни, о которых следует знать. Во-первых, промежуточное ПО может изменять ответ, сгенерированный далее по конвейеру, только если ответ *еще не был отправлен клиенту*. Это может быть проблемой, если, например, возникает ошибка, когда ASP.NET Core отправляет статический файл клиенту. Такая ситуация может стать проблемой, если, например, при отправке ASP.NET Core статического файла клиенту возникает ошибка. В этом случае ASP.NET Core может немедленно начать потоковую передачу байтов клиенту из соображений производительности. Когда такое происходит, компонент обработки ошибок не сможет запуститься, так как он не может сбросить ответ. В общем, с этой проблемой мало что можно сделать, но об этом нужно знать.

Более распространенная проблема возникает, когда путь обработки ошибок выбрасывает ошибку во время повторного выполнения конвейера. Представьте себе ошибку в коде, который генерирует меню в верхней части страницы приложения Razor Pages:

- 1 когда пользователь переходит на вашу домашнюю страницу, код для создания строки меню выбрасывает исключение;
- 2 исключение распространяется по конвейеру;
- 3 когда оно доходит до `ExceptionMiddleware`, тот перехватывает его, и конвейер запускается повторно, используя путь обработки ошибок;
- 4 когда страница с ошибкой выполняется, она пытается создать строку меню для вашего приложения, что снова вызывает исключение;
- 5 исключение распространяется по конвейеру;
- 6 `ExceptionMiddleware` уже пытался перехватить запрос, поэтому он позволит ошибке идти до самого верха конвейера;
- 7 веб-сервер возвращает низкоуровневую ошибку с кодом состояния 500, как будто компонента обработки ошибок и не было.

Из-за этой проблемы часто рекомендуется делать страницы обработки ошибок как можно проще, чтобы снизить вероятность возникновения ошибок.

ПРЕДУПРЕЖДЕНИЕ Если при попытке пройти по маршруту, который настроен для обработки ошибок, в свою очередь, возникает новая ошибка, тогда пользователь увидит общую ошибку браузера. Часто лучше использовать статическую страницу, которая

всегда работает, вместо динамической страницы, которая может выдать дополнительные ошибки при ее создании. Альтернативный подход с использованием специальной функции обработки ошибок можно увидеть в этом посте: <http://mng.bz/OKmx>.

Еще одно соображение при создании приложений с минимальными API заключается в том, что в этом случае в ответе не ожидается код HTML. Возврат HTML-страницы в приложение, ожидающее формат JSON, может легко вывести его из строя. Вместо этого код состояния HTTP 500 и тело сообщения в формате JSON, описывающее ошибку, более полезны для приложения-клиента. К счастью, ASP.NET Core позволяет нам делать именно это при создании минимальных API и контроллеров веб-API.

ПРИМЕЧАНИЕ О том, как добавить эту функциональность с помощью минимального API, я расскажу в главе 5, а о том, как сделать это с помощью веб-API, речь пойдет в главе 20.

Мы подошли к концу главы, посвященной промежуточному ПО в ASP.NET Core. Вы увидели, как использовать и составлять компоненты для формирования конвейера, а также как обрабатывать исключения в приложении. Эта информация поможет вам, когда вы начнете создавать свои первые приложения ASP.NET Core. Позже вы узнаете, как создавать собственные компоненты, а также как выполнять сложные операции с конвейером, например разветвлять его в ответ на определенные запросы. В главе 5 мы подробно изучим минимальные API и то, как их можно использовать для создания JSON API.

Резюме

- Промежуточное ПО выполняет ту же роль, что и HTTP-модули и обработчики в старом фреймворке ASP.NET, но оно проще;
- компоненты образуют конвейер, при этом выходные данные одного компонента передаются на вход следующего;
- конвейер является двусторонним: запросы проходят через каждый компонент на входе, а ответы проходят в обратном порядке на выходе;
- компонент может замкнуть конвейер, обработав запрос и вернув ответ, или может передать запрос следующему компоненту в конвейере;
- компонент может изменять запрос, добавляя данные в объект `HttpContext` или изменения его;
- если некоторые запросы достигают компонента, который замыкает конвейер, то последующие компоненты будут недоступны для таких запросов, и их функциональность не будет выполнена в этом случае;
- если запрос не обработан, конвейер вернет код состояния 404;
- порядок, в котором компоненты добавляются в `WebApplication`, определяет порядок, в котором они будут выполняться в конвейере;

- конвейер можно выполнить повторно, если не были отправлены заголовки ответа;
- при добавлении в конвейер `StaticFileMiddleware` будет возвращать запрашиваемые файлы, находящиеся в папке `wwwroot` вашего приложения;
- `DeveloperExceptionPageMiddleware` предоставляет множество информации об ошибках при разработке приложения, но его никогда не следует использовать в промышленном окружении;
- `ExceptionHandlerMiddleware` позволяет предоставлять пользователю удобные сообщения об обработке ошибок при возникновении исключения в конвейере. Он безопасен для использования в промышленном окружении, поскольку не раскрывает конфиденциальные сведения о вашем приложении;
- Microsoft предоставляет ряд распространенных компонентов, а также есть много сторонних вариантов, доступных в NuGet и GitHub.

5

Создание JSON API с помощью минимальных API

В этой главе:

- создание приложения с минимальным API для возврата клиентам ответа в формате JSON;
- генерация ответов с помощью `IResult`;
- использование фильтров для выполнения распространенных действий, например валидации;
- организация API с помощью групп маршрутов.

Пока в этой книге мы видели всего несколько примеров приложений с минимальным API, которые возвращают простые ответы `Hello World!`. Эти примеры отлично подходят для начала работы, но минимальные API также можно использовать для создания полнофункциональных приложений с HTTP API. В этой главе вы познакомитесь с HTTP API и увидите, чем они отличаются от приложений с отрисовкой на стороне сервера, узнаете, когда их использовать.

Раздел 5.2 начинается с подробного описания приложений с минимальными API, которые вы уже видели. Мы изучим ряд основных

концепций маршрутизации, и я покажу, как можно автоматически извлекать значения из URL-адреса. Затем вы научитесь работать с дополнительными методами, такими как POST и PUT, и изучите различные способы определения API.

В разделе 5.3 вы узнаете о различных типах возвращаемых значений, которые можно использовать с минимальными API. Вы увидите, как использовать вспомогательные классы `Results` и `TypedResults`, чтобы легко создавать HTTP-ответы, использующие такие коды состояния, как `201 Created` и `404 Not Found`. Вы также узнаете, как следовать веб-стандартам для описания ошибок с помощью встроенной поддержки сведений о проблеме.

В разделе 5.4 представлена одна из важных функций, добавленных в минимальные API в .NET 7, – фильтры. Их можно использовать для создания мини-конвейера (аналогичного конвейеру промежуточного ПО из главы 4) для каждой из конечных точек. Как и компоненты промежуточного ПО, фильтры отлично подходят для извлечения общего кода из обработчиков конечных точек, что упрощает чтение обработчиков.

О другой важной функции .NET 7 для минимальных API вы узнаете в разделе 5.5: это *группы маршрутов*. Их можно использовать, чтобы уменьшить дублирование в минимальных API, извлекая общие префиксы и фильтры маршрутизации, упрощая чтение API и сокращая количество шаблонного кода. В сочетании с фильтрами группы маршрутов устраниют многие распространенные жалобы, возникающие в отношении минимальных API, когда они были выпущены в .NET 6.

Одним из замечательных аспектов ASP.NET Core является разнообразие приложений, которые можно создавать с его помощью. Возможность легко создавать универсальные HTTP API дает возможность использовать ASP.NET Core в более широком диапазоне ситуаций по сравнению с традиционными веб-приложениями. Но *стоит ли* создавать HTTP API, и если да, то почему? В первом разделе этой главы я рассмотрю ряд причин, по которым вы, возможно, захотите (а возможно, и не захотите) создать веб-API.

5.1 Что такое HTTP API и когда его следует использовать?

Традиционные веб-приложения обрабатывают запросы, возвращая пользователю HTML-код, который отображается в веб-браузере. Вы можете легко создавать приложения такого рода, используя Razor Pages для генерации HTML-кода с помощью шаблонов Razor, как было показано в последних главах. Это распространенный и хорошо понятный подход, но современный разработчик приложений также может рассматривать ряд других возможностей (рис. 5.1), как вы впервые увидели в главе 2.

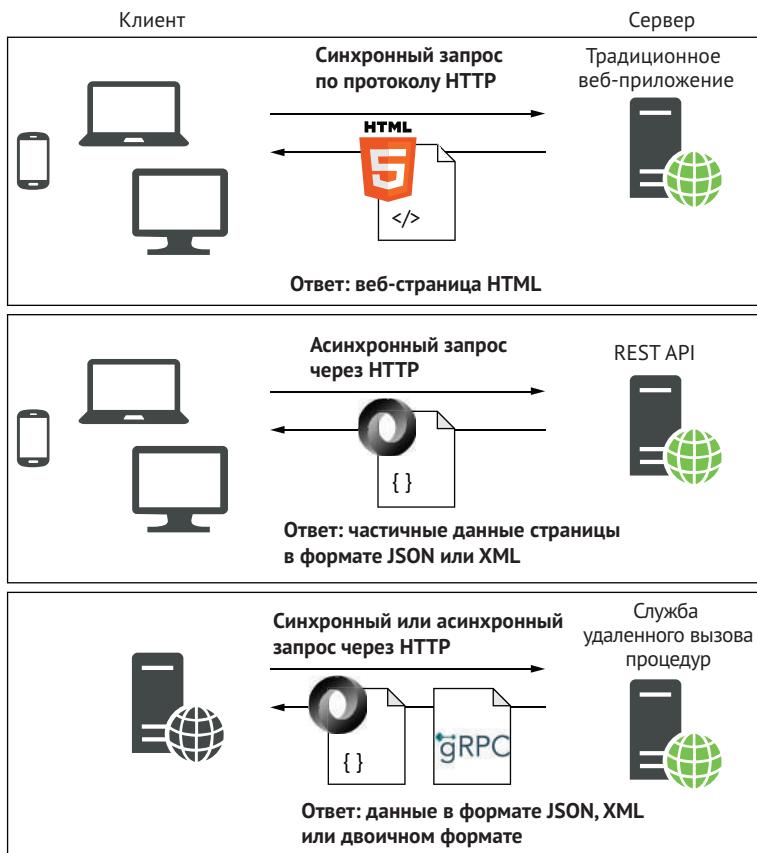


Рис. 5.1 Современные разработчики должны учитывать ряд различных потребителей своих приложений. Как и обычные пользователи с веб-браузерами, это могут быть одностраничные приложения, мобильные приложения или что-то иное

Одностраничные приложения на стороне клиента стали популярными в последние годы с развитием таких фреймворков, как Angular, React и Vue. Эти фреймворки обычно используют JavaScript, который запускается в веб-браузере пользователя, для создания HTML-кода, который они видят и с которым взаимодействуют. Сервер отправляет этот начальный код JavaScript в браузер, когда пользователь впервые обращается к приложению. Браузер пользователя загружает JavaScript и инициализирует одностраничное приложение перед загрузкой любых данных приложения с сервера.

ПРИМЕЧАНИЕ Blazor WebAssembly – это новый захватывающий фреймворк для создания одностраничных приложений. Он позволяет писать приложение, которое запускается в браузере, как и другие одностраничные приложения, но вместо JavaScript применяет C# и шаблоны Razor, используя новый веб-стандарт WebAssembly. В этой книге я не рассматриваю Blazor, поэтому

если хотите узнать больше, то рекомендую книгу Криса Сейнти «Blazor в действии» (ДМК Пресс, 2023).

После загрузки одностраничного приложения в браузер обмен данными с сервером по-прежнему осуществляется по протоколу HTTP, но вместо того, чтобы отправлять HTML-код непосредственно в браузер в ответ на запросы, приложение на стороне сервера отправляет данные – обычно в формате JSON – приложению на стороне клиента. Затем одностраничное приложение анализирует данные и генерирует соответствующий HTML-код, который увидит пользователь, как показано на рис. 5.2. Конечную точку приложения на стороне сервера, с которой взаимодействует клиент, иногда называют *HTTP API*, *JSON API* или *REST API*, в зависимости от особенностей конструкции API.

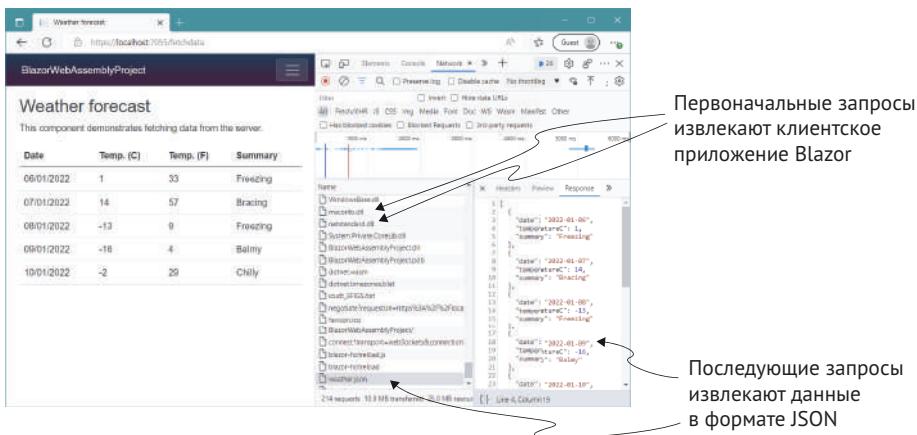


Рис. 5.2 Пример клиентского одностраничного приложения, где используется Blazor WebAssembly. Первоначальные запросы загружают файлы приложения в браузер, а последующие запросы извлекают данные из веб-API в формате JSON

ОПРЕДЕЛЕНИЕ HTTP API предоставляет несколько URL-адресов через протокол HTTP, которые можно использовать для доступа к данным на сервере или их изменения. Обычно он возвращает данные в формате JSON. HTTP API иногда называют веб-API, но поскольку веб-API относится к конкретной технологии в ASP.NET Core, в этой книге я использую термин *HTTP API* для обозначения общей концепции.

В наши дни мобильные приложения стали обычным явлением, и с точки зрения серверного приложения они похожи на клиентские одностраничные приложения. Мобильное приложение обычно обменивается данными с серверным приложением, используя HTTP API, получая данные в таком распространенном формате, как JSON, совсем как в одностраничном приложении. Затем оно изменяет пользовательский интерфейс приложения в зависимости от получаемых данных.

Один из последних вариантов использования HTTP API – это приложение, предназначенное для обработки запросов полностью или частично от других серверных приложений. Представьте, что вы создали веб-приложение для отправки электронных писем. Создавая HTTP API, вы можете разрешить другим разработчикам приложений использовать ваш сервис, отправляя вам адрес электронной почты и сообщение. Многие языки программирования и фреймворки имеют готовые библиотеки для генерации HTTP-запросов, поэтому они также могут подключаться к нашему API напрямую из своего кода.

И этого достаточно, чтобы работать с HTTP API извне. API предоставляет ряд конечных точек (URL-адресов), куда клиентские приложения могут отправлять запросы и получать от них данные. Они используются для управления поведением клиентских приложений, а также для предоставления всех данных, которые необходимы клиентским приложениям для отображения правильного пользовательского интерфейса.

ПРИМЕЧАНИЕ У вас появляется еще больше возможностей при создании API в ASP.NET Core. Вы можете создать API для удаленного вызова процедур, например с помощью gRPC, или представить альтернативный стиль HTTP API, используя стандарт GraphQL. Я не рассматриваю эти технологии в данной книге, но вы можете прочитать о gRPC на странице <https://docs.microsoft.com/aspnet/core/grpc> и узнать о GraphQL в книге Валерио Де Санктис «Создание веб-API с помощью ASP.NET Core» (Manning, 2023).

Нужен ли вам HTTP API для приложения ASP.NET Core, зависит от типа приложения, которое вы хотите создать. Возможно, вы знакомы с фреймворками для разработки приложений на стороне клиента, или, может быть, вам нужно разработать мобильное приложение, или у вас уже настроен конвейер сборки SPA. В любом из этих случаев вы, скорее всего, захотите добавить HTTP API для доступа клиентских приложений к нашему приложению.

Одним из преимуществ использования HTTP API является тот факт, что он может служить универсальной серверной частью для всех ваших клиентских приложений. Например, вы можете начать с создания клиентского приложения, использующего HTTP API. Позже вы можете добавить мобильное приложение, использующее тот же HTTP API, с небольшими изменениями или без изменений в вашем коде ASP.NET Core.

Если вы новичок в веб-разработке, то на начальном этапе HTTP API также могут быть проще для понимания, поскольку обычно они возвращают только JSON. Первая часть этой книги посвящена минимальным API, чтобы вы могли сосредоточиться на механике ASP.NET Core без необходимости писать HTML или CSS.

В третьей части вы узнаете, как использовать Razor Pages для создания приложений с отрисовкой на стороне сервера вместо минимальных API. Такие приложения могут быть очень производительными. Обычно они рекомендуются, когда вам не нужно вызывать приложение за пределами сервера.

лами веб-браузера или когда вы не хотите или не должны прилагать усилия по настройке клиентского приложения.

ПРИМЕЧАНИЕ Несмотря на то что в отрасли произошел сдвиг в сторону фреймворков для разработки приложений на стороне клиента, отрисовка на стороне сервера с использованием Razor по-прежнему актуальна. Какой подход выберете вы, во многом будет зависеть от ваших предпочтений – создавайте приложения традиционным способом либо с использованием JavaScript (или Blazor!) на стороне клиента.

При этом вам не обязательно заранее беспокоиться о том, использовать ли HTTP API в своем приложении. Вы всегда можете добавить их в приложение ASP.NET Core на более позднем этапе разработки, если возникнет такая необходимость.

Одностраничные приложения и ASP.NET Core

Кросс-платформенный и легкий дизайн ASP.NET Core означает, что он хорошо подходит для использования в качестве серверной части для выбранного вами фреймворка. Учитывая тематику этой книги и широкий спектр одностраничных приложений в целом, я не буду рассматривать здесь Angular, React или другие фреймворки. Вместо этого я предлагаю обратиться к ресурсам, соответствующим выбранному вами фреймворку. В издательстве Manning можно найти книги по всем распространенным JavaScript-фреймворкам для создания клиентских приложений, а также по Blazor:

- «React в действии» Марка Тиленса Томаса (*Manning, 2018*);
- «Angular в действии» Джереми Уилкена (*Manning, 2018*);
- «Vue.js в действии» Эрика Хэнчетта и Бенджамина Листвона (*Manning, 2018*);
- «Blazor в действии» Криса Сейнти (*Manning, 2021*).

После того как вы убедились, что вам нужен HTTP API для вашего приложения, его легко будет создать, поскольку в ASP.NET Core это тип приложения по умолчанию! В следующем разделе мы рассмотрим различные способы создания конечных точек минимальных API и способы работы с несколькими HTTP-методами.

5.2 Определение конечных точек минимальных API

В главах 3 и 4 вы познакомились с основными типами конечных точек минимальных API. В этом разделе мы будем опираться на эти базовые возможности, чтобы показать, как работать с несколькими HTTP-методами, и изучим различные способы написания обработчиков конечных точек.

5.2.1 Извлечение значений из URL-адреса с помощью маршрутизации

В этой книге было показано несколько приложений с минимальными API, но до сих пор во всех примерах использовались фиксированные пути для определения API, как в этом примере:

```
app.MapGet("/", () => "Hello World!");
app.MapGet("/person", () => new Person("Andrew", "Lock"));
```

Эти два API соответствуют маршрутам / и /person соответственно. Такая базовая функциональность полезна, но чаще нужно, чтобы некоторые из наших API были более динамичными. Например, такой маршрут в API, как /person, навряд ли будет полезен на практике, поскольку он всегда возвращает один и тот же объект Person. Чаще используются маршруты с возможностью указания конкретного имени пользователя, и тогда API может вернуть всех пользователей с этим именем.

Этого можно добиться, используя *параметризованные маршруты* для настроек API. Вы можете создать параметр в маршруте минимального API, используя выражение {someValue}, где someValue – это любое выбранное вами имя. Значение будет извлечено из пути URL-адреса запроса и может использоваться в лямбда-функции конечной точки.

ПРИМЕЧАНИЕ В этой главе я представляю только базовые примеры для получения значений из маршрутов. В главе 6 вы подробнее узнаете о маршрутизации, в том числе почему мы используем маршрутизацию и как она вписывается в конвейер ASP.NET Core, а также о синтаксисе, который можно использовать.

Если вы создаете API, используя, например, шаблон маршрута /person/{name}, и отправляете запрос по пути /person/Andrew, то параметр name будет иметь значение "Andrew". Можно использовать эту функцию для создания более полезных API, например такого, который показан в следующем листинге.

Листинг 5.1. Минимальный API, использующий значение из URL-адреса

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var people = new List<Person>
{
    new("Tom", "Hanks"),
    new("Denzel", "Washington"),
    new("Leondardo", "DiCaprio"),
    new("Al", "Pacino"),
    new("Morgan", "Freeman"),
};

app.MapGet("/person/{name}", (string name) => ←
```

Создает список
людей в качестве
данных для API

Маршрут параметри-
зован для извлечения
имени из URL-адреса

```
people.Where(p => p.FirstName.StartsWith(name)));
app.Run();
```

Извлеченное значение можно получить в лямбда-выражении обработчика

Если отправить запрос к `/person/Al` для приложения, определенного в листинге 5.1, параметр `name` будет иметь значение "Al", а API вернет следующий код в формате JSON:

```
[{"firstName": "Al", "lastName": "Pacino"}]
```

ПРИМЕЧАНИЕ По умолчанию минимальные API сериализуют объекты C# в JSON. Вы увидите, как возвращать другие типы результатов, в разделе 5.3.

Система маршрутизации ASP.NET Core достаточно сложно устроена, и мы рассмотрим ее более подробно в главе 6. Но с помощью даже такой простейшей маршрутизации уже можно создавать более сложные приложения.

5.2.2 Сопоставление HTTP-методов с конечными точками

До сих пор в этой книге мы определяли все конечные точки минимальных API с помощью функции `MapGet()`. Эта функция соответствует запросам, использующим HTTP-метод `GET`. `GET` – наиболее часто используемый метод; именно его применяет браузер, когда вы вводите URL-адрес в адресную строку браузера или переходите по ссылке на веб-странице.

Однако `GET` следует использовать только для *получения* данных с сервера. Никогда не следует использовать его для *отправки* данных или *изменения* данных на сервере. Вместо этого рекомендуется применять такие методы, как `POST` или `DELETE`. Обычно их нельзя использовать при навигации по веб-страницам в браузере, но их легко отправить из клиентского одностраничного или мобильного приложения.

СОВЕТ Если вы новичок в веб-программировании или хотите повысить свою квалификацию, то Mozilla Developer Network (MDN), создатель веб-браузера Firefox, предлагает неплохое введение в HTTP на странице <http://mng.bz/KeMK>.

Теоретически каждый из HTTP-методов имеет четко определенное назначение, но на практике можно встретить приложения, в которых используются только методы `POST` и `GET`. Это часто подходит для приложений с отрисовкой на стороне сервера, например `Razor Pages`, поскольку обычно это проще, но если вы создаете API, я рекомендую использовать HTTP-методы с соответствующей семантикой там, где это возможно.

Можно определить конечные точки для других команд с минимальными API, используя соответствующие функции `Map*`. Например, чтобы сопоставить конечную точку `POST`, нужно использовать метод `MapPost()`.

В табл. 5.1 показаны доступные методы `Map*` для минимальных API, соответствующие HTTP-методы и типичные семантические ожидания каждого метода в отношении типов операций, которые выполняет API.

Таблица 5.1 Сопоставление конечных точек минимальных API и HTTP-методов

Метод	HTTP-метод	Ожидаемая операция
<code>MapGet(path, handler)</code>	GET	Только получение данных; никаких изменений состояния. Может быть безопасно кешировать
<code>MapPost(path, handler)</code>	POST	Создание нового ресурса
<code>MapPut(path, handler)</code>	PUT	Создание или замена существующего ресурса
<code>MapDelete(path, handler)</code>	DELETE	Удаление данного ресурса
<code>MapPatch(path, handler)</code>	PATCH	Изменение данного ресурса
<code>MapMethods(path, methods, handler)</code>	Несколько методов	Несколько операций
<code>Map(path, handler)</code>	Все методы	Несколько операций
<code>MapFallback(handler)</code>	Все методы	Полезен для резервных маршрутов одностраничного приложения

Обычно REST-совместимые приложения (как описано в главе 2) по возможности придерживаются этих HTTP-методов, но некоторые фактические реализации могут отличаться, и можно легко увлечься педантичностью. Как правило, если вы будете придерживаться ожидаемых операций, описанных в табл. 5.1, вы создадите более понятный интерфейс для потребителей API.

ПРИМЕЧАНИЕ Возможно, вы заметили, что если используете методы `MapMethods()` и `Map()`, перечисленные в табл. 5.1, то ваш API, вероятно, не соответствует ожидаемым операциям поддерживаемых им HTTP-методов, поэтому я избегаю их, где это возможно. Метод `MapFallback()` не имеет пути и вызывается *только* в случае несовпадения никакой другой конечной точки. Резервные маршруты могут быть полезны, если у вас есть одностраничное приложение, использующее маршрутизацию на стороне клиента. См. <http://mng.bz/9DMl>, где приводится описание проблемы и дается альтернативное решение.

Как я упоминал в начале раздела 5.2.2, тестирование API, использующих методы, отличные от `GET`, в браузере – вещь непростая. Необходимо использовать инструменты, позволяющие отправлять произвольные запросы, например `Postman` (<https://www.postman.com>) или плагин HTTP-клиента в `JetBrains Rider`. В главе 11 вы узнаете, как применять инструмент `Swagger UI` для визуализации и тестирования API.

СОВЕТ Плагин HTTP-клиента в `JetBrains Rider` упрощает создание HTTP-запросов из API и даже автоматически обнаруживает

все конечные точки приложения, что упрощает их тестирование. Подробнее об этом можно прочитать на странице https://www.jetbrains.com/help/rider/Http_client_in_product_code_editor.html.

В заключение, прежде чем мы двинемся дальше, стоит упомянуть поведение, которое вы получаете, когда вызываете метод с неправильным HTTP-методом. Если вы определяете API, подобный тому, что показан в листинге 5.1:

```
app.MapGet("/person/{name}", (string name) =>
    people.Where(p => p.FirstName.StartsWith(name)));
```

и вызываете его, используя запрос методом POST к /person/A1 вместо запроса методом GET, обработчик не запустится, и полученный вами ответ будет иметь код состояния 405 Method Not Allowed.

СОВЕТ При вызове API и получении ответа с кодом 405 обязательно проверьте, что вы используете правильный HTTP-метод для запроса и правильный путь. Обычно ошибка 405 означает, что использовался правильный метод, но, возможно, есть ошибки в URL-адресе!

Во всех примерах этой книги мы предоставляем лямбда-функцию в качестве обработчика конечной точки. Но в разделе 5.2.3 вы увидите, что существует множество способов определения обработчика.

5.2.3 Определение обработчиков маршрутов с помощью функций

В базовых примерах использование лямбда-функции в качестве обработчика конечной точки часто является самым простым подходом, но, как показано в следующем листинге, можно использовать множество других подходов. В этом листинге также показано создание простого CRUD (Create, Read, Update, Delete) API, использующего разные HTTP-методы, как описано в разделе 5.2.1.

Листинг 5.2. Создание обработчиков маршрутов для простого CRUD API

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
app.MapGet("/fruit", () => Fruit.All); ← Лямбда-выражения – это самый простой, но наименее наглядный способ создания обработчика

var getFruit = (string id) => Fruit.All[id];
app.MapGet("/fruit/{id}", getFruit); ← Сохранение лямбда-выражения в виде переменной означает, что вы можете дать ему имя, в данном случае это getFruit

app.MapPost("/fruit/{id}", Handlers.AddFruit); ← Обработчики могут быть статическими методами в любом классе.

Handlers handlers = new();
app.MapPut("/fruit/{id}", handlers.ReplaceFruit()); ← Обработчики также могут быть методами экземпляра класса
```

```

app.MapDelete("/fruit/{id}", DeleteFruit); ← Вы также можете использовать локальные функции, представленные в C# 7.0, в качестве методов-обработчиков

app.Run();

void DeleteFruit(string id) ←
{
    Fruit.All.Remove(id);
}

record Fruit(string Name, int Stock)
{
    public static readonly Dictionary<string, Fruit> All = new();
};

class Handlers
{
    public void ReplaceFruit(string id, Fruit fruit) ← Обработчики также могут быть методами экземпляра классов
    {
        Fruit.All[id] = fruit;
    }

    public static void AddFruit(string id, Fruit fruit) ← Преобразует ответ в JsonObject
    {
        Fruit.All.Add(id, fruit);
    }
}

```

Листинг 5.2 демонстрирует различные способы передачи обработчиков конечной точке, моделируя простой API для взаимодействия с коллекцией элементов `Fruit`:

- лямбда-выражение, как в конечной точке `MapGet("/fruit")`;
- переменная `Func<T, TResult>`, как в конечной точке `MapGet("/fruit/{id}")`;
- статический метод, как в конечной точке `MapPost`;
- метод переменной экземпляра, как в конечной точке `MapPut`;
- локальная функция, как в конечной точке `MapDelete`.

Все эти подходы идентичны с функциональной точки зрения, поэтому можете использовать тот шаблон, который вам больше всего подходит.

У каждой записи `Fruit` в листинге 5.2 есть `Name` и `Stock`, и она хранится в словаре с идентификатором. Мы вызываем API, используя разные HTTP-методы для выполнения операций CRUD со словарем.

ВНИМАНИЕ Это простой API. Он не является потокобезопасным, не проверяет ввод пользователя и не обрабатывает пограничные случаи. Некоторые из этих недостатков мы исправим в разделе 5.3.

Обработчики конечных точек `POST` и `PUT` в листинге 5.2 принимают как параметр `id`, так и параметр `Fruit`, что демонстрирует еще одну важную возможность минимальных API. *Сложные типы*, т. е. типы, которые

невозможно извлечь из URL-адреса с помощью параметров маршрута, создаются путем десериализации JSON-объектов из тела запроса.

ПРИМЕЧАНИЕ В отличие от API, созданных с использованием контроллеров веб-API ASP.NET и ASP.NET Core (которые мы рассмотрим в главе 20), в минимальных API данные могут быть привязаны к телу запроса только в формате JSON и для их десериализации необходимо использовать библиотеку System.Text.Json.

На рис. 5.3 показан пример запроса методом POST, отправленного с помощью Postman. Postman отправляет тело запроса в формате JSON, который минимальный API автоматически десериализует в экземпляр Fruit перед вызовом обработчика конечной точки. Таким способом мы можем привязать только один объект в обработчике конечной точки к телу запроса. Подробнее о привязке модели рассказывается в главе 7.

Минимальные API позволяют свободно настраивать конечные точки любым удобным для вас способом. Эта гибкость может быть причиной не использовать их из-за опасений, что разработчики сохранят всю функциональность в одном файле, как в большинстве примеров (например, в листинге 5.2). На практике вам, скорее всего, понадобится извлекать конечные точки в отдельные файлы, чтобы сделать их модульными и облегчить понимание. Смиритесь с этим желанием; именно для этого они и предназначены!

Теперь у нас есть простой API, но если вы его попробуете, то быстро столкнетесь с ситуацией, в которой ваш API может не работать. В разделе 5.3 вы узнаете, как обрабатывать некоторые из таких сценариев, возвращая коды состояния.

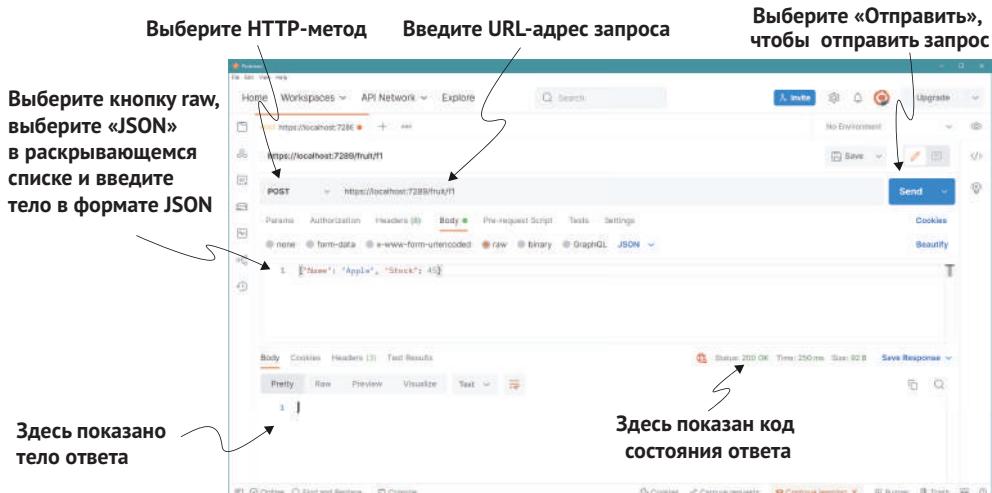


Рис. 5.3 Отправка запроса методом POST с помощью Postman. Минимальный API автоматически десериализует JSON в теле запроса в экземпляр Fruit перед вызовом обработчика конечной точки

5.3 Генерация ответов с помощью IResult

Мы познакомились с основами минимальных API, но до сих пор рассматривали только правильную маршрутизацию, при которой можно успешно обработать запрос и вернуть ответ. В этом разделе мы разберем, как обрабатывать неверные запросы и другие ошибки, возвращая различные коды состояния из нашего API.

API из листинга 5.2 работает корректно, пока выполняются только те операции, которые являются валидными для текущего состояния приложения. Например, если вы отправите запрос методом GET в /fruit, то всегда получите успешный ответ 200, но если вы отправите такой запрос в /fruit/f1 до того, как создадите Fruit с идентификатором f1, вы получите исключение и ошибку 500 Internal Server Error, как показано на рис. 5.4.

Выброс исключения всякий раз, когда пользователь запрашивает несуществующий идентификатор, – это плохой стиль программирования API. Лучшим подходом является возврат кода состояния, указывающего на проблему, например 404 Not Found или 400 Bad Request. Самый простой способ сделать это, используя минимальные API, – вернуть экземпляр объекта типа IResult.

Все обработчики конечных точек, которые вы видели до сих пор в этой книге, возвращали void, строку или объект POCO (то есть простые объекты без наследования и зависимостей), например Person или Fruit. Существует еще один тип объекта, который можно вернуть из конечной точки, – это реализация интерфейса IResult.

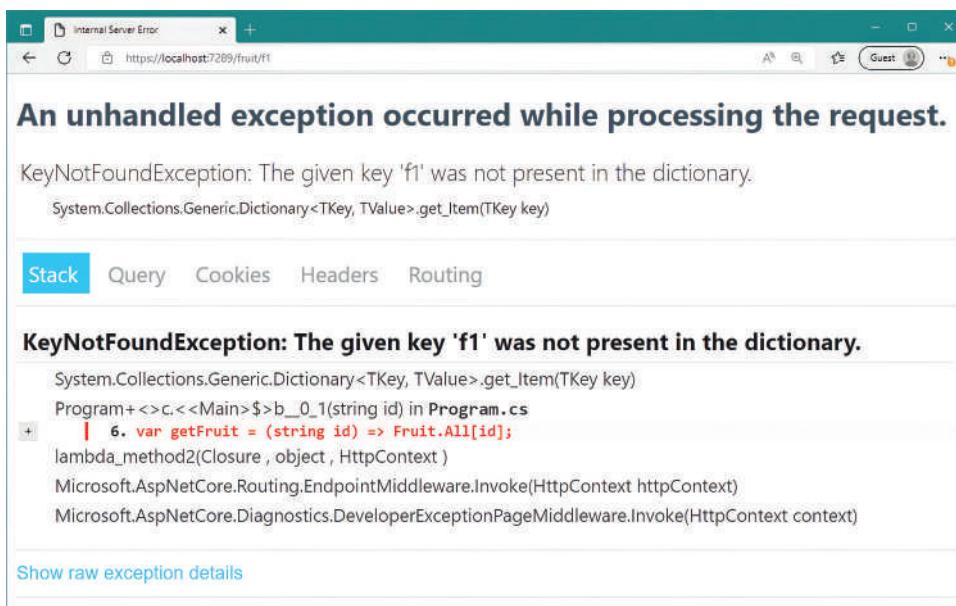


Рис. 5.4 Если вы попытаетесь получить объект, используя несуществующий идентификатор для минимального API из листинга 5.2, то такая конечная точка выбросит исключение. Это исключение обрабатывается DeveloperExceptionPageMiddleware, но дает малопонятный результат

Подведем итоги. Компонент конечной точки обрабатывает каждый тип возвращаемого значения следующим образом:

- `void` или `Task` – конечная точка возвращает ответ `200` без тела;
- `string` или `Task<string>` – конечная точка возвращает ответ `200` со строкой, сериализованной в теле как `text/plain`;
- `IResult` или `Task<IResult>` – конечная точка выполняет метод `IResult.ExecuteAsync`. В зависимости от реализации этот тип может настраивать ответ, возвращая любой код состояния;
- `T` или `Task<T>` – все остальные типы (например, объекты РОСО) сериализуются в `JSON` и возвращаются в теле ответа `200` как `application/json`.

Реализации `IResult` обеспечивают большую гибкость минимальных API, как вы увидите в разделе 5.3.1.

5.3.1 Возврат кодов состояния с помощью `Results` и `TypedResults`

Хорошо спроектированный API использует коды состояния, чтобы указать клиенту, что пошло не так, если запрос не выполнен, а также потенциально предоставляет более информативные коды в случае успешного запроса. Вы должны предвидеть распространенные проблемы, которые могут возникнуть, когда клиенты вызывают ваш API и возвращают соответствующие коды состояния, чтобы указать пользователям причины.

ASP.NET Core предоставляет простые статические вспомогательные типы `Results` и `TypedResults` в пространстве имен `Microsoft.AspNetCore.Http`. Эти типы можно использовать для создания ответа с общими кодами состояния, которые при необходимости включают тело `JSON`. Каждый из методов для `Results` и `TypedResults` возвращает реализацию `IResult`, которую компонент конечной точки выполняет для генерации окончательного ответа.

ПРИМЕЧАНИЕ `Results` и `TypedResults` выполняют ту же функцию, что и вспомогательные методы для генерирования кодов состояния. Единственное отличие состоит в том, что методы `Results` возвращают `IResult`, тогда как `TypedResults` возвращают конкретный обобщенный тип, например `Ok<T>`. С точки зрения функциональности разницы нет, но обобщенные типы проще использовать в модульных тестах и в документации OpenAPI, как вы увидите в главах 36 и 11. `TypedResults` были добавлены в .NET 7.

В следующем листинге показана обновленная версия листинга 5.2, в которой мы устранием некоторые недостатки API и используем `Results` и `TypedResults` для возврата клиентам различных кодов состояния.

Листинг 5.3. Использование Results и TypedResults в минимальном API

```

using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

Попытка
добавить
объект
в сло-
варь.
Если
иденти-
фикатор
еще не
добав-
лен, воз-
вращает
true...
var _fruit = new ConcurrentDictionary<string, Fruit>(); <-- Использует
app.MapGet("/fruit", () => _fruit); <-- конкурентный
                                         словарь, чтобы
                                         сделать API пото-
                                         кобезопасным

Пытается получить
данные об объекте
из словаря. Если
идентификатор су-
ществует в словаре,
возвращается true...
app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit) <-- ...Возвращается
                                                ответ 200 OK, се-
                                                риализуя объект
                                                в теле как JSON
    ? TypedResults.Ok(fruit) <-- Если идентификатор не
    : Results.NotFound(); <-- существует, то возвраща-
                                         ется ответ 404 Not Found

...Возвращается
ответ 201 с телом
JSON, и присваи-
вается значение
пути в заголовок
Location
app.MapPost("/fruit/{id}", (string id, Fruit fruit) =>
    _fruit.TryAdd(id, fruit)
    ? TypedResults.Created($"/fruit/{id}", fruit)
    : Results.BadRequest(new
        { id = "A fruit with this id already exists" })); <--

Если идентифи-
катор уже суще-
ствует, возвра-
щает ответ 400
Bad Request
с сообщением
об ошибке
app.MapPut("/fruit/{id}", (string id, Fruit fruit) =>
{
    _fruit[id] = fruit; <-- После добавления или
    return Results.NoContent(); <-- замены объектов возв-
                                         рашается ответ 204
                                         No Content

После удаления объекта
всегда возвращается ответ
204 No Content
app.MapDelete("/fruit/{id}", (string id) =>
{
    _fruit.TryRemove(id, out _); <-- После удаления объекта
    return Results.NoContent(); <-- всегда возвращается ответ
                                         204 No Content

});
app.Run();
record Fruit(string Name, int Stock);

```

В листинге 5.3 показано несколько кодов состояния, некоторые из которых могут быть вам неизвестны:

- 200 OK – стандартный ответ на успешный запрос. Часто включает содержимое в теле ответа, но это не обязательно;
- 201 Created – обычно используется, когда объект на сервере успешно создается. Результат Created в листинге 5.3 включает заголовок Location для описания URL-адреса, по которому можно найти объект, а также сам объект JSON в теле ответа;
- 204 No Content – аналогичен ответу 200, но без содержимого в теле ответа;
- 400 Bad Request – указывает, что запрос по каким-то причинам не является валидным; часто используется для обозначения ошибок проверки данных;
- 404 Not Found – указывает на то, что запрашиваемый объект не найден.

Эти коды состояния более точно описывают ваш API и могут упростить его использование. Тем не менее если вы используете только код состояния 200 OK для всех своих успешных запросов, то это тоже возможно. Сводку всех возможных кодов состояния и их предполагаемого использования можно увидеть на странице <http://mng.bz/jP4x>.

ПРИМЕЧАНИЕ В частности, код состояния 404 вызывает бесконечные споры на онлайн-форумах. Следует ли использовать его *только* в том случае, если запрос не соответствует конечной точке? Можно ли использовать 404 для обозначения отсутствующего объекта (как в предыдущем примере)? В обоих лагерях есть бесконечное количество сторонников, так что выбирайте сами!

Results и *TypedResults* включают методы для всех распространенных результатов кодов состояния, которые могут вам понадобиться, но если вы по какой-то причине не хотите их использовать, то всегда можете задать код состояния самостоятельно непосредственно в *HttpResponse*, как в листинге 5.4. Фактически в листинге показано, как вручную определить весь ответ, включая код состояния, тип контента и тело ответа. Вам не придется часто использовать этот ручной подход, но в некоторых ситуациях он может быть полезен.

Листинг 5.4. Написание ответа вручную с помощью *HttpResponse*

```
using System.Net.Mime
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
app.MapGet("/teapot", (HttpResponse response) => {
    response.StatusCode = 418;
    response.ContentType = MediaTypeNames.Text.Plain;
    return response.WriteAsync("I'm a teapot!");
});
app.Run();
```

Можно задать код статуса
прямо в ответе

Получает доступ
к *HttpResponse*, включая его в качестве параметра в обработчик конечной точки

Вы можете записывать данные в поток ответа вручную

Определяет тип содержимого, которое будет отправлено в ответе

HttpResponse представляет собой ответ, который будет отправлен клиенту, и является одним из специальных типов, которые минимальные API умеют внедрять в обработчики конечных точек (вместо того чтобы пытаться создать его путем десериализации из тела запроса). О других типах, которые можно использовать в обработчиках конечных точек, вы узнаете в главе 7.

5.3.2 Возврат полезных данных об ошибках с помощью *Problem Details*

В конечной точке *MapPost* из листинга 5.3 мы проверили, существует ли уже сущность с данным идентификатором. Если это так, то мы возвра-

щаем код состояния `400` с описанием ошибки. Проблема такого подхода заключается в том, что клиент – обычно мобильное или одностороннее приложение – должен знать, как читать и анализировать этот ответ. Если у каждого из ваших API свой формат ошибок, это может привести к путанице в API. К счастью, веб-стандарт под названием Problem Details описывает единый формат для использования.

ОПРЕДЕЛЕНИЕ Problem Details – это веб-спецификация (<https://www.rfc-editor.org/rfc/rfc7807.html>) для предоставления машиночитаемых ошибок для HTTP API. Он определяет обязательные и необязательные поля, которые должны быть в теле JSON в случае ошибок.

ASP.NET Core включает два вспомогательных метода для генерации ответов Problem Details из минимальных API: `Results.Problem()` и `Results.ValidationProblem()` (плюс их аналоги `TypedResults`). Оба этих метода возвращают ответ в формате JSON. Единственное отличие состоит в том, что методу `Problem()` по умолчанию присваивается код состояния `500`, тогда как методу `ValidationProblem()` по умолчанию присваивается код состояния `400` и требуется передать словарь ошибок валидации, как показано в следующем листинге.

Листинг 5.5. Возврат ответа Problem Details с помощью `Results.Problem`

```
using System.Collections.Concurrent;
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();
app.MapGet("/fruit", () => _fruit);

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
    ? TypedResults.Ok(fruit)
    : Results.Problem(statusCode: 404)); <-- Возвращает объект Problem
                                            Details с кодом состояния 404

app.MapPost("/fruit/{id}", (string id, Fruit fruit) =>
    _fruit.TryAdd(id, fruit)
    ? TypedResults.Created($""/fruit/{id}", fruit)
    : Results.ValidationProblem(new Dictionary<string, string[]>
    {
        {"id", new[] {"A fruit with this id already exists"}}
    }));

```

Возвращает
объект Problem
Details с кодом
состояния 400
и включает ошиб-
ки валидации

`ProblemHttpResult`, возвращаемый этими методами, заботится о включении правильного заголовка и описания на основе кода состояния и генерирует соответствующий код JSON, как показано на рис. 5.5. Можно переопределить заголовок и описание по умолчанию, передав дополнительные аргументы методам `Problem()` и `ValidationProblem()`.

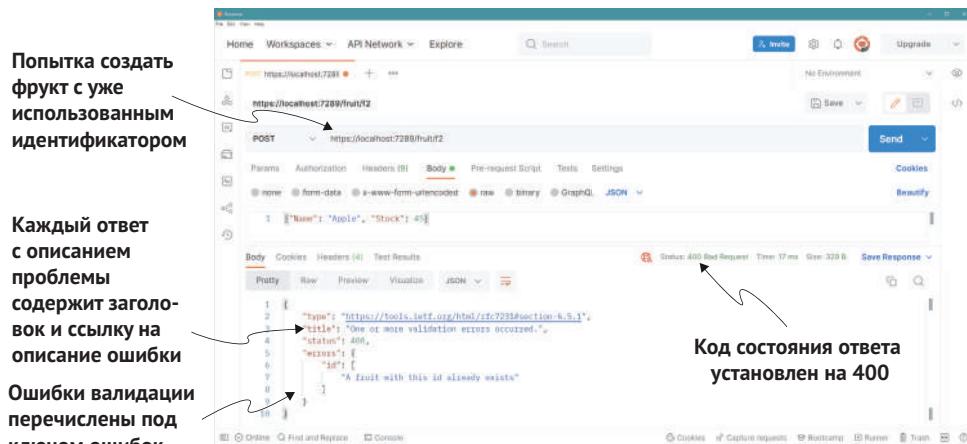


Рис. 5.5 Можно вернуть ответ Problem Details, используя методы *Problem* и *ValidationProblem*. Показанный здесь ответ *ValidationProblem* включает описание ошибки, а также ошибки валидации в стандартном формате. В этом примере показан ответ при попытке создать фрукт с уже использованным идентификатором

Выбор формата ошибки – важный шаг при создании API, а поскольку Problem Details уже являются веб-стандартом, это должен быть ваш основной подход, особенно в случае ошибок валидации. Далее вы узнаете, как убедиться, что все ваши ответы об ошибках – это ответы Problem Details.

5.3.3 Преобразование всех ответов в *Problem Details*

В разделе 5.3.2 было показано, как использовать методы *Results.Problem()* и *Results.ValidationProblem()* в конечных точках минимальных API для возврата сведений о проблеме в формате JSON. Единственная загвоздка состоит в том, что конечные точки минимальных API – не единственное, что может генерировать ошибки. В этом разделе вы узнаете, как убедиться, что все ваши ошибки возвращают сведения о проблеме в формате JSON, сохраняя согласованность ответов об ошибках во всем приложении.

Приложение с минимальным API может генерировать ответ об ошибке несколькими способами:

- возврат кода состояния ошибки из обработчика конечной точки;
- выброс исключения в обработчике конечной точки, которое перехватывается;
- *ExceptionHandlerMiddleware* или *DeveloperExceptionPageMiddleware* преобразуется в ответ, сообщающий об ошибке;
- конвейер промежуточного ПО возвращает ответ с кодом состояния 404, поскольку запрос не обрабатывается конечной точкой;
- компонент промежуточного ПО в конвейере выбрасывает исключение;
- компонент промежуточного ПО, возвращающий ответ, сообщающий об ошибке, поскольку запрос требует аутентификации, а учетные данные не были предоставлены.

По сути, существует два класса ошибок, которые обрабатываются по-разному: исключения и ответы с кодом состояния ошибки. Чтобы создать согласованный API для потребителей, необходимо убедиться, что оба типа ошибок возвращают в ответе код в формате JSON, соответствующий веб-спецификации Problem Details.

Преобразование исключений в Problem Details

В главе 4 вы узнали, как обрабатывать исключения с помощью `ExceptionHandlerMiddleware`. Вы видели, что этот компонент перехватывает любые исключения из более позднего компонента и генерирует ответ об ошибке, выполняя путь обработки ошибок. Можно добавить компонент в свой конвейер, указав путь обработки ошибок `"/error"`:

```
app.UseExceptionHandler("/error");
```

`ExceptionHandlerMiddleware` вызывает этот путь после захвата исключения для создания окончательного ответа. Проблема с этим подходом для минимальных API заключается в том, что нам нужна выделенная конечная точка ошибки, единственной целью которой является создание ответа Problem Details.

К счастью, в .NET 7 можно настроить `ExceptionHandlerMiddleware` (и `DeveloperExceptionPageMiddleware`) для автоматического преобразования исключения в ответ с описанием проблемы. В .NET 7 можно добавить новый `IProblemDetailsService` в приложение, вызвав метод `AddProblemDetails()`. Когда `ExceptionHandlerMiddleware` настроен без пути обработки ошибок, он автоматически использует `IProblemDetailsService` для генерации ответа, как показано на рис. 5.6.

ПРЕДУПРЕЖДЕНИЕ Вызов метода `AddProblemDetails()` регистрирует сервис `IProblemDetailsService` в контейнере внедрения зависимостей, чтобы другие сервисы и промежуточное ПО могли использовать ее. Если вы сконфигурируете `ExceptionHandlerMiddleware` без пути обработки ошибок, но забудете вызвать метод `AddProblemDetails()`, то получите исключение при запуске приложения. Подробнее о внедрении зависимостей вы узнаете в главах 8 и 9.

В листинге 5.6 показано, как настроить генерацию Problem Details в обработчиках исключений. Добавьте в свое приложение необходимый сервис `IProblemDetailsService` и вызовите метод `UseExceptionHandler()`, не предоставив путь обработки ошибок, а промежуточное ПО автоматически сгенерирует ответ с описанием проблемы, когда перехватит исключение.

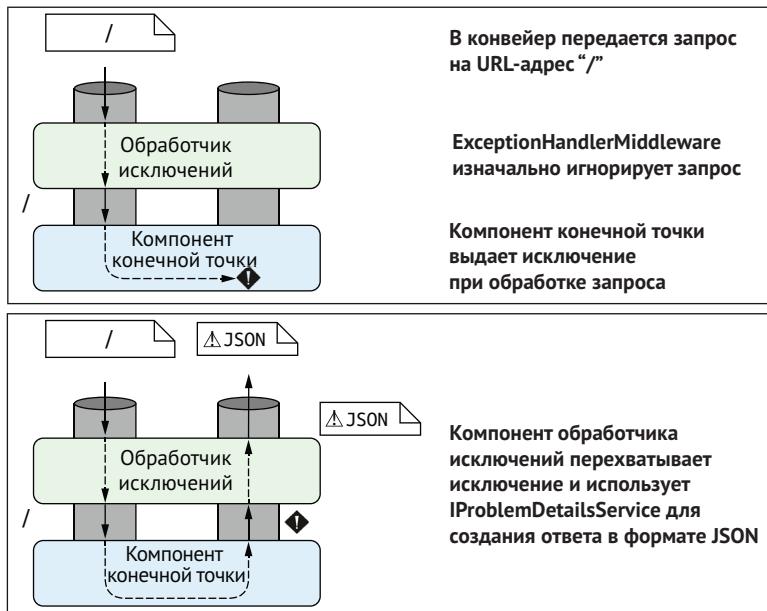


Рис. 5.6 ExceptionHandlerMiddleware перехватывает исключения, которые возникают позже в конвейере промежуточного ПО. Если промежуточное ПО не настроено для повторного выполнения конвейера, оно генерирует ответ с описанием проблемы с помощью IProblemDetailsService

Листинг 5.6. Настройка ExceptionHandlerMiddleware для использования Problem Details

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails(); <-- Добавляет реализацию IProblemDetailsService
WebApplication app = builder.Build();
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler(); <-- Настраивает ExceptionHandlerMiddleware без пути, чтобы он использовал IProblemDetailsService
}
app.MapGet("/", void () => throw new Exception()); <-- Выбрасывает исключение для демонстрации поведения
app.Run();
```

Как обсуждалось в главе 4, *WebApplication* автоматически добавляет *DeveloperExceptionPageMiddleware* в приложение в окружении разработки. Этот компонент промежуточного ПО аналогичным образом поддерживает возврат ответа *Problem Details* при выполнении двух условий:

- вы зарегистрировали *IProblemDetailsService* в приложении (вызвав метод *AddProblemDetails()* в файле *Program.cs*);
- в запросе указано, что он не поддерживает HTML. Если клиент поддерживает HTML, то вместо этого компонент использует страницу исключений разработчика из главы 4.

`ExceptionHandlerMiddleware` и `DeveloperExceptionMiddleware` позаботятся о преобразовании всех ваших исключений в ответы `Problem Details`, но вам все равно нужно подумать об ошибках, не связанных с исключениями, таких как автоматический ответ с кодом состояния `404`, генерируемый, когда запрос не соответствует ни одной конечной точке.

Преобразование кодов состояния ошибок в формат `Problem Details`

Возврат кодов состояния ошибок – это распространенный способ сообщить об ошибках клиенту при использовании минимальных API. Чтобы обеспечить согласованность API для потребителей, следует возвращать ответ `Problem Details` каждый раз, когда вы возвращаете ошибку. К сожалению, как уже говорилось, мы не контролируем все места, где может появиться код ошибки.

Конвейер промежуточного ПО автоматически возвращает ответ с кодом состояния `404`, когда, например, несовпадающий запрос достигает конца конвейера.

Вместо генерации ответа `Problem Details` в обработчиках конечных точек можно добавить компонент для автоматического преобразования ответов в формат `Problem Details` с помощью `StatusCodePagesMiddleware`, как показано на рис. 5.7. У любого ответа, достигающего компонента промежуточного ПО с кодом состояния ошибки и у которого еще нет тела ответа, компонентом добавляется тело ответа `Problem Details`. Этот компонент автоматически преобразует все ответы об ошибках, независимо от того, были ли они сгенерированы конечной точкой или другим компонентом.

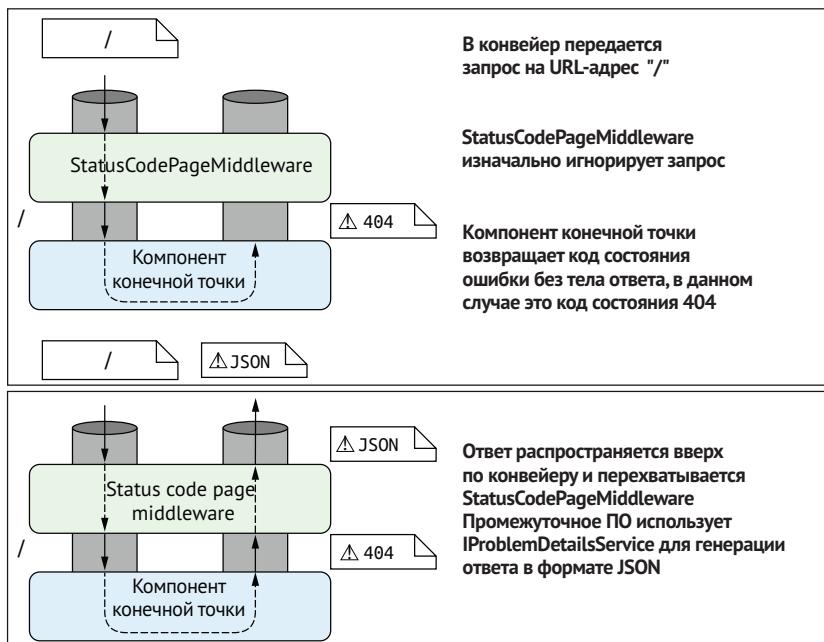


Рис. 5.7. `Status Code Page middleware` перехватывает ответы с кодом состояния ошибки, которые не имеют тела ответа, и добавляет тело ответа `Problem Details`

ПРИМЕЧАНИЕ Вы также можете использовать *StatusCodesMiddleware* для повторного выполнения конвейера промежуточного ПО путем обработки ошибок, как в случае с *ExceptionHandlerMiddleware* (глава 4). Этот метод наиболее полезен для приложений *Razor Pages*, когда вам нужны разные страницы ошибок для определенных кодов состояния, как вы увидите в главе 15.

Добавьте *StatusCodesMiddleware* в свое приложение с помощью метода расширения *UseStatusCodePages()*, как показано в следующем листинге. Убедитесь, что вы также добавили *IProblemDetailsService* с помощью метода *AddProblemDetails()*.

Листинг 5.7. Использование *StatusCodesMiddleware* для возврата ответа *Problem Details*

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails(); <-- Добавляет реализацию
WebApplication app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler();
}

app.UseStatusCodePages(); <-- Добавляет
                           StatusCodePagesMiddleware

app.MapGet("/", () => Results.NotFound()); <-- StatusCodePagesMiddleware
                                                автоматически добавляет
                                                тело Problem Details в ответ
                                                с кодом состояния 404

app.Run();
```

StatusCodesMiddleware в сочетании с компонентом для обработки исключений гарантирует, что ваш API возвращает ответ *Problem Details* для всех ответов об ошибках.

СОВЕТ Вы также можете настроить способ генерации ответа *Problem Details*, передав параметры методу *AddProblemDetails()* или реализовав собственный *IProblemDetailsService*.

До сих пор в разделе 5.3 я описывал возврат объектов в формате *JSON*, возврат строки в виде текста, а также возврат собственных кодов состояния и ответов *Problem Details* с помощью класса *Results*. Однако иногда нам нужно вернуть нечто покрупнее, например обычный или двоичный файл. К счастью, для этой задачи также можно использовать удобный класс *Results*.

Методы классов *Results* и *TypedResults* – это удобные способы возврата распространенных ответов, поэтому вполне естественно, что они содержат вспомогательные методы для других распространенных ситуаций, таких как возврат файла или двоичных данных:

- `Results.File()` – укажите путь к возвращаемому файлу, и ASP.NET Core позаботится о его потоковой передаче клиенту;
- `Results.Byte()` – для возврата двоичных данных можно передать этому методу `byte[]` для возврата;
- `Results.Stream()` – можно отправлять данные клиенту асинхронно, используя поток.

В каждом из этих случаев можно указать заголовок `Content-Type` для данных и имя файла, которое будет использоваться клиентом. Браузеры предлагают сохранять файлы двоичных данных, используя предложенное имя файла. Методы `File` и `Byte` даже поддерживают запросы с указанием диапазона данных, если указать для `EnableRangeProcessing` значение `true`.

ОПРЕДЕЛЕНИЕ Клиенты могут создавать запросы с указанием диапазона данных, используя заголовок `Range`, чтобы запрашивать у сервера определенный диапазон байтов, а не весь файл, что снижает требуемую для запроса пропускную способность. Если запросы с указанием диапазона активированы для `Results.File()` или `Results.Byte()`, ASP.NET Core автоматически генерирует соответствующий ответ. Подробнее о запросах с указанием диапазона можно прочитать на странице <http://mng.bz/Wzd0>.

Если встроенные вспомогательные методы `Results` не предоставляют необходимую функциональность, всегда можно вернуться к созданию ответа вручную, как показано в листинге 5.4. Если вы обнаружите, что создаете один и тот же ответ вручную несколько раз, то можно рассмотреть возможность создания собственного типа `IResult` для инкапсуляции этой логики. В этом посте я показываю, как создать собственный `IResult`, который возвращает XML и регистрирует его как расширение: <http://mng.bz/8rNP>.

5.4 Запуск общего кода с фильтрами конечных точек

В разделе 5.3 вы узнали, как использовать класс `Results` для возврата различных ответов, когда запрос не является валидным. Мы рассмотрим валидацию более подробно в главе 7, а в этом разделе вы узнаете, как применять фильтры для извлечения общего кода, который выполняется до (или после) выполнения конечной точки.

Начнем с добавления дополнительной валидации в API `fruit` из листинга 5.5. В следующем листинге мы добавим дополнительную валидацию к конечной точке `MapGet`, чтобы убедиться, что предоставленный идентификатор не пуст и начинается с буквы `f`.

Листинг 5.8. Добавление базовой валидации к конечным точкам минимального API

```
using System.Collections.Concurrent;
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
```

```

var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
{
    >> if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
    {
       Добавляет до-  
полнительную  
валидацию,  
чтобы удосто-  
вериться, что  
предостав-  
ленный иден-  
тификатор  
имеет требуе-  
мый формат
    }
    return Results.ValidationProblem(new Dictionary<string, string>[]
    {
        {"id", new[] {"Invalid format. Id must start with 'f'"}}
    });
}

Добавляет до-  
полнительную  
валидацию,  
чтобы удосто-  
вериться, что  
предостав-  
ленный иден-  
тификатор  
имеет требуе-  
мый формат
    return _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404);
});

app.Run()

```

Несмотря на то что это базовая валидация, она начинает загромождать наш обработчик конечной точки, затрудняя чтение того, что делает конечная точка. В качестве одного из улучшений можно было бы переместить код валидации во вспомогательную функцию. Также обработчики конечных точек могут содержать в себе код для вызова методов, которые не связаны с основной функцией конечной точки.

ПРИМЕЧАНИЕ Дополнительные шаблоны валидации подробно обсуждаются в главе 7.

Обычно для каждой конечной точки выполняются различные сквозные действия. Я уже упоминал о валидации, но, кроме этого, другие сквозные действия могут включать в себя журналирование, авторизацию и аутентификацию. В ASP.NET Core имеетсястроенная поддержка некоторых из этих функций, например авторизации (глава 24), но, скорее всего, у вас есть некий общий код, который не вписывается в определенные категории валидации или авторизации.

К счастью, ASP.NET Core в минимальных API включает в себя функцию для решения этих второстепенных задач, это фильтры конечных точек. Можно указать фильтр для конечной точки, вызвав метод `AddEndpointFilter()` для результата вызова метода `MapGet` (или аналогичного метода), и передать функцию для выполнения. Вы даже можете добавить несколько вызовов метода `AddEndpointFilter()`, который создает конвейер фильтрации конечных точек, аналогичный конвейеру промежуточного ПО. На рис. 5.8 показано, что конвейер функционально идентичен конвейеру промежуточного ПО на рис. 4.3.

Каждый фильтр конечной точки имеет два параметра: параметр `context`, который предоставляет сведения о выбранном обработчике конечной точки, и параметр `next`, обозначающий конвейер фильтра. Когда вы вызываете метод, подобный параметру `next`, путем вызова `next(context)`, вы вызываете оставшуюся часть конвейера фильтра. Если в конвейере больше нет фильтров, вызывается обработчик конечной точки, как показано на рис. 5.8.

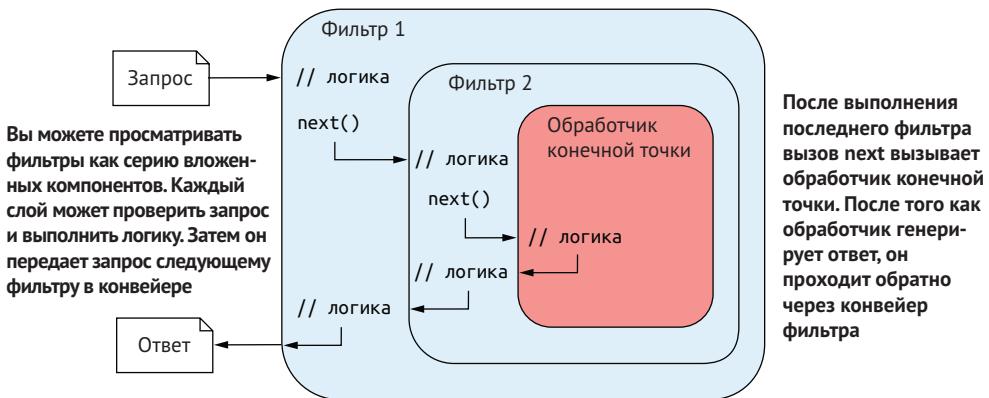


Рис. 5.8 Конвейер фильтра конечных точек. Фильтры выполняют код, а затем вызывают `next(context)` для вызова следующего фильтра в конвейере. Если в конвейере больше нет фильтров, вызывается обработчик конечной точки. После выполнения обработчика фильтры могут запустить дальнейший код

В листинге 5.9 показано, как запустить ту же логику валидации, которую вы видели в листинге 5.8, используя фильтр конечных точек. Функция фильтра получает доступ к аргументам метода конечной точки с помощью функции `context.GetArgument<T>()`, передавая позицию; 0 – это первый аргумент обработчика конечной точки, 1 – второй аргумент и т. д. Если аргумент не является валидным, функция фильтра возвращает ответ в виде объекта `IResult`. Если аргумент валиден, то фильтр вызывает `await next(context)`, выполняя обработчик конечной точки.

Листинг 5.9. Использование AddEndpointFilter для извлечения общего кода

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .AddEndpointFilter(ValidationHelper.ValidateId); <-- Добавляет фильтр в конечную точку с помощью AddEndpointFilter

app.Run();

class ValidationHelper
{
    ▷ internal static async ValueTask<object?> ValidateId(
        EndpointFilterInvocationContext context,
        EndpointFilterDelegate next) <-- context представляет аргументы метода конечной точки и HttpContext
    {
        Mетод должен возвращать ValueTask
    }
} <-- next представляет метод фильтра (или конечную точку), который будет вызываться следующим
```

```

Вы можете  
получить аргу-  
менты метода  
из контекста ↗ var id = context.GetArgument<string>(0);
if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
{
    return Results.ValidationProblem(
        new Dictionary<string, string[]>
    {
        {"id", new[]{"Invalid format. Id must start with 'f'"}}
    });
}
return await next(context); ← Вызов next выполняет оставшиеся  
фильтры в конвейере
}
}

```

ПРИМЕЧАНИЕ `EndpointFilterDelegate` – это именованный тип делегата. По сути, это `Func<EndpointFilterInvocationContext, ValueTask<object?>>`.

Между конвейером промежуточного ПО и конвейером конечной точки фильтра существует множество параллелей. Мы рассмотрим их в разделе 5.4.1.

5.4.1 Добавление нескольких фильтров к конечной точке

Конвейер промежуточного ПО обычно является лучшим местом для решения таких сквозных задач, как журналирование, аутентификация и авторизация, поскольку эти функции применяются ко всем запросам. Тем не менее, как уже обсуждалось, нередко возникают дополнительные сквозные задачи для конкретных конечных точек. Если вам нужно много операций для этих конечных точек, то можно рассмотреть возможность использования нескольких фильтров.

Как было показано на рис. 5.8, при добавлении нескольких фильтров к конечной точке создается конвейер. Как и конвейер промежуточного ПО, конвейер фильтров конечных точек может выполнять код как до, так и после выполнения остальной части конвейера. Аналогично конвейер фильтров можно замкнуть так же, как и конвейер промежуточного ПО, возвращая результат и не вызывая `next`.

ПРИМЕЧАНИЕ Вы уже видели пример замыкания в конвейере фильтров. В листинге 5.9 мы замыкаем конвейер, если идентификатор недействителен, возвращая объект `Problem Details` вместо вызова `next(context)`.

Как и в случае с промежуточным ПО, важен порядок добавления фильтров в конвейер фильтров конечной точки. Фильтры, которые вы добавляете первыми, вызываются первыми в конвейере, а фильтры, которые вы добавляете последними, вызываются последними. На обратном пути по конвейеру, после вызова обработчика конечной точки, фильтры вызываются в обратном порядке, как и в конвейере промежуточного ПО. В качестве примера рассмотрим следующий листинг, где дополнительный фильтр добавляется к конечной точке, показанной в листинге 5.9.

Листинг 5.10. Добавление нескольких фильтров в конвейер фильтров конечных точек

```

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .AddEndpointFilter(ValidationHelper.ValidateId) <-->
    .AddEndpointFilter(async (context, next) =>
    {
        app.Logger.LogInformation("Executing filter..."); <-->
        object? result = await next(context);
        app.Logger.LogInformation($"Handler result: {result}"); <-->
        return result;
    });
}

app.Run(); <--> Возвращает результат
                    без изменений

```

Добавляет новый фильтр с использованием лямбда-функции

Выполняет оставшуюся часть конвейера и обработчик конечной точки

Добавляет фильтр валидации, как и раньше

Регистрирует сообщение перед выполнением остальной части конвейера

Регистрирует результат, возвращаемый остальной частью конвейера

Дополнительный фильтр реализован как лямбда-функция и просто записывает сообщение журнала при выполнении. Затем он запускает остальную часть конвейера фильтров (который в данном примере содержит только обработчик конечной точки) и регистрирует результат, возвращенный конвейером. О журналировании подробно рассказывается в главе 26. В этом примере мы рассмотрим журналы, которые записываются в консоль.

На рис. 5.9 показаны сообщения журнала, записанные при отправке двух запросов к API из листинга 5.10. Первый запрос предназначен для существующей записи, поэтому он возвращает результат 200 OK. Второй запрос использует неверный формат идентификатора, поэтому первый фильтр отклоняет его. На рис. 5.9 показано, что в этом случае не запускается ни второй фильтр, ни обработчик конечной точки; конвейер фильтров замыкается.

Добавляя вызовы `AddEndpointFilter`, можно создавать конвейеры фильтров конечных точек произвольного размера, но тот факт, что вы можете это сделать, не означает, что вам следует так поступать. Перемещение кода в фильтры может уменьшить беспорядок в конечных точках, но затрудняет понимание потока приложения. Я предлагаю вам избегать использования фильтров, если вы не обнаружите дублированный код в нескольких конечных точках, а затем отдавать предпочтение фильтру вместо простого вызова метода только в том случае, если он значительно упрощает требуемый код.

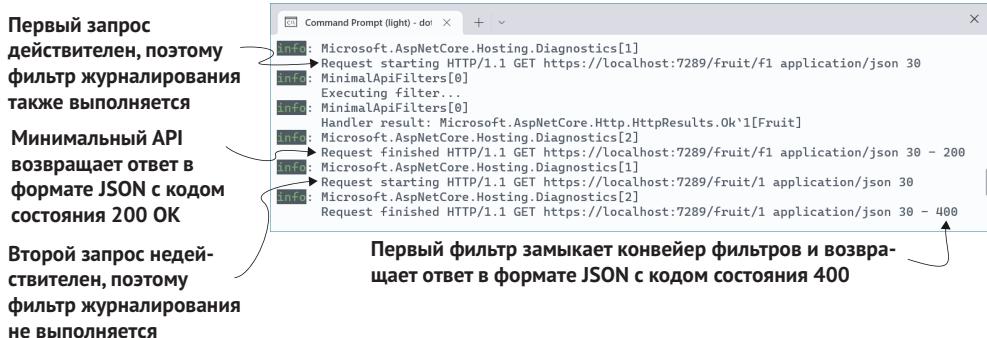


Рис. 5.9 Отправка двух запросов к API из листинга 5.10. Первый запрос действителен, поэтому выполняются оба фильтра. Во втором запросе указан недопустимый идентификатор, поэтому первый фильтр замыкает запросы, и второй фильтр не выполняется

5.4.2 Фильтры или промежуточное ПО: что выбрать?

Конвейер фильтров конечных точек во многом похож на конвейер промежуточного ПО, но при принятии решения о том, какой подход использовать, следует учитывать несколько тонкостей. Сходства включают в себя три основные параллели:

- *на входе запросы проходят через компонент промежуточного ПО, а на выходе ответы проходят снова.* Аналогично фильтры конечных точек могут запускать код перед вызовом следующего фильтра в конвейере и могут запускать код после генерации ответа, как показано на рис. 5.8;
- *компонент может замкнуть запрос, возвращая ответ, вместо того чтобы передавать его более позднему компоненту.* Фильтры также могут замыкать конвейер фильтров, возвращая ответ;
- *промежуточное ПО часто используется для решения сквозных задач приложений, таких как журналирование, профилирование производительности и обработка исключений.* Фильтры также подходят для решения сквозных задач.

Напротив, между промежуточным ПО и фильтрами есть три основных различия:

- промежуточное ПО может выполнятся для всех запросов; фильтры будут выполнятся только для запросов, которые достигают компонента `EndpointMiddleware` и вызывают связанную конечную точку;
- фильтры имеют доступ к дополнительным сведениям о конечной точке, которая будет выполняться, например к возвращаемому значению конечной точки, например `IResult`. Промежуточное программное обеспечение обычно не видит эти промежуточные шаги, поэтому оно видит только сгенерированный ответ;
- фильтры можно легко ограничить подмножеством запросов, например одной конечной точкой или группой конечных точек. Промежуточное ПО обычно применяется ко всем запросам (чего-то подобного можно добиться с помощью собственных компонентов промежуточного ПО).

Все это хорошо, но как учитывать эти различия? Когда следует предпочесть одно другому?

Мне нравится рассматривать промежуточное ПО и фильтры как вопрос специфики. Промежуточное ПО – это более общая концепция, работающая с примитивами более низкого уровня, такими как `HttpContext`, поэтому у нее более широкий охват. Если необходимая вам функциональность не имеет требований к конечной точке, следует использовать компонент промежуточного ПО. Отличный пример – обработка исключений. Исключения могут возникнуть в любом месте приложения, и необходимо их обрабатывать, поэтому использование компонента для обработки исключений не лишено смысла.

С другой стороны, если вам *все же* нужен доступ к сведениям о конечной точке или нужно иное поведение для некоторых запросов, следует рассмотреть возможность использования фильтра. Хороший пример – валидация. Не все запросы нуждаются в одинаковой валидации. Например, запросы к статическим файлам не требуют валидации параметров, в отличие от запросов к конечной точке API. В этом случае имеет смысл применить валидацию к конечным точкам с помощью фильтров.

СОВЕТ Там, где это возможно, рассмотрите возможность использования промежуточного ПО для решения сквозных задач. Используйте фильтры, когда вам нужно разное поведение для разных конечных точек или когда функциональность зависит от концепций конечных точек, таких как объекты `IResult`.

До сих пор фильтры, которые мы рассматривали, предназначались для одной конечной точки. В разделе 5.4.3 мы рассмотрим создание обобщенных фильтров, которые можно применять к нескольким конечным точкам.

5.4.3 Обобщенные фильтры конечных точек

Одна из распространенных проблем с фильтрами заключается в том, что они тесно связаны с реализацией обработчиков конечных точек. Например, в листинге 5.9 предполагается, что параметр `id` является первым параметром метода. В этом разделе вы узнаете, как создавать обобщенные версии фильтров, которые работают с несколькими обработчиками конечных точек.

API `fruit`, с которым мы работали в этой главе, содержит несколько обработчиков конечных точек, принимающих несколько параметров. Например, обработчик `MapPost` принимает параметры `string id` и `Fruit fruit`:

```
app.MapPost("/fruit/{id}", (string id, Fruit fruit) => { /* */ });
```

В данном примере параметр `id` указан первым, но это не обязательно. Параметры обработчика можно поменять местами, и конечная точка будет функционально идентична:

```
app.MapPost("/fruit/{id}", (Fruit fruit, string id) => { /* */ });
```

К сожалению, при таком порядке фильтр `ValidateId`, описанный в листинге 5.9, работать не будет. Данный фильтр предполагает, что первым параметром обработчика является `id`, но в нашей обновленной реализации `MapPost` это не так.

ASP.NET Core предоставляет решение, использующее для фильтров паттерн «Фабрика». Фабрику фильтров можно зарегистрировать с помощью метода `AddEndpointFilterFactory()`. *Фабрика фильтров* – это метод, возвращающий *функцию фильтра*. ASP.NET Core запускает фабрику фильтров при создании приложения и включает возвращенный фильтр в конвейер фильтров для приложения, как показано на рис. 5.10. Вы можете использовать одну и ту же такую функцию для создания разных фильтров для каждой конечной точки, при этом каждый фильтр будет адаптирован к параметрам конечной точки.

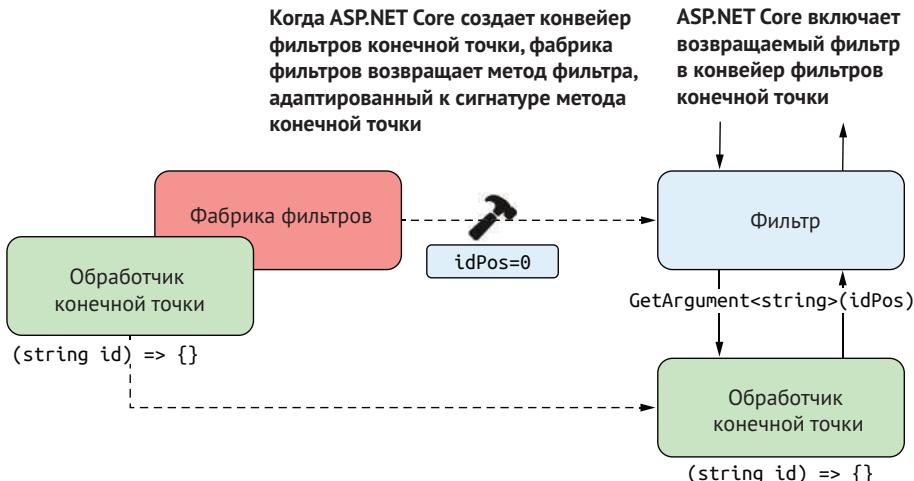


Рис. 5.10 Фабрика фильтров – это универсальный способ добавления фильтров конечных точек. Фабрика считывает сведения о конечной точке, например сигнатуру ее метода, и создает функцию фильтра. Эта функция включается в конвейер финального фильтра для конечной точки. Шаг сборки означает, что одна фабрика фильтров может создавать фильтры для нескольких конечных точек с разными сигнатурами методов

В листинге 5.11 показан пример использования паттерна «Фабрика» на практике. Фабрика фильтров применяется к нескольким конечным точкам. Для каждой конечной точки фабрика сначала проверяет параметр `id`; если его не существует, фабрика возвращает `next` и не добавляет фильтр в конвейер. Если параметр `id` существует, фабрика возвращает функцию фильтра, которая практически идентична функции фильтра из листинга 5.9. Основное отличие состоит в том, что данный фильтр обрабатывает переменное расположение параметра `id`.

Листинг 5.11. Использование фабрики фильтров для создания фильтра конечной точки

```

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .AddEndpointFilterFactory(ValidationHelper.ValidateIdFactory);      ← Фабрика фильтров может
                                                                     обрабатывать конечные
                                                                     точки с различными сиг-
                                                                     натурами методов

app.MapPost("/fruit/{id}", (Fruit fruit, string id) =>
    _fruit.TryAdd(id, fruit)
        ? TypedResults.Created($"/fruit/{id}", fruit)
        : Results.ValidationProblem(new Dictionary<string, string[]>
    {
        { "id", new[] { "A fruit with this id already exists" } }
    })
    .AddEndpointFilterFactory(ValidationHelper.ValidateIdFactory);      ←

app.Run();
```

Параметр context предоставляет подробную информацию о методе обработчика конечной точки

Если параметр id существует, возвращаем функцию фильтра (фильтр, выполняемый для конечной точки)

```

class ValidationHelper
{
    internal static EndpointFilterDelegate ValidateIdFactory(
        EndpointFilterFactoryContext context,
        EndpointFilterDelegate next)
    {
        ParameterInfo[] parameters =
            context.MethodInfo.GetParameters();
        int? idPosition = null;
        for (int i = 0; i < parameters.Length; i++)
        {
            if (parameters[i].Name == "id" &&
                parameters[i].ParameterType == typeof(string))
            {
                idPosition = i;      ← Перебирает параметры, что-
                                     бы найти параметр строку-
                                     идентификатор и запомнить
                                     его позицию
            }
        }

        if (!idPosition.HasValue)
        {
            return next;      ← Если параметр id не найден, фильтр не
                           добавляется, а возвращает оставшуюся
                           часть конвейера
        }

        return async (invocationContext) =>
        {
```

Метод GetParameters() предоставляет подробную информацию о параметрах вызываемого обработчика

```

        var id = invocationContext
            .GetArgument<string>(idPosition.Value);
        if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
        {
            return Results.ValidationProblem(
                new Dictionary<string, string[]>
                {{ "id", new[] { "Id must start with 'f'" }}}));
        }
    } > return await next(invocationContext);
}
}

```

Если идентификатор валидный, выполняется следующий фильтр в конвейере

Если идентификатор не является валидным, возвращаем результат Problem Details

Код в листинге 5.11 сложнее всего, что мы видели до сих пор, поскольку он имеет дополнительный уровень абстракции. Компонент конечной точки передает объект `EndpointFilterFactoryContext` в фабричную функцию, которая содержит дополнительные сведения о конечной точке по сравнению с контекстом, передаваемым в обычную функцию фильтра. В частности, он включает свойства `MethodInfo` и `EndpointMetadata`.

ПРИМЕЧАНИЕ О метаданных конечных точек рассказывается в главе 6.

Свойство `MethodInfo` можно использовать для управления созданием фильтра на основе определения обработчика конечной точки. В листинге 5.11 показано, как перебирать параметры, чтобы проверить нужные сведения (в данном случае параметр строки-идентификатора) и настроить возвращаемую функцию фильтра.

Если вы считаете, что все эти сигнатуры методов сбивают вас с толку, я вас не виню. Запомнить разницу между `EndpointFilterFactoryContext` и `EndpointFilterInvocationContext`, а затем пытаться удовлетворить компилятор с помощью лямбда-методов – все это может раздражать. Иногда хочется реализовать старый добрый интерфейс. Сделаем это сейчас.

5.4.4 Реализация интерфейса `IEndpointFilter`

Создание корректного лямбда-выражения для метода `AddEndpointFilter()` может оказаться утомительным занятием в зависимости от уровня поддержки, которую обеспечивает ваша интегрированная среда разработки (IDE). В этом разделе вы узнаете, как обойти данную проблему, определив вместо этого класс, реализующий `IEndpointFilter`.

Можно реализовать `IEndpointFilter`, определив класс с методом `InvokeAsync()`, который имеет ту же сигнатуру, что и лямбда-выражение, определенное в листинге 5.9. Преимущество использования `IEndpointFilter` заключается в том, что вы получаете `IntelliSense` и автодополнение для сигнатуры метода. В следующем листинге показано, как реализовать класс `IEndpointFilter`, эквивалентный листингу 5.9.

Листинг 5.12. Реализация интерфейса IEndpointFilter

```

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .AddEndpointFilter<IdValidationFilter>(); // Добавляет фильтр
app.Run(); // Фильтр должен реализовать
           // IEndpointFilter...
class IdValidationFilter : IEndpointFilter // ...
{
    public async ValueTask<object?> InvokeAsync(
        EndpointFilterInvocationContext context,
        EndpointFilterDelegate next) // ...который требует
                                      // реализации одного
                                      // метода
    {
        var id = context.GetArgument<string>(0);
        if (string.IsNullOrEmpty(id) || !id.StartsWith('f'))
        {
            return Results.ValidationProblem(
                new Dictionary<string, string[]>
                {
                    {"id", new[] {"Invalid format. Id must start with 'f'"}}
                });
        }
        return await next(context);
    }
}

```

Реализация `IEndpointFilter` – хороший вариант, когда ваши фильтры становятся более сложными, но учтите, что для паттерна «Фабрика фильтров», показанного в разделе 5.4.3, не существует эквивалентного интерфейса. Если вы хотите использовать обобщенные фильтры с помощью фабрики фильтров, вам придется придерживаться подхода с лямбда-функцией (или вспомогательным методом), показанного в листинге 5.11.

5.5 Организация API с помощью групп маршрутов

Одно из критических замечаний в адрес минимальных API в .NET 6 заключалось в том, что они были довольно громоздкими, требовали большого количества дублированного кода и часто приводили к громоздким методам обработчика конечных точек. В .NET 7 представлены два новых механизма с целью избавиться от этой критики:

- **фильтры** – представленные в разделе 5.4 фильтры помогают отделить валидацию и сквозные функции, например журналирование, от важной логики в функциях обработчика конечной точки;
- **группы маршрутов** – описанные в этом разделе группы маршрутов помогают уменьшить дублирование за счет применения фильтров и маршрутизации к нескольким обработчикам одновременно.

При проектировании API важно поддерживать согласованность маршрутов, которые используются для конечных точек, что часто означает дублирование части шаблона маршрута между несколькими API. Например, все конечные точки API `fruit`, описанные в этой главе (например, в листинге 5.3), начинаются с префикса маршрута `/fruit`:

- `MapGet("/fruit", () => {/* */});`
- `MapGet("/fruit/{id}", (string id) => {/* */});`
- `MapPost("/fruit/{id}", (Fruit fruit, string id) => {/* */});`
- `MapPut("/fruit/{id}", (Fruit fruit, string id) => {/* */});`
- `MapDelete("/fruit/{id}", (string id) => {/* */});`

Кроме того, последние четыре конечные точки должны проверить параметр `id`. Эту проверку можно извлечь во вспомогательный метод и применить в качестве фильтра, но все равно нужно *не забыть* применить фильтр при добавлении новой конечной точки.

Все это дублирование можно устраниТЬ с помощью групп маршрутов. Можно использовать группы маршрутов для извлечения общих сегментов пути или фильтров в одно место, уменьшая дублирование в определениях конечных точек. Группа маршрутов создается путем вызова `MapGroup("/fruit")` экземпляра `WebApplication`, предоставляя префикс маршрута для группы (`/fruit` в данном случае), а метод `MapGroup()` возвращает `RouteGroupBuilder`.

Если у вас есть `RouteGroupBuilder`, вы можете вызывать те же методы расширения `Map*` класса `RouteGroupBuilder`, что и в случае с `WebApplication`. Единственное отличие состоит в том, что все конечные точки, которые вы определяете в группе, будут иметь префикс `/fruit`, примененный к каждой конечной точке, которую вы определяете, как показано на рис. 5.11. Аналогичным образом можно вызвать метод `AddEndpointFilter()` для группы маршрутов, и все конечные точки группы также будут использовать этот фильтр.

Вы можете создать группу маршрутов, вызвав метод `MapGroup` класса `WebApplication` и указав префикс маршрута

`MapGroup("/fruit")`

Вы можете создавать конечные точки непосредственно в `RouteGroupBuilder`

- `MapGet("/", () => {})`
- `MapGet("/{id}", () => {})`
- `MapPost("/{id}", () => {})`

Фильтр

Фильтр

MapGet("/fruit/", () => {})

Фильтр

MapGet("/fruit/{id}", () => {})

Фильтр

MapPost("/fruit/{id}", () => {})

Каждая конечная точка в группе маршрутов наследует префикс маршрута в своем конечном шаблоне, а также любые фильтры, добавленные в группу маршрутов

Рис. 5.11 Использование групп маршрутов для упрощения определения конечных точек. Можно создать группу маршрутов, вызвав метод `MapGroup()` и указав префикс. Любые конечные точки, созданные в группе маршрутов, наследуют префикс шаблона маршрута, а также все фильтры, добавленные в группу

Вы даже можете создавать вложенные группы, вызывая метод `MapGroup()` для группы. Префиксы применяются к конечным точкам по порядку, поэтому первый вызов `MapGroup()` определяет префикс, используемый в начале маршрута. Например, `app.MapGroup("/fruit").MapGroup("/citrus")` будет иметь префикс `"/fruit/citrus"`.

СОВЕТ Если вы не хотите добавлять префикс, но все же хотите использовать группу маршрутов для применения фильтров, можно передать префикс `"/"` методу `MapGroup()`.

В листинге 5.13 показан пример API `fruit` с использованием групп маршрутов. Он создает `fruitApi` верхнего уровня, который применяет префикс `"/fruit"` и создает вложенную группу маршрутов `FruitApiWithValidation` для конечных точек, которым требуется фильтр. Полный пример сравнения версий с группами маршрутов и без них можно найти в исходном коде для этой главы.

Листинг 5.13. Уменьшение дублирования с помощью групп маршрутов

```
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();

RouteGroupBuilder fruitApi = app.MapGroup("/fruit"); <-- Создает группу маршрутов, вызывая MapGroup и предоставляемый префикс

fruitApi.MapGet("/", () => _fruit); <-- Конечные точки, определенные в группе маршрутов, будут иметь префикс группы, добавленный к маршруту

RouteGroupBuilder fruitApiWithValidation = fruitApi.MapGroup("/") <-- Можно создавать вложенные группы маршрутов с несколькими префиксами
    .AddEndpointFilter(ValidationHelper.ValidateIdFactory); <--

fruitApiWithValidation.MapGet("/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit) <-- Можно добавить фильтры в группу маршрутов...
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404));
fruitApiWithValidation.MapPost("/{id}", (Fruit fruit, string id) =>
    _fruit.TryAdd(id, fruit)
        ? TypedResults.Created($"/fruit/{id}", fruit)
        : Results.ValidationProblem(new Dictionary<string, string[]>
    {
        { "id", new[] { "A fruit with this id already exists" } }
    }));
}

fruitApiWithValidation.MapPut("/{id}", (string id, Fruit fruit) =>
{
    _fruit[id] = fruit;
    return Results.NoContent();
});
```

```
fruitApiWithValidation.MapDelete("/fruit/{id}", (string id) =>
{
    _fruit.TryRemove(id, out _);
    return Results.NoContent();
});

app.Run();
```

В .NET 6 минимальные API были слишком многословными, чтобы их можно было рекомендовать в целом, но с добавлением групп маршрутов и фильтров минимальные API приобрели свою самобытность. В главе 6 вы подробнее узнаете о маршрутизации и синтаксисе шаблонов маршрутов, а также о том, как создавать ссылки на другие конечные точки.

Резюме

- HTTP-методы определяют семантическое ожидание от запроса. Метод `GET` используется для извлечения данных, `POST` создает ресурс, `PUT` создает или заменяет ресурс, а `DELETE` удаляет ресурс. Следование этим соглашениям облегчит использование вашего API;
- каждый HTTP-ответ содержит код состояния. Распространенными кодами являются `200 OK`, `201 Created`, `400 Bad Request` и `404 Not Found`. Важно использовать правильный код состояния, поскольку клиенты используют эти коды для определения поведения вашего API;
- HTTP API предоставляет методы или конечные точки, которые можно использовать для доступа к данным на сервере или их изменения с использованием протокола HTTP. Обычно HTTP API вызывается мобильными приложениями или клиентской частью веб-приложений;
- конечные точки минимальных API определяются путем вызова функций `Map*` экземпляра `WebApplication`, передавая соответствующий шаблон маршрута и функцию-обработчик. Функции-обработчики запускаются в ответ на совпадающие запросы;
- для каждого HTTP-метода существуют разные методы расширения. Например, `MapGet` обрабатывает запросы методом `GET`, а `MapPost` соответствует запросам методом `POST`. Эти методы расширения используются, чтобы определить, как ваше приложение обрабатывает заданный маршрут и HTTP-метод;
- можно определить обработчики конечных точек как лямбда-выражения, переменные `Func<T, TResult>` и `Action<T>`, локальные функции, методы экземпляра или статические методы. Лучший подход зависит от сложности обработчика, а также от личных предпочтений;
- возврат `void` из обработчика конечной точки по умолчанию генерирует ответ с кодом состояния `200` без тела. Возврат строки генерирует ответ `text/plain`. Возврат экземпляра `IResult` может сгенерировать любой ответ. Любой другой объект, возвращаемый обработчиком конечной точки, сериализуется в `JSON`. Это соглашение помогает сделать обработчики конечных точек краткими;

- можно настроить ответ, внедрив объект `HttpResponse` в обработчик конечной точки, а затем задав код состояния и текст ответа. Этот подход может быть полезен, если у вас сложные требования к конечной точке;
- вспомогательные методы `Results` и `TypedResults` содержат статические методы для генерации распространенных ответов, таких как `404 Not Found` при использовании `Result.NotFound()`. Эти методы упрощают возврат распространенных кодов состояния;
- можно вернуть стандартный объект сведений о проблеме, используя `Results.Problem()` и `Results.ValidationProblem()`. По умолчанию метод `Problem()` генерирует ответ с кодом состояния `500` (который можно изменить), а метод `ValidationProblem()` генерирует ответ с кодом состояния `400` со списком ошибок валидации. Эти методы делают возврат объектов `Problem Details` более кратким, чем создание ответа вручную;
- можно использовать вспомогательные методы для `Results` для создания других распространенных типов, например `File()` для возврата файла с диска, `Bytes()` для возврата произвольных двоичных данных и `Stream()` для возврата произвольного потока;
- можно извлечь общий или дополнительный код из обработчиков конечных точек, используя фильтры конечных точек, что облегчит чтение обработчиков конечных точек;
- добавьте фильтр к конечной точке, вызвав метод `AddEndpointFilter()` и предоставив для запуска лямбда-функцию (или используйте статический метод либо метод экземпляра). Вы также можете реализовать интерфейс `IEndpointFilter` и вызвать `AddEndpointFilter<T>()`, где `T` – имя класса реализации;
- можно сделать функции фильтра более общими, создав фабрику, используя перегруженный вариант метода `AddEndpointFilter()`, который принимает `EndpointFilterFactoryContext`. Можно использовать этот подход для поддержки обработчиков конечных точек с различными сигнатурами методов;
- можно уменьшить дублирование в маршрутах конечных точек и конфигурации фильтров, используя группы маршрутов. Вызовите метод `MapGroup()` класса `WebApplication` и укажите префикс. Все конечные точки, созданные в возвращенном `RouteGroupBuilder`, будут использовать этот префикс в своих шаблонах маршрутов;
- также можно вызывать метод `AddEndpointFilter()` для групп маршрутов. Любые конечные точки, определенные в группе, тоже будут иметь фильтр, как если бы вы определили их непосредственно в конечной точке, что устраняет необходимость дублировать вызов.



Сопоставление URL-адресов с конечными точками с помощью маршрутизации

В этой главе:

- сопоставление URL-адресов с обработчиками конечных точек;
- использование ограничений и значений по умолчанию для сопоставления URL-адресов;
- генерация URL-адресов из параметров маршрута.

В главе 5 вы узнали, как определять минимальные API, как возвращать ответы и как работать с фильтрами и группами маршрутов. Одним из важнейших аспектов минимальных API, которого мы коснулись лишь слегка, является то, как ASP.NET Core выбирает конкретную конечную точку из всех определенных обработчиков на основе URL-адреса входящего запроса. Этому процессу, называемому *маршрутизацией*, и посвящена данная глава.

Данная глава начинается с определения необходимости маршрутизации и ее пользы. Вы узнаете о системе маршрутизации конечных

точек, представленной в ASP.NET Core 3.0, и почему это было сделано, познакомитесь с гибкостью маршрутизации, которую можно обеспечить для предоставляемых вами URL-адресов.

Основная часть этой главы посвящена синтаксису шаблона маршрута и тому, как его можно использовать с минимальными API. Вы узнаете о таких функциях, как необязательные параметры, параметры по умолчанию и ограничения, а также о том, как автоматически извлекать значения из URL-адреса. Хотя в данной главе мы фокусируемся на минимальных API, в Razor Pages и Model-View-Controller (MVC) используется та же система маршрутизации, что вы увидите в главе 14.

В разделе 6.4 описывается, как использовать систему маршрутизации для *генерации* URL-адресов, которые можно применять для создания ссылок и перенаправления запросов в приложении. Одним из преимуществ применения системы маршрутизации является тот факт, что она отделяет ваши обработчики от базовых URL-адресов, с которыми они связаны. Чтобы не засорять свой код жестко заданными URL-адресами, например /product/view/3, можно генерировать URL-адреса во время выполнения на основе системы маршрутизации. Это упрощает изменение URL-адреса для заданной конкретной точки. Вместо того чтобы искать, где вы использовали URL-адрес конечной точки, URL-адреса будут обновляться автоматически, без каких-либо дополнительных изменений.

К концу этой главы вы должны иметь более четкое представление о том, как работает приложение ASP.NET Core. Можно рассматривать маршрутизацию как связующий элемент, который связывает конвейер промежуточного ПО с конечными точками. Используя конвейер, конечные точки и маршрутизацию, вы будете писать веб-приложения в кратчайшие сроки!

6.1 Что такое маршрутизация?

Маршрутизация – это процесс сопоставления входящего запроса с методом, который будет его обрабатывать. Маршрутизацию можно использовать для управления URL-адресами, которые вы предоставляете в своем приложении, а также для активации таких мощных функций, как сопоставление нескольких URL-адресов с одним и тем же обработчиком и автоматическое извлечение данных из URL-адреса запроса.

В главе 4 вы видели, что приложение ASP.NET Core содержит конвейер промежуточного ПО, который определяет поведение приложения. Промежуточное ПО хорошо подходит для обработки как сквозных задач, таких как журналирование и обработка ошибок, так и узконаправленных запросов, таких как запросы изображений и файлов CSS.

Для обработки более сложной логики приложения обычно используется компонент `EndpointMiddleware` в конце конвейера промежуточного ПО, как было показано в главе 4. Он может обрабатывать соответствующий запрос, вызывая метод, известный как обработчик, и использовать его результат для генерации ответа. В предыдущих главах

описывалось использование минимальных API обработчиков конечных точек, но существуют и другие типы обработчиков, например методы действий MVC и Razor Pages, о которых вы узнаете во второй части этой книги.

Есть один аспект, который я упустил. Он заключается в том, как `EndpointMiddleware` выбирает, какой обработчик выполняется при получении запроса. Что делает запрос «уместным» для данного обработчика? Процесс сопоставления запроса с обработчиком называется *маршрутизацией*.

ОПРЕДЕЛЕНИЕ *Маршрутизация* в ASP.NET Core – это процесс выбора конкретного обработчика входящего HTTP-запроса. В минимальных API это обработчик конечной точки, связанный с маршрутом. В Razor Pages это метод обработчика страницы, определенный на странице Razor Pages. В MVC это метод действия в контроллере.

В главах 3–5 вы видели несколько простых приложений, созданных с использованием минимальных API. В главе 5 мы изучили основы маршрутизации для минимальных API, но стоит выяснить, почему маршрутная полезна, а также как ее использовать. Даже простой путь URL-адреса, например `/person`, использует маршрутную, чтобы определить, какой обработчик должен быть выполнен, как показано на рис. 6.1.

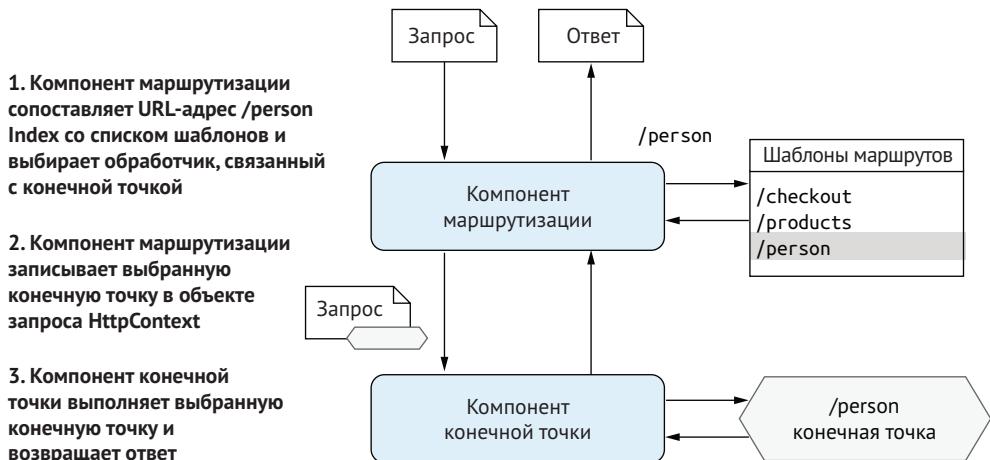


Рис. 6.1 Маршрутизатор сравнивает URL-адрес запроса со списком настроенных шаблонов маршрутов, чтобы определить, какой обработчик следует выполнить

На первый взгляд это кажется довольно простым. Вы можете спросить, зачем нужна целая глава, чтобы объяснить это очевидное сопоставление. Простота сопоставления в данном случае скрывает то, насколько мощной может быть маршрутная. Если бы этот подход, основанный на прямом сравнении со статическими строками, был единственным доступным, то вы были бы сильно ограничены в приложениях, которые можно было бы создать.

Например, рассмотрим приложение электронной коммерции, используемое для продажи разных товаров. У каждого продукта должен быть собственный URL-адрес, поэтому если бы вы использовали чисто статическую систему маршрутизации, то у вас было бы только два варианта:

- использовать *разные обработчики для каждого продукта в своем ассортименте*. Такой подход был бы неосуществим практически для любого реалистичного ассортимента продукции;
- использовать *один обработчик и строку запроса, чтобы различать продукты*. Этот подход гораздо более практичен, но в итоге вы получите не очень красивые URL-адреса, например "/product?name=big-widget" или "/product?id=12".

ОПРЕДЕЛЕНИЕ Стока запроса является частью URL-адреса, который содержит дополнительные данные, не входящие в путь. Она не используется инфраструктурой маршрутизации, чтобы определить, какой обработчик следует выполнить, но ASP.NET Core может автоматически извлекать значения из строки запроса в процессе, называемом *привязкой модели*, как вы увидите в главе 7. Стока запроса в предыдущем примере – это `id=12`.

Благодаря маршрутизации у вас может быть *один обработчик* конечной точки, который будет обрабатывать *несколько URL-адресов* без необходимости прибегать к уродливым строкам запроса. С точки зрения обработчика конечной точки подходы к строке запроса и маршрутизации очень похожи – обработчик динамически возвращает результаты для нужного продукта, если это необходимо. Разница состоит в том, что с помощью маршрутизации можно полностью настроить URL-адреса, как показано на рис. 6.2. Это дает гораздо больше гибкости и может быть важно в реальных приложениях по причинам, связанным с поисковой оптимизацией (SEO).

ПРИМЕЧАНИЕ Благодаря гибкости маршрутизации можно правильно закодировать иерархию своего сайта в URL-адресах, как описано в руководстве Google по SEO для начинающих: <http://mng.bz/EQ2J>.

Помимо активации динамических URL-адресов, маршрутизация принципиально отделяет URL-адреса в приложении от определения обработчиков.

Маршрутизация на основе файловой системы

В одной из альтернатив URL-адрес, который используется для вызова обработчика, определяется его расположением на диске. Недостатком данного подхода является тот факт, что если вы хотите изменить открытый URL-адрес, вам также необходимо изменить расположение обработчика на диске.

Этот подход может показаться странным выбором, но для некоторых приложений он имеет множество преимуществ, в первую очередь с точки зрения простоты. Как вы увидите во второй части, Razor Pages частично основан на файлах, но также использует маршрутизацию, чтобы получить лучшее от обоих миров!

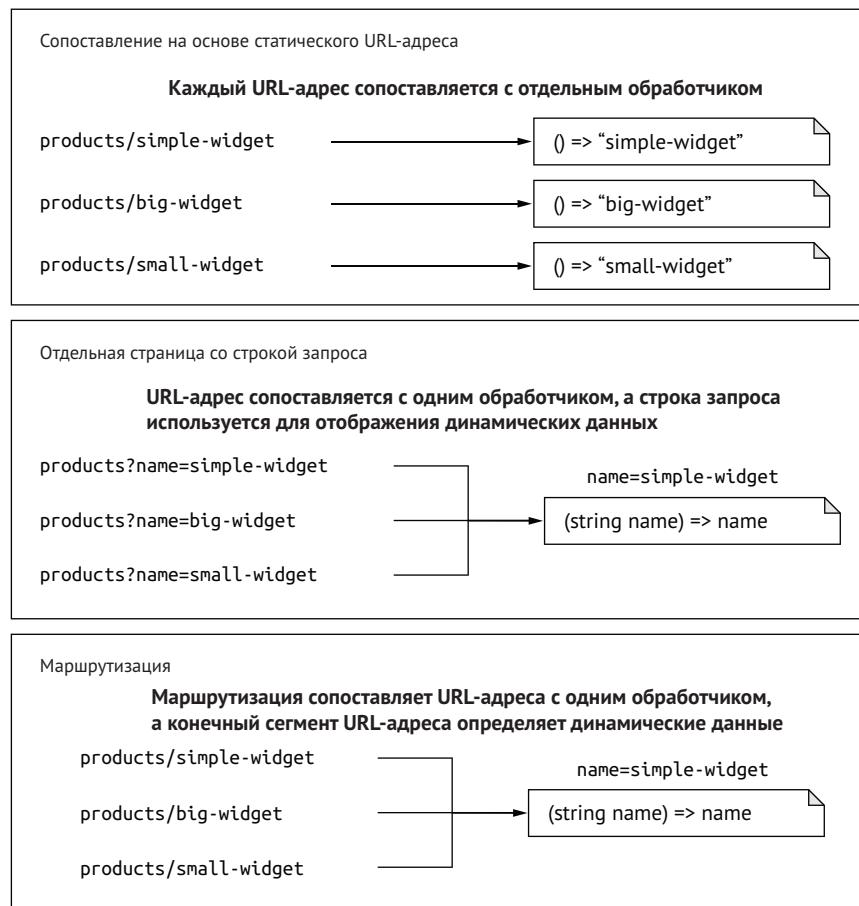


Рис. 6.2 Если вы используете статическое сопоставление на основе URL-адресов, вам понадобится отдельный обработчик для каждого продукта в вашем ассортименте. Со строкой запроса можно использовать один обработчик, а строка запроса содержит данные. При маршрутизации нескольких URL-адресов сопоставляются с одним обработчиком, а динамический параметр фиксирует разницу в URL-адресах

С помощью маршрутизации можно легко изменить предоставленные URL-адреса, не меняя имен файлов или местоположений. Вы также можете использовать маршрутизацию для создания более удобных URL-адресов для пользователей, что может улучшить обнаружение и «возможность взлома». Все следующие маршруты могут указывать на один и тот же обработчик:

- /rates/view/1;
- /rates/view/USD;
- /rates/current-exchange-rate/USD;
- /current-exchange-rate-for-USD.

Такой уровень настройки требуется нечасто, но очень полезно иметь возможность настраивать URL-адреса, когда вам это нужно.

В следующем разделе мы рассмотрим, как на практике работает маршрутизация в ASP.NET Core.

6.2 Маршрутизация конечных точек в ASP.NET Core

В этом разделе описано, как работает маршрутизация конечных точек в ASP.NET Core, в особенности что касается минимальных API и конвейера промежуточного ПО. В главе 14 вы узнаете, как маршрутизация используется в Razor Pages и фреймворке ASP.NET Core MVC.

Маршрутизация была частью ASP.NET Core с момента ее создания, но претерпела некоторые крупные изменения. В ASP.NET Core 2.0 и 2.1 маршрутизация была ограничена Razor Pages и фреймворком ASP.NET Core MVC. В конвейере промежуточного ПО не было выделенного компонента маршрутизации; маршрутизация осуществлялась только в Razor Pages или компонентах MVC.

К сожалению, ограничение маршрутизации инфраструктурой MVC и Razor Pages сделало некоторые вещи немного запутанными. Некоторые сквозные задачи, такие как авторизация, были ограничены инфраструктурой MVC, и их было трудно использовать из другого промежуточного ПО в приложении. Это ограничение привело к неизбежному дублированию, что было неидеально.

В ASP.NET Core 3.0 представлена новая система маршрутизации: *маршрутизация конечных точек*. Маршрутизация конечных точек делает систему маршрутизации более фундаментальной функцией ASP.NET Core и больше не привязывает ее к инфраструктуре MVC. Теперь Razor Pages, MVC и другое промежуточное ПО могут использовать одну и ту же систему маршрутизации. .NET 7 продолжает использовать ту же систему маршрутизации конечных точек, которая является неотъемлемой частью функциональности минимальных API, представленной в .NET 6.

Маршрутизация конечных точек имеет фундаментальное значение для всех приложений ASP.NET Core, кроме самых простых. Она реализуется с помощью двух компонентов, которые вы уже видели:

- `EndpointRoutingMiddleware` – этот компонент выбирает, какие зарегистрированные конечные точки будут выполняться для данного запроса во время выполнения. Чтобы было легче различать эти два типа, на протяжении всей книги я буду называть его `RoutingMiddleware`;
- `EndpointMiddleware` – этот компонент обычно размещается в конце конвейера промежуточного ПО. Он *вызывает* конечную точку, выбранную `RoutingMiddleware` во время выполнения.

Конечные точки регистрируются в приложении путем вызова функции `Map*` экземпляра `IEndpointRouteBuilder`. В приложениях .NET 7 данный экземпляр обычно является экземпляром `WebApplication`, но это не обязательно, как вы увидите в главе 30.

ОПРЕДЕЛЕНИЕ Конечная точка в ASP.NET Core – это обработчик, возвращающий ответ. Каждая конечная точка связана с шаблоном URL-адреса. В зависимости от типа приложения, которое вы создаете

те, обработчики минимальных API, обработчики Razor Pages или методы действий контроллера MVC обычно составляют основную часть конечных точек в приложении. Например, вы также можете использовать простое промежуточное ПО в качестве конечной точки или конечную точку проверки работоспособности.

`WebApplication` реализует интерфейс `IEndpointRouteBuilder`, поэтому можно регистрировать конечные точки непосредственно в нем. В листинге 6.1 показано, как зарегистрировать несколько конечных точек:

- обработчик минимальных API, использующий метод `MapGet()`, как было показано в предыдущих главах;
- конечная точка проверки работоспособности с использованием метода `MapHealthChecks()`. Подробнее о проверках работоспособности можно прочитать на странице <http://mng.bz/N2YD>;
- все конечные точки Razor Pages в приложении используют метод `MapRazorPages()`. Подробнее о маршрутизации с помощью Razor Pages вы узнаете в главе 14.

Листинг 6.1. Регистрация нескольких конечных точек с помощью `WebApplication`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddHealthChecks();
builder.Services.AddRazorPages();
```

Добавляет сервисы, необходимые для компонента проверки работоспособности и Razor Pages

```
WebApplication app = builder.Build();

app.MapGet("/test", () => "Hello world!"); ←
app.MapHealthChecks("/healthz"); ←
app.MapRazorPages();
```

Регистрирует конечную точку минимального API, которая возвращает «Hello World!» на маршруте /test

Регистрирует конечную точку проверки работоспособности на маршруте /healthz

```
app.Run();
```

Регистрирует все страницы Razor Pages в приложении в качестве конечных точек

Каждая конечная точка связана с *шаблоном маршрута*, который определяет, с какими URL-адресами конечная точка должна совпасть. В предыдущем листинге видно два шаблона маршрута, `"/healthz"` и `"/test"`.

ОПРЕДЕЛЕНИЕ *Шаблон маршрута* – это шаблон URL-адреса, который используется для сопоставления с URL-адресами запроса. Это строки с фиксированными значениями, как, например, `"/test"` из предыдущего листинга. Они также могут содержать заполнители для переменных, как вы увидите в разделе 6.3.

`WebApplication` сохраняет зарегистрированные маршруты и конечные точки в словаре, который используется совместно `RoutingMiddleware` и `EndpointMiddleware`.

ПРИМЕЧАНИЕ По умолчанию WebApplication автоматически добавляет RoutingMiddleware в начало, а EndpointMiddleware в конец конвейера промежуточного ПО, однако можно переопределить местоположение в конвейере, вызвав методы UseRouting() или UseEndpoints(). См. раздел 4.2.3 для получения более подробной информации об автоматически добавляемых компонентах промежуточного ПО.

Во время выполнения RoutingMiddleware сравнивает входящий запрос с маршрутами, зарегистрированными в словаре. Если RoutingMiddleware находит подходящую конечную точку, то отмечает, какая конечная точка была выбрана, и прикрепляет ее к объекту запроса HttpContext, после чего вызывает следующий компонент промежуточного ПО в конвейере. Когда запрос достигает EndpointMiddleware, компонент промежуточного ПО проверяет, какая конечная точка была выбрана, и выполняет ее (и любые связанные фильтры конечных точек), как показано на рис. 6.3.

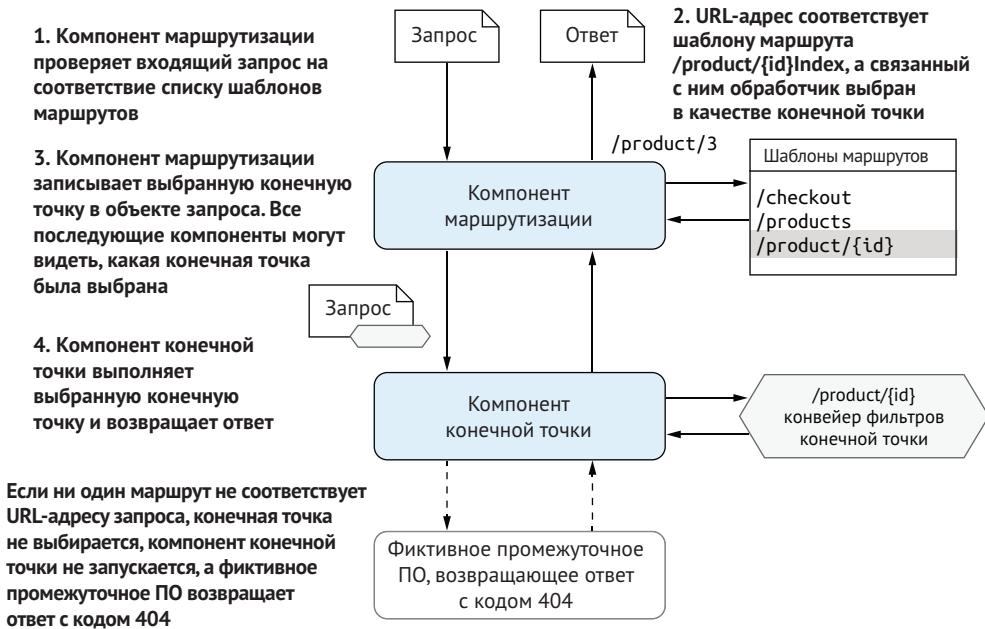


Рис. 6.3 Маршрутизация конечных точек использует двухэтапный процесс. RoutingMiddleware выбирает конечную точку для выполнения, а EndpointMiddleware выполняет ее. Если URL-адрес запроса не соответствует шаблону маршрута, компонент конечной точки не генерирует ответ

Если URL-адрес запроса *не* соответствует шаблону маршрута, RoutingMiddleware не выберет конечную точку, но запрос все равно продолжит идти по конвейеру промежуточного ПО. Поскольку конечная точка не выбрана, EndpointMiddleware молча игнорирует запрос и передает его следующему компоненту в конвейере. EndpointMiddleware – обычно последний компонент в конвейере, поэтому «следующий» компонент, как правило, является фиктивным и всегда возвращает ответ 404 Not Found, как вы уже видели в главе 4.

СОВЕТ Если URL-адрес запроса не соответствует шаблону маршрута, конечная точка не выбирается и не выполняется. Весь конвейер промежуточного ПО по-прежнему выполняется, но обычно возвращается ответ 404, когда запрос достигает фиктивного компонента 404.

Преимущество наличия двух отдельных компонентов промежуточного ПО для обработки этого процессса на первый взгляд может быть неочевидным. Рисунок 6.3 намекает на главное преимущество: все компоненты, размещенные после `RoutingMiddleware`, могут видеть, какая конечная точка будет выполнена, прежде чем это случится.

ПРИМЕЧАНИЕ Только компонент промежуточного ПО, размещенный после `RoutingMiddleware`, может определить, какая конечная точка будет выполняться.

На рис. 6.4 показан более реалистичный конвейер промежуточного ПО, в котором компонент размещается как *перед* `RoutingMiddleware`, так и *между* `RoutingMiddleware` и `EndpointMiddleware`.

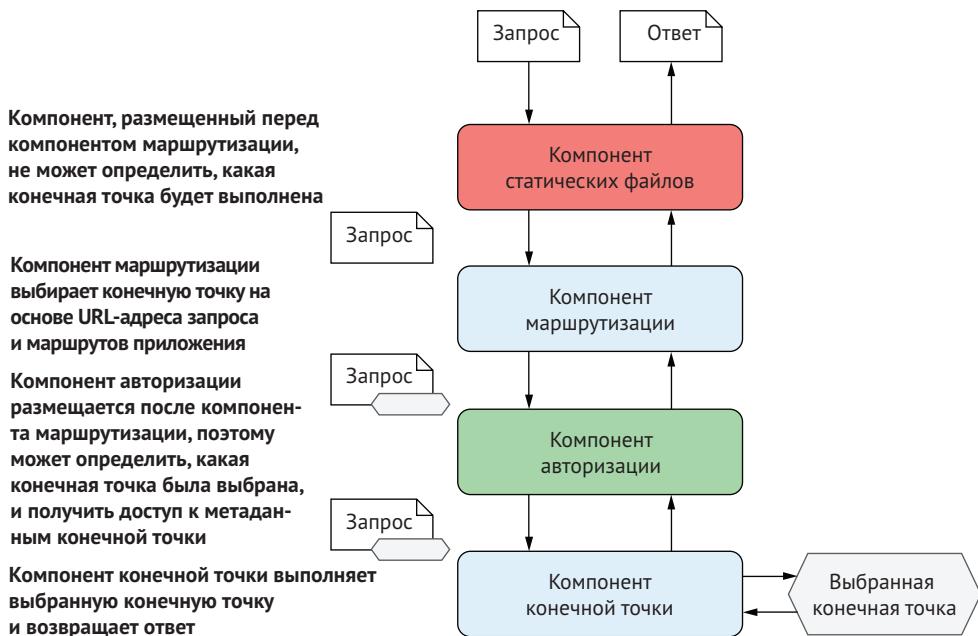


Рис. 6.4 Компонент, размещенный перед компонентом маршрутизации, не может определить, какая конечная точка будет выбрана. Компоненту, размещенному между компонентом маршрутизации и компонентом конечной точки, известна выбранная конечная точка

`StaticFileMiddleware` на рис. 6.4 помещен перед `RoutingMiddleware`, поэтому он выполняется до того, как будет выбрана конечная точка. И наоборот, `AuthorizationMiddleware` размещен *после* `RoutingMiddleware`, когда известно, какая конечная точка минимального API в конечном итоге будет выполнена. Кроме того, он может получить доступ к опре-

деленным метаданным о конечной точке, например ее имени, и необходимые полномочия для доступа к ней.

СОВЕТ `AuthorizationMiddleware` нужно знать, какая конечная точка будет выполняться, поэтому в конвейере его необходимо разместить *после* `RoutingMiddleware` и *перед* `EndpointMiddleware`. Более подробно авторизация обсуждается в главе 24.

При разработке приложения важно помнить о различных ролях двух типов компонента маршрутизации. Если у вас есть компонент промежуточного ПО, который должен знать, какую конечную точку (если она есть) будет выполнять данный запрос, то необходимо убедиться, что он будет помещен после `RoutingMiddleware` и перед `EndpointMiddleware`.

СОВЕТ Если вы хотите разместить компонент перед `RoutingMiddleware`, например `StaticFileMiddleware` на рис. 6.4, необходимо переопределить автоматическое промежуточное ПО, добавленное `WebApplication`, вызвав метод `UseRouting()` в соответствующем месте конвейера промежуточного ПО. Пример см. в листинге 4.3 в главе 4.

Я рассказал, как взаимодействуют `RoutingMiddleware` и `EndpointMiddleware`, обеспечивая возможности маршрутизации в ASP.NET Core, но до сих пор мы рассматривали только простые шаблоны маршрутов. В следующем разделе мы разберем некоторые из многих функций, доступных в шаблонах маршрутов.

6.3 Изучение синтаксиса шаблона маршрута

До сих пор в этой книге мы рассматривали простые шаблоны маршрутов, состоящие из фиксированных значений, например `/person` и `/test`, а также использующие базовый параметр маршрута, такой как `/fruit/{id}`.

В этом разделе мы рассмотрим полный спектр функций, доступных в шаблонах маршрутов, таких как значения по умолчанию, необязательные сегменты и ограничения.

6.3.1 Работа с параметрами и литеральными сегментами

Шаблоны маршрутов обладают богатым и гибким синтаксисом. Однако на рис. 6.5 показан простой пример, похожий на те, которые вы уже видели.

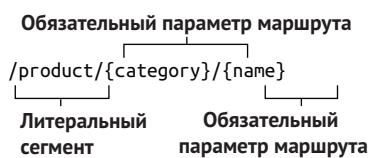


Рис. 6.5 Простой шаблон маршрута с литеральным сегментом и двумя обязательными параметрами маршрута

Компонент маршрутизации анализирует шаблон маршрута, разбивая его на сегменты. Обычно сегмент отделяется символом `/`, но это может быть любой допустимый символ.

ОПРЕДЕЛЕНИЕ Сегменты, в которых используется символ, отличный от /, называются *сложными сегментами*. Обычно я рекомендую избегать их и придерживаться использования символа / в качестве разделителя. У сложных сегментов есть некоторые особенности, которые затрудняют их использование, поэтому обязательно ознакомьтесь с документацией на странице <http://mng.bz/D4RE>, прежде чем использовать их.

Каждый сегмент – это либо:

- *литеральное значение*, как, например, product на рис. 6.5;
- *параметр маршрута*, например {category} и {name} на рис. 6.5.

URL-адрес запроса должен точно соответствовать литературным значениям (без учета регистра). Если вам нужно точно сопоставить определенный URL-адрес, можно использовать шаблон, состоящий только из литералов.

ПОДСКАЗКА Литеральные сегменты в ASP.NET Core не чувствительны к регистру.

Представьте, что в вашем приложении есть минимальный API, определенный с помощью

```
app.MapGet("/About/Contact", () => {/* */})
```

Этот шаблон маршрута "/About/Contact" состоит только из литературных значений, поэтому он соответствует только точному URL-адресу (без учета регистра). Ни один из следующих URL-адресов не соответствует этому шаблону маршрута:

- /about;
- /about-us/contact;
- /about/contact/email;
- /about/contact-us.

Параметры маршрута – это разделы URL-адреса, которые могут различаться, но по-прежнему будут соответствовать шаблону. Они определяются путем присвоения им имени и помещения их в фигурные скобки, например {category} или {name}. При таком использовании параметры обязательны, поэтому в URL-адресе запроса должен быть сегмент, которому они соответствуют, но значение может быть разным.

Возможность использовать параметры маршрута дает большую гибкость. Простой шаблон маршрута "{category}/{name}" можно использовать для сопоставления всех URL-адресов страниц продукта в приложении онлайн-торговли:

- /bag/rucksack-a, где category=bags, а name=rucksack-a;
- /shoes/black-size9, где category=shoes, а name=black-size9.

Но обратите внимание, что этот шаблон *не* будет соответствовать следующим URL-адресам:

- /socks/ – параметр name не указан;
- /trousers/mens/formal – дополнительный сегмент URL-адреса, formal, отсутствует в шаблоне маршрута.

Когда шаблон маршрута определяет параметр маршрута и маршрут соответствует URL-адресу, значение, связанное с параметром, фиксируется и сохраняется в словаре значений, связанных с запросом. Эти значения маршрута обычно определяют другое поведение в конечной точке, и их можно внедрить в обработчики (как было вкратце показано в главе 5) в процессе под названием *привязка модели*.

ОПРЕДЕЛЕНИЕ Значения маршрута – это значения, извлеченные из URL-адреса на основе заданного шаблона маршрута. Каждый параметр маршрута в шаблоне будет иметь связанное значение маршрута, и они хранятся в словаре в виде пары строк. Их можно использовать во время привязки модели, как вы увидите в главе 7.

Литеральные сегменты и параметры маршрута – это два краеугольных камня шаблонов маршрутов ASP.NET Core. Используя эти две концепции, можно создавать любые URL-адреса для своего приложения. В оставшейся части раздела 6.3 мы рассмотрим дополнительные функции, позволяющие использовать необязательные сегменты URL-адресов, предоставлять значения по умолчанию, если сегмент не указан, и накладывать дополнительные ограничения на значения, являющиеся допустимыми для данного параметра маршрута.

6.3.2 Использование необязательных значений и значений по умолчанию

В предыдущем разделе вы видели простой шаблон маршрута с литературным сегментом и двумя обязательными параметрами маршрутизации. На рис. 6.6 показан более сложный маршрут, использующий ряд дополнительных функций.

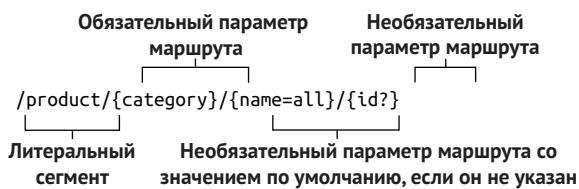


Рис. 6.6 Более сложный шаблон маршрута с литературными сегментами, именованными параметрами маршрута, необязательными параметрами и значениями по умолчанию

Литеральный сегмент `product` и обязательный параметр `{category}` такие же, как на рис. 6.5. Параметр `{name}` выглядит *похожим*, но для него задано значение по умолчанию с помощью `=all`. Если URL-адрес не содержит сегмента, соответствующего параметру `{name}`, маршрутизатор будет использовать вместо него значение `all`.

Последний сегмент рис. 6.6, `{id?}`, определяет необязательный параметр маршрута `id`. Этот сегмент URL-адреса является необязательным: если он присутствует, маршрутизатор захватит значение параметра `{id}`; если его там нет, то он не будет создавать значение маршрута для `id`. В своих шаблонах можно указать любое количество параметров маршрута, и эти значения будут доступны вам, когда дело дойдет до привязки модели. Сложный шаблон маршрута на рис. 6.6 позволяет сопоставить большее количество URL-адресов, сделав параметры `{name}` и `{id}` необязательными и предоставив значение по умолчанию для `{name}`.

В табл. 6.1 показаны некоторые возможные URL-адреса, которым будет соответствовать этот шаблон, и соответствующие значения маршрута, которые задал бы маршрутизатор.

Обратите внимание, что нельзя указать значение для необязательного параметра `{id}`, не указав также параметры `{category}` и `{name}`. Можно поместить необязательный параметр (не имеющий значения по умолчанию) *только в конец* шаблона маршрута.

Использование значений по умолчанию позволяет иметь несколько способов вызова одного и того же URL-адреса, что может быть полезно в некоторых случаях. Учитывая шаблон маршрута, показанный на рис. 6.6, следующие два URL-адреса эквивалентны:

- `/product/shoes;`
- `/product/shoes/all.`

Оба этих адреса будут выполнять один и тот же обработчик конечной точки с одинаковыми значениями маршрута `category=shoes` и `name=all`. Использование значений по умолчанию позволяет применять более короткие и запоминающиеся URL-адреса в приложении для распространенных вариантов, но при этом иметь гибкость, позволяющую сопоставить множество маршрутов в одном шаблоне.

Таблица 6.1 URL-адреса, которые будут соответствовать шаблону на рис. 6.6, и их соответствующие значения маршрута

URL-адрес	Значения маршрута
<code>/product/shoes/formal/3</code>	<code>category=shoes, name=formal, id=3</code>
<code>/product/shoes/formal</code>	<code>category=shoes, name=formal</code>
<code>/product/shoes</code>	<code>category=shoes, name=all</code>
<code>/product/bags/satchels</code>	<code>category=bags, name=satchels</code>
<code>/product/phones</code>	<code>category=phones, name=all</code>
<code>/product/computers/laptops/ABC-123</code>	<code>category=computers, name=laptops, id=ABC-123</code>

6.3.3 Добавление дополнительных ограничений к параметрам маршрута

Определив, является ли параметр маршрута обязательным или необязательным и имеет ли он значение по умолчанию, можно сопоставить широкий диапазон URL-адресов с довольно кратким синтаксисом шаблона. К сожалению, в некоторых случаях такой подход оказывается слишком широким. Маршрутизация сопоставляет сегменты URL-адреса только с параметрами маршрута; она ничего не знает о данных, которые, как вы ожидаете, будут содержать эти параметры. Если рассматривать шаблон, аналогичный тому, что изображен на рис. 6.6, "`{category}/{name=all}/{id??"}`", то следующие URL-адреса будут соответствовать:

- `/shoes/sneakers/test;`
- `/shoes/sneakers/123;`
- `/Account/ChangePassword;`
- `/ShoppingCart/Checkout/Start;`
- `/1/2/3.`

Все эти URL-адреса являются вполне допустимыми с учетом синтаксиса шаблона, но некоторые из них могут вызвать проблемы у приложения. Они состоят из двух или трех сегментов, поэтому маршрутизатор с радостью назначает значения маршрута и сопоставляет шаблон, когда вы, возможно, этого и не хотите! Это присвоенные значения маршрута:

- `/shoes/sneakers/test` содержит значения маршрута `Category=shoes`, `name=sneakers` и `id=test`;
- `/shoes/sneakers/123` содержит значения маршрута `Category=shoes`, `name=sneakers` и `id=123`;
- `/Account/ChangePassword` имеет значения маршрута `Category=Account` и `name=ChangePassword`;
- `/Cart/Checkout/Start` имеет значения маршрута `Category=Cart`, `name=Checkout` и `id=Start`;
- `/1/2/3` имеет значения маршрута `Category=1`, `name=2` и `id=3`.

Обычно маршрутизатор передает значения маршрутов обработчикам через привязку модели, которую вы кратко видели в главе 5 (и которая подробно обсуждается в главе 7). Конечная точка минимального API, определенная как

```
app.MapGet("/fruit/{id}", (int id) => "Hello world!");
```

получит аргумент `id` из значения `id` маршрута. Если параметру маршрута `id` в конечном итоге будет присвоено *не целочисленное* значение из URL-адреса, вы получите исключение, когда оно будет привязано к целочисленному параметру `id`.

Чтобы избежать этой проблемы, можно добавить к шаблону маршрута дополнительные *ограничения*, которые должны быть выполнены, чтобы URL-адрес считался совпадающим. Ограничения можно определить в шаблоне маршрута для данного параметра маршрута с помощью двоеточия. Например, `{id: int}` добавит ограничение `IntRouteConstraint` к параметру `id`. Чтобы данный URL-адрес считался совпадающим, значение, присвоенное значению маршрута `id`, должно быть преобразовано в целое число.

Можно применить большое количество ограничений маршрута к шаблонам маршрутов, чтобы обеспечить преобразование значений маршрута в соответствующие типы. Также можно использовать более сложные ограничения, например выполнить проверку на предмет того, что целочисленное значение имеет определенное минимальное значение или строковое значение имеет максимальную длину. В табл. 6.2 описан ряд доступных ограничений, а более полный список можно найти в документации Microsoft на странице <http://mng.bz/BmRJ>.

ПОДСКАЗКА Как видно из таблицы, также можно комбинировать несколько ограничений, разделяя их двоеточиями.

Использование ограничений позволяет сузить число URL-адресов, которым будет соответствовать данный шаблон маршрута. Когда компонент маршрутизации сопоставляет URL-адрес с шаблоном маршрута, он опрашивает ограничения, чтобы убедиться, что все они действительны. Если это не так, то шаблон маршрута не считается совпадающим, и конечная точка не будет выполнена.

Таблица 6.2 Возможные ограничения и их описание

Ограничение	Пример	Описание	Примеры совпадений
int	{qty:int}	Соответствует любому целому числу	123, -123, 0
Guid	{id:guid}	Соответствует любому глобальному уникальному идентификатору	d071b70c-a812-4b54-87d2-7769528e2814
decimal	{cost:decimal}	Соответствует любому значению decimal	29.99, 52, -1.01
min(value)	{age:min(18)}	Соответствует целочисленным значениям от 18 и больше	18, 20
length(value)	{name:length(6)}	Соответствует строковым значениям длиной 6 символов	Andrew,123456
optional int	{qty:int?}	Отсутствует либо соответствует любому целому числу	123, -123, 0, null
optional int max(value)	{qty:int:max(10)?}	Отсутствует либо соответствует любому целому числу 10 или меньше	3, -123, 0, null

ВНИМАНИЕ! Не используйте ограничения маршрутов для проверки общих вводимых данных, например для проверки правильности адреса электронной почты. Это приведет к появлению сообщения об ошибке «Страница не найдена», которое будет сбивать с толку пользователя. Вы также должны знать, что все эти встроенные ограничения предполагают инвариантные языковые и региональные параметры, что может оказаться проблематичным, если ваше приложение использует URL-адреса, локализованные для других языков.

Ограничения лучше всего использовать экономно, но они могут быть полезны, если у вас есть строгие требования к URL-адресам, используемым приложением, поскольку они могут позволить обойти некоторые сложные комбинации. Вы даже можете создавать собственные ограничения, как описано в документации: <http://mng.bz/d14Q>.

Ограничения и порядок в маршрутизации на основе атрибутов

Если у вас есть хорошо разработанный набор URL-адресов для вашего приложения, то вы, вероятно, обнаружите, что на самом деле вам не нужно использовать ограничения маршрутов. Эти ограничения действительно полезны, когда у вас есть «перекрывающиеся» шаблоны маршрутов.

Предположим, что у вас есть конечная точка с шаблоном маршрута "{number}/{name}" и еще одна – с шаблоном "{product}/{id}". Какой шаблон выбирается при поступлении запроса с URL-адресом /shoes/123?

Они оба совпадают, поэтому компонент маршрутизации начинает паниковать и выбрасывает исключение. Неидеальный вариант.

Это можно исправить, используя ограничения. Например, если обновить первый шаблон, чтобы он выглядел так: "{number: int}/{name}", то целочисленное ограничение будет означать, что URL-адрес больше не соответствует, и компонент маршрутизации может сделать правильный выбор. Однако

обратите внимание, что URL-адрес `/123/shoes` по-прежнему соответствует обоим шаблонам маршрута, поэтому опасность еще не миновала.

Как правило, следует избегать дублирования подобных шаблонов маршрутов, поскольку они часто сбивают с толку и вызывают больше проблем. Если ваши шаблоны маршрутов четко определены, так что каждый URL-адрес сопоставляется только с одним шаблоном, маршрутизация будет работать без каких-либо трудностей. Придерживаться встроенных соглашений, насколько это возможно, – лучший способ, чтобы все шло гладко!

На этом наше изучение шаблонов маршрутов подошло к концу, но, прежде чем мы продолжим, есть еще один тип параметров, о котором следует подумать: универсальный параметр.

6.3.4 Сопоставление произвольных URL-адресов с помощью универсального параметра

Вы уже видели, как шаблоны маршрутов берут сегменты URL-адреса и пытаются сопоставлять их с параметрами или литеральными строками. Эти сегменты обычно разделяются символом косой черты `/`, поэтому сами параметры маршрута не содержат такого символа. Что делать, если вам нужно, чтобы они содержали косую черту, или вы не знаете, сколько сегментов у вас будет?

Представьте, что вы создаете приложение – конвертер валют, которое показывает обменный курс одной валюты к одной или нескольким валютам. Вам сказали, что URL-адреса этой страницы должны содержать все валюты в виде отдельных сегментов. Вот несколько примеров:

- `/USD/convert/GBP` – показывать доллары США с курсом обмена на фунты стерлингов;
- `/USD/convert/GBP/EUR` – отображать доллары США с курсами обмена на фунты стерлингов и евро;
- `/USD/convert/GBP/EUR/CAD` – показывать доллары США с курсами обмена на фунты стерлингов, евро и канадские доллары.

Если вы хотите поддерживать отображение *любого* количества валют, как это делают приведенные выше URL-адреса, вам нужен способ перехватывать *все*, что идет после сегмента `convert`. Этого можно было бы добиться с помощью универсального параметра в шаблоне маршрута, как показано на рис. 6.7.



Рис. 6.7 Можно использовать универсальные параметры для сопоставления оставшейся части URL-адреса. Эти параметры могут включать в себя символ `«/»` или могут быть пустой строкой

Универсальные параметры можно объявить с помощью одной или двух звездочек внутри определения параметра, например `{*others}` или `**others`. Они будут соответствовать оставшейся несовпадающей части URL-адреса, включая любые косые черты и другие символы, не входящие в состав более ранних параметров. Они также могут соот-

ветствовать пустой строке. Для URL-адреса /USD/convert/GBP/EUR значение `others` будет представлять собой одну строку "GBP/EUR".

СОВЕТ Универсальные параметры – жадные. Это означает, что они захватывают всю несовпадающую часть URL-адреса. По возможности, чтобы избежать путаницы, избегайте определения шаблонов маршрутов с такого рода параметрами, которые перекрывают другие шаблоны маршрутов.

Варианты с одной и двумя звездочками ведут себя идентично при маршрутизации входящего запроса в конечную точку. Разница появляется только тогда, когда вы *генерируете* URL-адреса (эту тему мы рассмотрим в следующем разделе): URL-адрес с одной звездочкой кодирует косую черту, а версия с двумя звездочками – нет. Как правило, вам потребуется поведение версии с двумя звездочками.

СОВЕТ Примеры и сравнение вариантов с одной и двумя звездочками см. в документации на странице <http://mng.bz/rWyX>.

Вы правильно поняли последний абзац: сопоставление URL-адресов с конечными точками – это лишь половина обязанностей системы маршрутизации в ASP.NET Core. Он также используется для создания URL-адресов, чтобы можно было с легкостью ссылаться на свои конечные точки из других частей приложения.

6.4 Генерация URL-адресов из параметров маршрута

В этом разделе мы рассмотрим вторую половину обязанностей маршрутизации – генерацию URL-адресов. Вы узнаете, как сгенерировать URL-адреса в виде строки, которую можно использовать в своем коде, и как автоматически отправлять URL-адреса перенаправления в качестве ответа от ваших конечных точек.

Одним из побочных продуктов использования инфраструктуры маршрутизации в ASP.NET Core является тот факт, что ваши URL-адреса могут быть изменчивы. Вы можете изменить шаблоны маршрутов в приложении по своему усмотрению – например, переименовав /cart в /basket – и не получите никаких ошибок компиляции.

Конечно, конечные точки не изолированы; вам неизбежно захочется включить ссылку из одной конечной точки в другую. Попытка управлять этими ссылками в приложении вручную приведет к душевной боли, неработающим ссылкам и ошибкам 404. Если бы ваши URL-адреса были жестко закодированы, то вам нужно было бы непременно выполнять поиск и замену при каждом переименовании!

К счастью, можно использовать инфраструктуру маршрутизации для динамической генерации соответствующих URL-адресов во время выполнения. Это освободит вас от подобного бремени. По сути, это почти полная противоположность процессу сопоставления URL-адреса со страницей Razor Pages, как показано на рис. 6.8. В случае маршрутизации компонент маршрутизации принимает URL-адрес,

сопоставляет его с шаблоном маршрута и разбивает его на значения. В случае генерации URL-адреса генератор принимает значения маршрута и объединяет их с шаблоном маршрута для создания адреса.

Для создания URL-адресов для минимальных API можно использовать класс `LinkGenerator`. Его можно применять в любой части приложения, равно как и в промежуточном ПО и любых других сервисах. У класса `LinkGenerator` имеются различные методы для создания URL-адресов, например `GetPathByPage` и `GetPathByAction`, которые используются специально для маршрутизации в Razor Pages и действиях MVC, поэтому мы рассмотрим их в главе 14. Нас интересуют методы, связанные с именованными маршрутами.

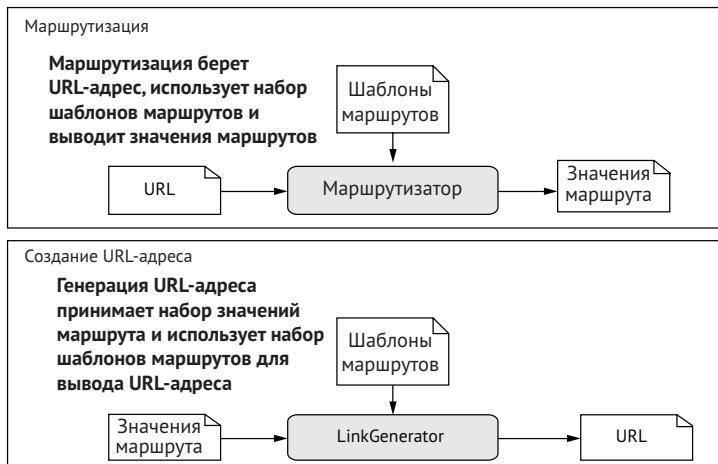


Рис. 6.8 Сравнение маршрутизации и генерации URL-адреса. Маршрутизация принимает URL-адрес и генерирует значения маршрута, а генерация URL-адресов использует значения маршрута для создания адреса

6.4.1 Генерация URL-адресов для конечной точки минимального API с помощью класса `LinkGenerator`

Вам нужно будет создавать URL-адреса в разных местах своего приложения, и одно из наиболее распространенных мест – это конечные точки минимальных API. В следующем листинге показано, как создать ссылку на одну конечную точку из другой, аннотируя целевую конечную точку именем и используя класс `LinkGenerator`.

Листинг 6.2. Генерация URL-адреса и именованной конечной точки

```

Конечная точка выводит имя, полученное
в шаблоне маршрута.

app.MapGet("/product/{name}", (string name) => $"The product is {name}") <|
    .WithName("product"); <| Дает конечной точке имя, добавляя
                                к ней метаданные.

app.MapGet("/links", (LinkGenerator links) => <| Ссылается на класс
                                LinkGenerator в обработ-
                                чике конечной точки.

```

```

{
    string link = links.GetPathByName("product",
        new { name = "big-widget"});
    return $"View the product at {link}"; <-- 
});      Возвращает значение «View the product
          at /product/big-widget».

```

**Создает ссылку, ис-
пользуя имя маршрута
«product», и предос-
тавляет значение для па-
раметра маршрута.**

Метод `WithName()` добавляет метаданные к вашим конечным точкам, чтобы на них могли ссылаться другие части вашего приложения. В данном случае мы добавляем имя конечной точке, чтобы иметь возможность обратиться к ней позже. Подробнее о метаданных рассказываеться в главе 11.

ПРИМЕЧАНИЕ Имена конечных точек чувствительны к регистру (в отличие от самих шаблонов маршрутов) и должны быть глобально уникальными. Повторяющиеся имена вызывают исключения во время выполнения.

`LinkGenerator` – это служба, доступная в любом месте ASP.NET Core. Вы можете получить к ней доступ из своих конечных точек, включив ее в качестве параметра в обработчик.

ПРИМЕЧАНИЕ Вы можете ссылаться на `LinkGenerator` в своем обработчике, поскольку он автоматически регистрируется в контейнере внедрения зависимостей. Вы узнаете о внедрении зависимостей в главах 8 и 9.

Метод `GetPathByName()` принимает имя маршрута и, при необходимости, данные маршрута. Данные маршрута упаковываются в виде пар «ключ–значение» в один анонимный объект C#. Если нужно передать несколько значений маршрута, можно добавить дополнительные свойства к анонимному объекту. Затем вспомогательная функция сгенерирует путь на основе шаблона маршрута указанной конечной точки.

В листинге 6.2 показано, как сгенерировать путь. Но вы также можете создать полный URL-адрес, используя метод `GetUriByName()` и указав значения для хоста и схемы, как в этом примере:

```
links.GetUriByName("product", new { Name = "super-fancy-widget" },
    "https", new HostString("localhost"));
```

Кроме того, некоторые методы, доступные в `LinkGenerator`, принимают `HttpContext`. Эти методы зачастую проще использовать в обработчике конечной точки, поскольку они извлекают внешние значения, например схему и имя хоста, из входящего запроса и повторно используют их для генерации URL-адресов.

ВНИМАНИЕ! Будьте осторожны при использовании метода `GetUriByName`. В вашем приложении могут вскрыться уязвимости, если вы используете непроверенные значения хоста. Дополнительную информацию о фильтрации хостов и о том, почему это важно, можно найти в этом посте: <http://mng.bz/V1d5>.

В листинге 6.2, помимо имени маршрута, я передал в `GetPathByName` анонимный объект:

```
string link = links.GetPathByName("product", new { name = "big-widget"});
```

Этот объект предоставляет дополнительные значения маршрута при создании URL-адреса, в данном случае присваивая параметру `name` значение `"big-widget"`.

Если выбранный маршрут явно включает определенное значение маршрута в свое определение, например как в шаблоне маршрута `"/product/{name}"`, значение маршрута будет использоваться в пути URL-адреса, в результате чего получится `/product/big-widget`. Если маршрут не содержит значение маршрута явно, как в шаблоне `"/product"`, значение маршрута добавляется к строке запроса в качестве дополнительных данных, как в `/product?name=big-widget`.

6.4.2 Генерация URL-адресов с помощью `IResults`

Создание URL-адресов, ссылающихся на другие конечные точки, часто встречается, например, при создании REST API. Но не всегда нужно отображать URL-адреса. Иногда вам нужно автоматически перенаправить пользователя на URL-адрес. В этой ситуации можно использовать `Results.RedirectToRoute()` для обработки генерации URL-адреса.

ПРИМЕЧАНИЕ Перенаправления чаще встречаются в приложениях с отрисовкой на стороне сервера, например Razor Pages, но они прекрасно подходят и для API-приложений.

В листинге 6.3 показано, как вернуть ответ из конечной точки, который автоматически перенаправляет пользователя в конечную точку с другим именем. Метод `RedirectToRoute()` принимает имя конечной точки и все необходимые параметры маршрута и генерирует URL-адрес аналогично `LinkGenerator`. Фреймворк автоматически отправляет сгенерированный URL-адрес в качестве ответа, поэтому вы никогда не увидите URL-адрес в своем коде. Затем браузер пользователя считывает URL-адрес из ответа и автоматически перенаправляет его на новую страницу.

Листинг 6.3. Генерация перенаправления на URL-адрес с помощью `Results.RedirectToRoute()`

```
app.MapGet("/test", () => "Hello world!")  
    .WithName("hello");
```

← Аннотирует маршрут именем «hello»


```
app.MapGet("/redirect-me",  
    () => Results.RedirectToRoute("hello"))
```

← Генерирует ответ, который отправляет перенаправление в конечную точку «hello».

По умолчанию метод `RedirectToRoute()` генерирует ответ 302 Found и включает сгенерированный URL-адрес в заголовок ответа `Location`. Вы можете управлять используемым кодом состояния, присвоив дополнительным параметрам `preserveMethod` и `permanent` следующие значения:

- permanent=false, preserveMethod=false – 302 Found;
- permanent=true, preserveMethod=false – 301 Moved Permanently;
- permanent=false, preserveMethod=true – 307 Temporary Redirect;
- permanent=true, preserveMethod=true – 308 Permanent Redirect.

ПРИМЕЧАНИЕ Каждый из кодов состояния перенаправления имеет немного разное семантическое значение, хотя на практике многие сайты просто используют 302. Будьте осторожны с кодами состояния постоянного перемещения; они заставят браузеры никогда не вызывать исходный URL-адрес, всегда отдавая предпочтение месту перенаправления. Неплохое объяснение этих кодов (и полезного кода состояния 303 See Other) можно найти в документации Mozilla на странице <http://mng.bz/x4GB>.

Помимо перенаправления на определенную конечную точку, можно выполнить перенаправление на произвольный URL-адрес, используя метод `Results.Redirect()`. Этот метод работает так же, как `RedirectToRoute()`, но вместо имени маршрута принимает URL-адрес и может быть полезен для перенаправления на внешние URL-адреса.

Независимо от того, генерируете ли вы URL-адреса с помощью `LinkGenerator` или метода `RedirectToRoute()`, следует соблюдать осторожность при использовании этих методов генерации маршрутов. Обязательно укажите правильное имя конечной точки и все необходимые параметры маршрута. Если вы сделаете что-то не так – например, если у вас опечатка в имени конечной точки или вы забыли включить обязательный параметр маршрута, – сгенерированный URL-адрес будет нулевым. Иногда стоит явно проверить сгенерированный адрес на наличие значения `null`, чтобы убедиться в отсутствии проблем.

6.4.3 Управление созданными URL-адресами с помощью `RouteOptions`

Маршруты конечных точек – это общедоступная поверхность ваших API, поэтому у вас вполне может сложиться мнение о том, как они должны выглядеть. По умолчанию `LinkGenerator` делает все возможное, чтобы генерировать маршруты так же, как вы их определяете; если вы определяете конечную точку с помощью шаблона маршрута `/MyRoute`, `LinkGenerator` генерирует путь `/MyRoute`. Но что, если этот путь вам не нужен? Что, если вы предпочитаете, чтобы `LinkGenerator` создавал более красивые пути, например `/мурoute` или `/мурoute/`? В этом разделе вы узнаете, как настроить генерацию URL-адресов как глобально, так и для каждого конкретного случая.

ПРИМЕЧАНИЕ Добавлять ли косую черту в конце URL-адреса – это во многом дело вкуса, но этот выбор имеет некоторые последствия с точки зрения удобства использования и результатов поиска. Обычно я добавляю косую черту в конце для приложений Razor Pages, но не для API. Подробности см. на странице <http://mng.bz/Ao1W>.

Когда ASP.NET Core сопоставляет входящий URL-адрес с вашими шаблонами маршрутов с помощью маршрутизации, он использует

сравнение без учета регистра, как вы видели в главе 5. Таким образом, если у вас есть шаблон маршрута /MyRoute, запросы к /myroute, /MYROUTE, и даже /myROUTE совпадают. Но при создании URL-адресов LinkGenerator необходимо выбрать один вариант для использования. По умолчанию используется тот же регистр, который вы определили в шаблонах маршрутов. Поэтому если вы напишете

```
app.MapGet("/MyRoute", () => "Hello world!").WithName("route1");
```

LinkGenerator.GetPathByName("route1") вернет /MyRoute.

Хотя это хороший вариант по умолчанию, вы, вероятно, предпочитаете, чтобы все ссылки, создаваемые вашим приложением, были единобразными. Мне нравится, чтобы все мои ссылки были в нижнем регистре, даже если я случайно указал шаблон маршрута не строчными буквами.

Правилами генерации маршрутов можно управлять с помощью RouteOptions. Чтобы сконфигурировать его для своего приложения, используется метод расширения Configure<T> в WebApplicationBuilder.Services, который обновляет экземпляр RouteOptions для приложения с помощью системы конфигурации.

ПРИМЕЧАНИЕ Вы узнаете все о системе конфигурации и методе Configure<T> в главе 10.

RouteOptions содержит несколько параметров конфигурации, как показано в листинге 6.4. Эти настройки определяют, должны ли URL-адреса, генерируемые приложением, быть строчными, должна ли строка запроса также быть строчной и должна ли добавляться косая черта (/) к конечным URL-адресам. В листинге я задал, чтобы URL-адрес был в нижнем регистре, добавлялась косая черта и чтобы строка запроса оставалась неизменной.

ПРИМЕЧАНИЕ В листинге 6.4 весь путь указан в нижнем регистре, включая любые сегменты параметров маршрута, например {name}. Только строка запроса сохраняет исходный регистр.

Листинг 6.4. Настройка генерации ссылок с помощью RouteOptions

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.Configure<RouteOptions>(o =>
{
    o.LowercaseUrls = true;                                | Настраивает RouteOptions,
    o.AppendTrailingSlash = true;                           | используемый для генера-
    o.LowercaseQueryStrings = false; <--                  | ции ссылок
});                                                       | По умолчанию для всех настроек
                                                        | задано значение false

WebApplication app = builder.Build();
app.MapGet("/HealthCheck", () => Results.Ok()).WithName("healthcheck");
app.MapGet("/{name}", (string name) => name).WithName("product");

app.MapGet("/", (LinkGenerator links) =>
new []
```

```

{
    links.GetPathByName("healthcheck"), <-- Возвращает /healthcheck/
    links.GetPathByName("product",
        new { Name = "Big-Widget", Q = "Test"}) | Возвращает
    );} /big-widget/?Q=Test

app.Run();

```

Какие бы параметры по умолчанию вы ни выбрали, следует попытаться использовать их во всем приложении, но в некоторых случаях это может быть невозможно. Например, у вас может быть устаревший API, который вам нужно имитировать, и вы не можете использовать URL-адреса в нижнем регистре. В этих случаях можно переопределить значения по умолчанию, передав необязательный параметр `LinkOptions` методам `LinkGenerator`. Значения, заданные в `LinkOptions`, переопределяют значения по умолчанию, заданные в `RouteOptions`. При создании ссылки для приложения в листинге 6.4 с помощью следующего кода

```

links.GetPathByName("healthcheck",
    options: new LinkOptions
    {
        LowercaseUrls = false,
        AppendTrailingSlash = false,
    });

```

вернется значение `/HealthCheck`. Без параметра `LinkOptions GetPathByName` вернет `/healthcheck/`.

Поздравляю: вы прошли весь путь, посвященный подробному обсуждению маршрутизации! Маршрутизация – одна из тех тем, на которых часто застревают при создании приложения, и это может расстраивать. Мы вернемся к маршрутизации при рассмотрении Razor Pages в главе 14 и контроллеров веб-API в главе 20, но будьте уверены, что в этой главе раскрыты все сложные детали!

В главе 7 мы подробно изучим привязку модели. Вы увидите, как значения маршрута, сгенерированные во время маршрутизации, привязываются к параметрам обработчика конечной точки и, что, возможно, более важно, как проверять предоставленные вами значения.

Резюме

- Маршрутизация – это процесс сопоставления URL-адреса входящего запроса с конечной точкой, которая будет выполняться для генерации ответа. Маршрутизация обеспечивает гибкость реализации API, позволяя, например, сопоставить несколько URL-адресов с одной конечной точкой;
- ASP.NET Core использует два компонента промежуточного ПО для маршрутизации: `EndpointRoutingMiddleware` и `EndpointMiddleware`. По умолчанию `WebApplication` добавляет оба компонента в конвейер, поэтому обычно они не добавляются в приложение вручную;

- шаблоны маршрутов определяют структуру известных URL-адресов в вашем приложении. Это строки с заполнителями для переменных, которые могут содержать необязательные значения и сопоставляться с обработчиками конечных точек. Следует тщательно продумывать свои маршруты, поскольку они являются общедоступным интерфейсом вашего приложения;
- параметры маршрута – это значения переменных, извлекаемые из URL-адреса запроса. Можно использовать параметры маршрута для сопоставления нескольких URL-адресов с одной конечной точкой и автоматического извлечения значения переменной из URL-адреса;
- параметры маршрута могут быть необязательными и могут использовать значения по умолчанию, если значение отсутствует. Вам следует применять необязательные параметры и параметры по умолчанию с осторожностью, поскольку они могут усложнить понимание ваших API, но в некоторых случаях они могут быть полезны. Необязательные параметры должны быть последним сегментом маршрута;
- параметры маршрута могут иметь ограничения, которые ограничивают возможные допустимые значения. Если параметр маршрута не соответствует ограничениям, маршрут не считается совпадающим. Данный подход может помочь устраниТЬ неоднозначность между двумя похожими маршрутами, но не следует использовать ограничения для валидации;
- применяйте универсальный параметр, чтобы записать оставшуюся часть URL-адреса в значение маршрута. В отличие от стандартных параметров маршрута, универсальные параметры могут содержать косую черту (/) в значениях;
- можно использовать инфраструктуру маршрутизации для создания внутренних URL-адресов приложения. Такой подход гарантирует, что все ваши ссылки останутся правильными, если вы измените шаблоны маршрутов конечной точки;
- `LinkGenerator` можно использовать для генерации URL-адресов из конечных точек минимальных API. Укажите имя конечной точки для ссылки и все необходимые значения маршрута для создания соответствующего URL-адреса;
- можно использовать метод `RedirectToRoute` для создания URL-адресов, а также для создания ответа на перенаправление. Такой подход полезен, когда не нужно ссылаться на URL-адрес в коде;
- по умолчанию URL-адреса генерируются с использованием того же регистра, что и шаблон маршрута, и любых предоставленных параметров маршрута. Вместо этого можно принудительно использовать URL-адреса в нижнем регистре, строки запроса в нижнем регистре и конечную косую черту, настроив `RouteOptions` с помощью вызова `builder.Services.Configure<RouteOptions>()`;
- можно изменить настройки для создания одного URL-адреса, передав объект `LinkOptions` методам `LinkGenerator`. Эти методы могут быть полезны, когда вам нужен отличный от значений по умолчанию вариант для одной конечной точки, например когда вы пытаетесь сопоставить существующий устаревший маршрут.



7

Привязка модели и валидация в минимальных API

В этой главе:

- использование значений из запроса для создания моделей привязки;
- настройка процесса привязки модели;
- проверка пользовательского ввода с помощью атрибутов DataAnnotations.

В предыдущей главе было показано, как определить маршрут с параметрами – например, для уникального идентификатора в API продукта. Но предположим, что клиент отправляет запрос к API продукта. Что тогда? Как получить доступ к значениям, указанным в запросе, и прочитать JSON в теле запроса?

Большую часть этой главы, в разделах 7.1–7.9, мы будем рассматривать привязку модели и то, как она упрощает чтение данных из запроса в минимальных API. Вы увидите, как взять данные, переданные в теле запроса или в URL-адресе, и привязать их к объектам C#, которые затем передаются методам обработчика конечной точки в качестве аргументов. Когда обработчик выполняется, он может использовать эти

значения, чтобы сделать что-то полезное, например вернуть сведения о продукте или изменить его название.

Когда ваш код выполняется в методе-обработчике конечной точки, простиительно думать, будто вы можете успешно использовать модель привязки без каких-либо дальнейших размышлений. Хотя подождите. Откуда взялись эти данные? От пользователя – а вы знаете, что пользователям нельзя доверять! Раздел 7.10 посвящен тому, как убедиться, что предоставленные пользователем значения являются допустимыми и имеют смысл для приложения.

Привязка модели – это процесс получения необработанного HTTP-запроса пользователя и предоставления его вашему коду путем заполнения объектов РОСО, предоставляя входные данные обработчикам конечных точек. Начнем с рассмотрения того, какие значения в запросе доступны для привязки и где привязка модели подходит для вашего работающего приложения.

7.1 Извлечение значений из запроса с привязкой модели

В главах 5 и 6 вы узнали, что параметры маршрута можно извлечь из пути запроса и использовать для выполнения обработчиков минимальных API. В этом разделе мы более подробно рассмотрим процесс извлечения параметров маршрута и концепцию привязки модели.

К настоящему моменту вы должны быть знакомы с тем, как ASP.NET Core обрабатывает запрос, выполняя обработчик конечной точки. Вы также уже видели несколько обработчиков, подобных этому:

```
app.MapPost("/square/{num}", (int num) => num * num);
```

Обработчики конечных точек – это обычные методы C#, поэтому фреймворк ASP.NET Core должен иметь возможность вызывать их обычным способом. Когда обработчики принимают параметры как часть сигнатуры метода, например `num` в предыдущем примере, фреймворку нужен способ генерировать эти объекты. Откуда они берутся и как создаются?

Я уже намекал, что в большинстве случаев эти значения берутся из самого запроса. Но HTTP-запрос, который получает сервер, представляет собой набор строк. Как ASP.NET Core превращает это в объект .NET? Здесь на помощь приходит привязка модели.

ОПРЕДЕЛЕНИЕ *Привязка модели* извлекает значения из запроса и использует их для создания .NET-объектов. Эти объекты передаются в качестве параметров метода исполняемому обработчику конечной точки.

Этот механизм отвечает за просмотр поступившего запроса и поиск значений, которые будут использоваться. Затем он создает объекты соответствующего типа и присваивает эти значения вашей модели в ходе процесса, называемого *привязкой*.

ПРИМЕЧАНИЕ Привязка модели в минимальных API (и в Razor Pages, и в MVC) – это одностороннее заполнение объектов из запроса, а не двусторонняя привязка данных, которая иногда используется при разработке настольных или мобильных приложений.

ASP.NET Core автоматически создает аргументы, которые передаются обработчику, используя свойства запроса, например URL-адрес запроса, любые заголовки, отправленные в HTTP-запросе, любые данные, явно отправленные с помощью метода POST в теле запроса, и т. д.

Привязка модели происходит до выполнения конвейера фильтра и обработчика конечной точки в `EndpointMiddleware`, как показано на рис. 7.1. Компонент `RoutingMiddleware` отвечает за сопоставление входящего запроса с конечной точкой и за извлечение значений параметров маршрута, но все значения в этой точке являются строками. Только в `EndpointMiddleware` строковые значения преобразуются в реальные типы аргументов (например, `int`), необходимые для выполнения обработчика конечной точки.

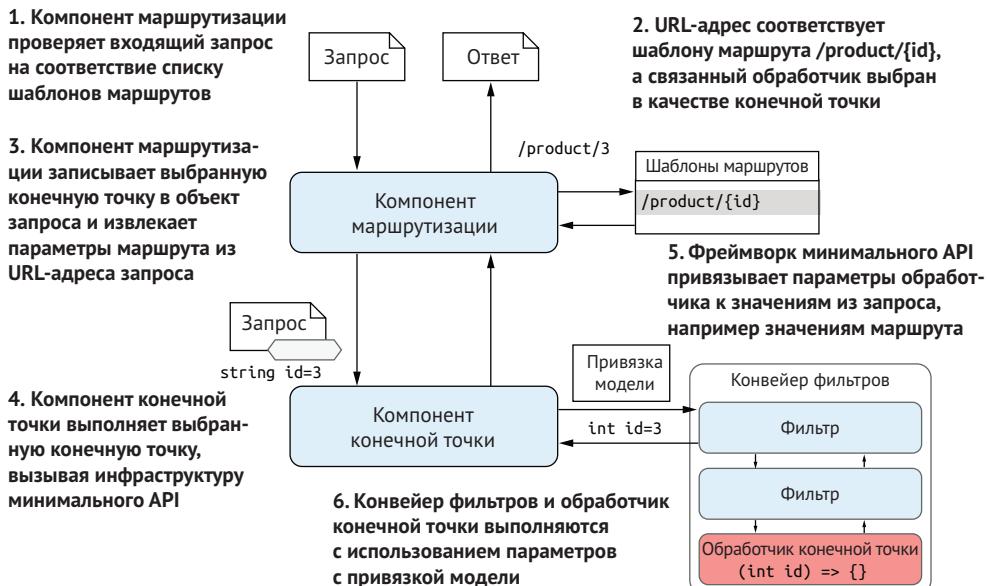


Рис. 7.1 `RoutingMiddleware` сопоставляет входящий запрос с конечной точкой и извлекает параметры маршрута в виде строк. Когда `EndpointMiddleware` выполняет конечную точку, инфраструктура минимального API использует привязку модели для создания аргументов, необходимых для выполнения обработчика конечной точки, преобразуя строковые значения маршрута в реальные типы аргументов, например `int`

Для каждого параметра в обработчике конечной точки минимального API ASP.NET Core должен решить, как создать соответствующие аргументы. Минимальные API могут использовать шесть различных источников привязки для создания аргументов обработчика:

- *значения маршрута* – эти значения получаются из сегментов URL-адресов или значений по умолчанию после сопоставления маршрута, как было показано в главе 5;
- *значения строки запроса* – эти значения передаются в конце URL-адреса и не используются во время маршрутизации;
- *значения заголовков* – значения заголовков предоставляются в HTTP-запросе;
- *тело JSON* – к телу JSON запроса может быть привязан только один параметр;
- *сервисы внедрения зависимостей* – сервисы, доступные посредством внедрения зависимостей, можно использовать в качестве аргументов обработчика конечной точки. Мы рассмотрим внедрение зависимостей в главах 8 и 9;
- *собственная привязка* – ASP.NET Core предоставляет методы, позволяющие настроить привязку типа, предоставляя доступ к объекту `HttpRequest`.

ВНИМАНИЕ! В отличие от контроллеров MVC и страниц Razor Pages, минимальные API не привязываются автоматически к телу запросов, отправленных в виде форм, с использованием mime-типа `application/x-www-form-urlencoded`. Минимальные API будут привязываться только к телу запроса JSON. Если вам нужно работать с данными формы в конечной точке минимального API, можно получить к ним доступ через `HttpRequest.Form`, но автоматическая привязка не принесет вам пользы.

В разделе 7.8 мы рассмотрим точный алгоритм, который использует ASP.NET Core для выбора источника привязки, но для начала разберем, как ASP.NET Core привязывает простые типы, например `int` и `double`.

7.2 Привязка простых типов к запросу

При создании обработчиков минимальных API часто требуется извлечь простое значение из запроса. Например, если вы загружаете список продуктов по категории, то вам, скорее всего, понадобится идентификатор категории, а в примере с калькулятором в начале раздела 7.1 вам понадобится возвести число в квадрат.

Когда вы создаете обработчик конечной точки, содержащий простые типы, например `int`, `string` и `double`, ASP.NET Core автоматически пытается привязать значение к параметру маршрута или значению строки запроса:

- если имя параметра обработчика совпадает с именем параметра маршрута в шаблоне маршрута, ASP.NET Core выполняет привязку к связанному значению маршрута;
- если имя параметра обработчика не соответствует ни одному параметру в шаблоне маршрута, ASP.NET Core пытается выполнить привязку к значению строки запроса.

Например, если вы сделаете запрос к `/products/123`, то это будет соответствовать следующей конечной точке:

```
app.MapGet("/products/{id}", (int id) => $"Received {id}");
```

ASP.NET Core привязывает аргумент обработчика `id` к параметру маршрута `{id}`, поэтому функция обработчика вызывается с `id=123`. И наоборот, если вы сделаете запрос к `/products?id=456`, он будет соответствовать следующей конечной точке:

```
app.MapGet("/products", (int id) => $"Received {id}");
```

В этом случае в шаблоне маршрута нет параметра `id`, поэтому вместо этого ASP.NET Core выполняет привязку к строке запроса, а функция-обработчик вызывается с `id=456`.

В дополнение к этому «автоматическому» выводу можно заставить ASP.NET Core выполнить привязку из определенного источника, добавив атрибуты к параметрам. `[FromRoute]` явно привязывается к параметрам маршрута, `[FromQuery]` – к строке запроса, а `[FromHeader]` – к значениям заголовка, как показано на рис. 7.2.

Привязка модели сопоставляет

значения из HTTP-запроса с параметрами в обработчике конечной точки.
Строковые значения из запроса автоматически преобразуются в тип параметра конечной точки

Параметры маршрута автоматически сопоставляются с соответствующими параметрами конечной точки, или можно сделать это явно

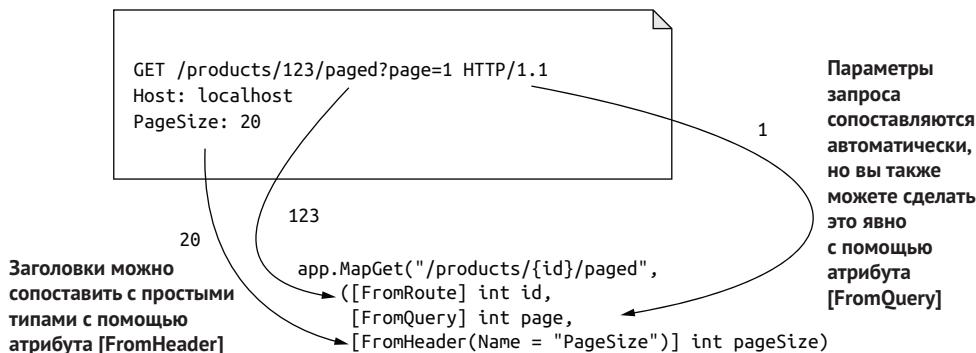


Рис. 7.2 Модель привязки с HTTP-методом GET. Атрибуты `[FromRoute]`, `[FromQuery]` и `[FromHeader]` заставляют параметры конечной точки привязываться к определенным частям запроса. В данном случае требуется только атрибут `[FromHeader]`; параметр маршрута и строка запроса будут выведены автоматически

Атрибуты `[From*]` переопределяют логику ASP.NET Core по умолчанию и принудительно загружают параметры из определенного источника привязки. В листинге 7.1 показаны три возможных атрибута `[From*]`:

- `[FromQuery]` – как вы уже видели, этот атрибут принудительно привязывает параметр к строке запроса;
- `[FromRoute]` – этот атрибут заставляет параметр привязать значение параметра маршрута. Обратите внимание: если параметр с требуемым именем не существует в шаблоне маршрута, во время выполнения вы получите исключение;
- `[FromHeader]` – этот атрибут привязывает параметр к значению заголовка в запросе.

Листинг 7.1. Привязка простых значений с помощью атрибутов [From]

```
using Microsoft.AspNetCore.Mvc; <-- Все атрибуты [From*] находятся
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/products/{id}/paged",
    ([FromRoute] int id, <-- [FromRoute] принудительно при-
    [FromHeader] привязывает <--вязывает аргумент к значению
    аргумент <--маршрута
    к указанному <--[FromHeader(Name = "PageSize")] int pageSize)
    заголовку <--=> $"Received id {id}, page {page}, pageSize {pageSize}");

app.Run(); <-- [FromQuery] принудительно привязывает
          аргумент к строке запроса
```

Позже вы увидите другие атрибуты, например [FromBody] и [FromServices], но предыдущие три атрибута – единственные атрибуты [From*], которые работают с простыми типами, такими как `int` и `double`. Я предпочитаю избегать использования [FromQuery] и [FromRoute] везде, где это возможно, и вместо этого полагаться на соглашения о привязке по умолчанию, поскольку считаю, что они засоряют сигнатуры методов, и обычно очевидно, будет ли простой тип привязываться к строке запроса или значению маршрута.

СОВЕТ ASP.NET Core выполняет привязку к параметрам маршрута и значениям строки запроса на основе соглашения, но единственный способ привязки к значению заголовка – атрибут [FromHeader].

Возможно, вам интересно, что произойдет, если попытаться привязать тип к несовместимому значению. Что, если вы попытаетесь связать целое число, например, со строковым значением "two"? В этом случае ASP.NET Core выдает исключение `BadHttpRequestException` и возвращает ответ `400 Bad Request`.

ПРИМЕЧАНИЕ Когда инфраструктура минимального API не может привязать параметр обработчика из-за несовместимого формата, она выдает исключение `BadHttpRequestException` и возвращает ответ `400 Bad Request`.

В этом разделе я несколько раз упоминал, что значения маршрутов, значения строк запроса и заголовки можно привязывать к простым типам, но что такое простой тип? *Простой тип* определяется как любой тип, который содержит любой из следующих методов `TryParse`, где `T` – реализующий тип:

```
public static bool TryParse(string value, out T result);
public static bool TryParse(
    string value, IFormatProvider provider, out T result);
```

Такие типы, как `int` и `bool`, содержат один или оба этих метода. Но также стоит отметить, что можно создавать свои собственные типы, реализующие один из этих методов, и они будут рассматриваться

как простые типы, способные выполнять привязку на основе значений маршрута, значений строки запроса и заголовков.

На рис. 7.3 показан пример реализации простого строго типизированного идентификатора ID¹, который рассматривается как простой тип благодаря предоставляемому им методу TryParse. Когда вы отправляете запрос на /product/p123, ASP.NET Core видит, что тип ProductId, используемый в обработчике конечной точки, содержит метод TryParse и что имя параметра id имеет соответствующее имя параметра маршрута. Он создает аргумент id, вызывая ProductId.TryParse(), и передает значение маршрута p123.

В листинге 7.2 показано, как реализовать метод TryParse для ProductId. Этот метод создает ProductId из строк, состоящих из целого числа с префиксом 'p' (например, p123 или p456). Если входная строка соответствует требуемому формату, она создает экземпляр ProductId и возвращает true. Если формат недействителен, она возвращает false, привязка завершается неудачей и возвращается ответ 400 Bad Request.

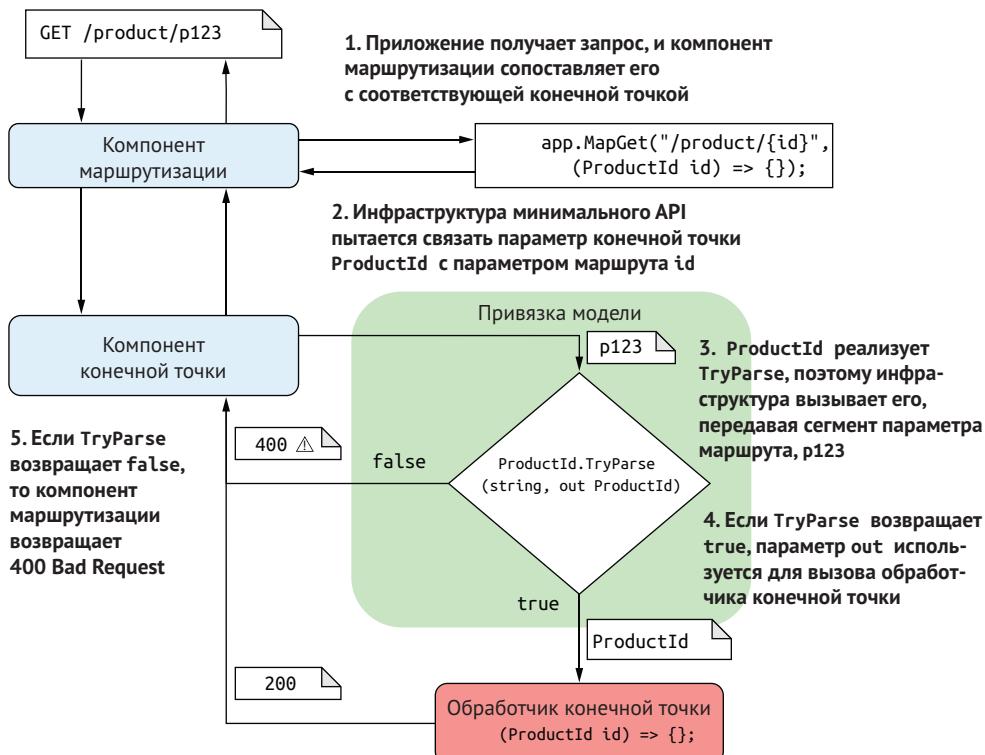


Рис. 7.3 Компонент маршрутизации сопоставляет входящий URL-адрес с конечной точкой. Компонент конечной точки пытается связать идентификатор параметра маршрута с параметром конечной точки. Тип параметра конечной точки ProductId реализует TryParse. Если парсинг прошел успешно, то параметр parsed используется для вызова обработчика конечной точки. В случае неудачи компонент конечной точки возвращает ответ 400 Bad Request

¹ В моем блоге на странице <http://mng.bz/a1Kz> есть серия статей, посвященных строго типизированным идентификаторам и их преимуществам.

Листинг 7.2. Реализация TryParse в собственном типе, позволяющая выполнить разбор значений маршрута

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/product/{id}", (ProductId id) => $"Received {id}");
app.Run();
```

ProductId – это record struct, которые появились в C# 10

→ readonly record struct ProductId(int Id)

{

→ public static bool TryParse(string? s, out ProductId result)

Реализует TryParse, поэтому минимальные API рассматривают его как простой тип

Эффективно пропускает первый символ, рассматривая строку как ReadOnlySpan

}

if(s is not null
 && s.StartsWith('p')
 && int.TryParse(
 s.AsSpan().Slice(1),
 out int id))

{
 result = new ProductId(id);
 return true;
}

result = default;
return false;

ProductId автоматически привязывается к значениям маршрута, поскольку реализует TryParse

Проверяет, что строка не равна нулю и что первый символ в строке — это 'p'...

и если это так, пытается проанализировать оставшиеся символы как целое число

Если строка была проанализирована успешно, id содержит проанализированное значение

Все успешно проанализировано, поэтому создается новый ProductId и возвращается true

Что-то пошло не так, поэтому возвращает false и присваивает значение по умолчанию (неиспользуемому) результату

Использование современных функций C# и .NET

Листинг 7.2 включает в себя ряд функций C# и .NET, с которыми вы, возможно, раньше не встречались, в зависимости от вашего опыта:

- **сопоставление с образцом для нулевых значений – s is not null.** Функции сопоставления с образцом постепенно вводились в C#, начиная с C# 7. Шаблон `is not null`, представленный в C# 9, имеет некоторые незначительные преимущества по сравнению с обычным выражением `!= null`. Все о сопоставлении с образцом можно прочитать на странице <http://mng.bz/gBxI>;
- **записи (Record) и записи структур (record struct) – `readonly record struct`.** Записи – это синтаксический сахар по сравнению с обычными объявлениями классов и структур, который делает объявление новых типов более кратким и предоставляет удобные методы для работы с неизменяемыми типами. Записи структур появились в C# 10. Дополнительную информацию можно найти на странице <http://mng.bz/5wWz>;
- **Span<T> для повышения производительности – s.AsSpan().** `Span<T>` и `ReadOnlySpan<T>` были представлены в .NET Core 2.1 и особенно полезны для сокращения выделения памяти при работе со строковыми значениями. Подробнее о них можно прочитать на странице <http://mng.bz/6DNy>;

- `ValueTask<T>` – он не показан в листинге 7.2, но многие API в ASP.NET Core используют `ValueTask` вместо более распространенной `Task` для API, которые обычно выполняются асинхронно. О том, почему они были введены и когда их следует использовать, можно прочитать на странице <http://mng.bz/o1GM>.

Если вы хотите использовать новые функции, то можно рассмотреть возможность реализации интерфейса `IParsable` при реализации `TryParse`. Данный интерфейс использует статические абстрактные интерфейсы, представленные в C# 11, и требует реализаций методов `TryParse` и `Parse`. Подробнее об интерфейсе `IParsable` можно прочитать в посте на странице <http://mng.bz/nW2K>.

Мы подробно рассмотрели привязку простых типов к значениям маршрутов, строкам запроса и заголовкам. В разделе 7.3 мы познакомимся с привязкой к телу запроса путем десериализации JSON в сложные типы.

7.3 Привязка сложных типов к телу запроса в формате JSON

Привязка модели в минимальных API основана на определенных соглашениях, упрощающих код, который вам нужно писать. Одно из таких соглашений, которое вы уже видели, касается привязки к параметрам маршрута и значениям строки запроса. Еще одно важное соглашение заключается в том, что конечные точки минимальных API предполагают, что запросы будут отправляться с использованием формата JSON.

Минимальные API могут привязать тело запроса к одному сложному типу в обработчике конечной точки путем десериализации запроса из JSON. Это означает, что если у вас есть конечная точка, как та, что показана в следующем листинге, то ASP.NET Core автоматически десериализует для вас запрос из JSON, создав аргумент `Product`.

Листинг 7.3. Автоматическая десериализация запроса в формате JSON из тела

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/product", (Product product) => $"Received {product}");
app.Run();
```

Product – это сложный тип, поэтому он привязан к телу запроса в формате JSON

⇒ record Product(int Id, string Name, int Stock);
Product не реализует TryParse, поэтому это сложный тип

Если вы отправляете запрос методом POST в `/product` для приложения из листинга 7.3, необходимо указать действительный JSON в теле запроса, например:

```
{ "id": 1, "Name": "Shoes", "Stock": 12 }
```

ASP.NET Core использует встроенную библиотеку System.Text.Json для десериализации JSON в экземпляр Product и применяет его в качестве аргумента product в обработчике.

Настройка привязки JSON с помощью System.Text.Json

Библиотека System.Text.Json, представленная в .NET Core 3.0, предоставляет высокопроизводительную библиотеку сериализации JSON с низким объемом выделения памяти. Она была разработана как своего рода преемник вездесущей библиотеки Newtonsoft.Json, но в ней гибкость сочетается с производительностью.

Минимальные API используют System.Text.Json как для десериализации JSON (при привязке к телу запроса), так и для сериализации (при записи результатов, как вы видели в главе 6).

В отличие от MVC и Razor Pages, нельзя заменить библиотеку сериализации JSON, используемую минимальными API, поэтому нет возможности использовать Newtonsoft.Json. Но можно настроить некоторые параметры сериализации библиотеки для своих минимальных API.

Например, можно настроить System.Text.Json, чтобы ослабить ограничения и разрешить конечные запятые в JSON, контролировать сериализацию имен свойств с помощью кода, как, например, здесь:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.ConfigureRouteHandlerJsonOptions(o => {
    o.SerializerOptions.AllowTrailingCommas = true;
    o.SerializerOptions.PropertyNamingPolicy =
        JsonNamingPolicy.CamelCase;
    o.SerializerOptions.PropertyNameCaseInsensitive = true;
});
```

Обычно автоматическая привязка запросов в формате JSON удобна, поскольку большинство современных API построены на запросах и ответах в формате JSON. Встроенная привязка использует наиболее производительный подход и устраниет большое количество шаблонного кода, который в противном случае вам пришлось бы писать самостоятельно. Тем не менее при привязке к телу запроса имейте в виду следующее:

- к телу запроса в формате JSON можно привязать только один параметр обработчика. Если к нему привязать несколько сложных параметров, вы получите исключение во время выполнения, когда приложение получит свой первый запрос;
- если формат тела запроса отличается от JSON, обработчик конечной точки не запустится, и EndpointMiddleware вернет ответ 415 Unsupported Media Type;
- если вы попытаетесь привязать тело HTTP-метода, у которого обычно нет тела (GET, HEAD, OPTIONS, DELETE, TRACE и CONNECT), то получите исключение во время выполнения. Если вы измените конечную точку из листинга 7.3, например используете MapGet вместо MapPost, то получите исключение при первом запросе, как показано на рис. 7.4;

- если вы уверены, что хотите привязать тело этих запросов, то можете переопределить поведение по умолчанию, применив атрибут [FromBody] к параметру обработчика. Однако я настоятельно не советую использовать этот подход: отправка тела с запросами методом GET необычна, может сбить с толку потребителей вашего API и не рекомендуется в спецификации HTTP (<https://www.rfc-editor.org/rfc/rfc9110#name-get>);
- это необычно, но вы также можете применить атрибут [FromBody] к параметру простого типа, чтобы заставить его привязаться к телу запроса, а не к строке маршрута или запроса. Что касается сложных типов, то тело десериализуется из JSON в ваш параметр.

Мы обсудили привязку простых и сложных типов. К сожалению, сейчас пришло время признать наличие запутанной ситуации – это массивы, которые могут быть простыми или сложными типами.

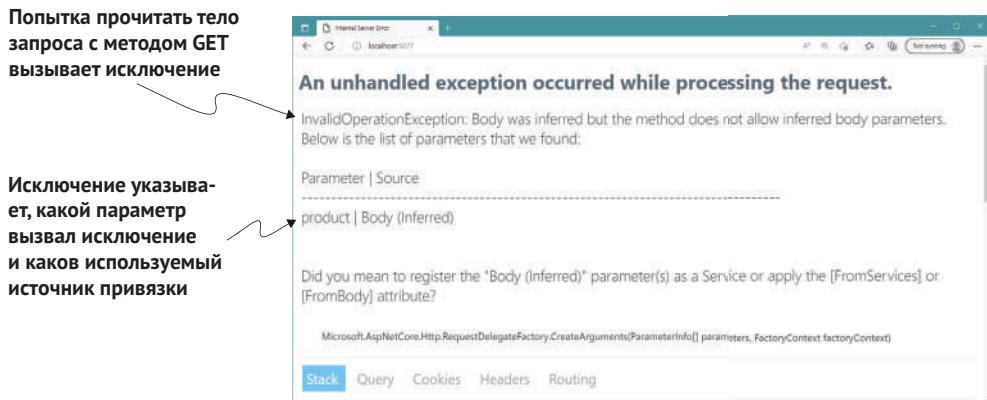


Рис. 7.4 Если вы попытаетесь привязать тело к параметру запроса с методом GET, то получите исключение, когда приложение получит свой первый запрос.

7.4 Массивы: простые типы или сложные?

Малоизвестный факт: записи в строке запроса URL-адреса не обязательно должны быть уникальными. Например, следующий URL-адрес является допустимым, даже если содержит повторяющийся параметр id:

```
/products?id=123&id=456
```

Так как же получить доступ к этим значениям строки запроса с помощью минимального API? Если вы создаете конечную точку, например

```
app.MapGet("/products", (int id) => $"Received {id}");
```

то запрос к /products?id=123 привяжет параметр id к строке запроса, как и следовало ожидать. Но запрос, который включает в строку запроса два значения id, например /products?id=123&id=456, вызовет ошибку времени выполнения, как показано на рис. 7.5. ASP.NET Core возвращает ответ 400 Bad Request без запуска обработчика или конвейера фильтров.

Если вы хотите обрабатывать строки запроса, подобные этой, чтобы пользователи могли при необходимости передавать несколько возможных значений параметра, необходимо использовать массивы. В следующем листинге показан пример конечной точки, которая принимает несколько значений идентификатора из строки запроса и привязывает их к массиву.

Листинг 7.4. Привязка нескольких значений параметра в строке запроса к массиву

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/products/search",
    (int[] id) => $"Received {id.Length} ids"); <-- Массив будет при-
    app.Run();                                вязан к несколь-
                                                ким экземплярам
                                                id в строке запроса
```

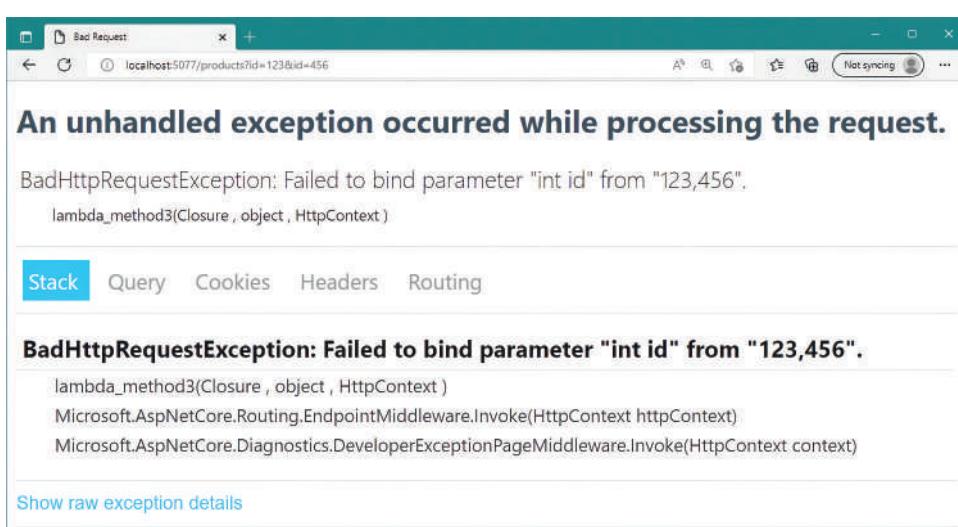


Рис. 7.5 Попытка привязать обработчик с сигнатурой, например `(int id)`, к строке запроса, содержащей `?id=123&id=456`, вызывает исключение во время выполнения и ответ 400 Bad Request.

Если вы чем-то похожи на меня, то тот факт, что параметр обработчика `int[]` из листинга 7.4 называется `id`, а не `ids`, вас действительно расстроит. К сожалению, здесь необходимо использовать `id`, чтобы параметр правильно привязался к строке запроса, например `?id=123&id=456`. Если вы переименовали его в `ids`, то строка запроса должна выглядеть так: `?ids=123&ids=456`.

К счастью, есть другой вариант. Можно управлять именем параметра, к которому привязывается обработчик, используя атрибуты `[FromQuery]` и `[FromRoute]` аналогично тому, как вы используете `[FromHeader]`.

В данном примере вы можете получить лучшее из обоих миров, переименовав идентификаторы параметров обработчика и добавив атрибут [FromQuery]:

```
app.MapGet("/products/search",
    ([FromQuery(Name = "id")] int[] ids) => $"Received {ids.Length} {ids}");
```

Теперь можете спать спокойно. У параметра обработчика имя корректнее, и он по-прежнему правильно привязывается к строке запроса `?id=123&id=456`.

СОВЕТ Можно привязать параметры массива к нескольким значениям заголовка так же, как и к значениям строки запроса, используя атрибут [FromHeader].

В примере из листинга 7.4 мы привязываем `int[]`, но вы можете привязать массив любого простого типа, включая собственные типы, с помощью метода TryParse (листинг 7.2), а также `string[]` и `StringValues`.

ПРИМЕЧАНИЕ `StringValues` – это вспомогательный тип в пространстве имен `Microsoft.Extensions.Primitives`, который эффективно представляет ноль, одну или множество строк.

Так где же та непонятная ситуация, о которой я упомянул? Массивы работают, как я и описывал, только если:

- вы используете HTTP-метод, который обычно не включает тело запроса, например `GET`, `HEAD` или `DELETE`;
- массив представляет собой массив простых типов (или `string[]` или `StringValues`).

Если какое-либо из этих утверждений неверно, то ASP.NET Core попытается привязать массив к телу запроса в формате JSON. Для запросов методом `POST` (или других методов, которые обычно имеют тело запроса) этот процесс работает без проблем: тело в формате JSON десериализуется в массив параметров. Для запросов методом `GET` (и других методов без тела) это вызывает то же необработанное исключение, которое показано на рис. 7.4, когда в одном из этих методов обнаруживается привязка тела.

ПРИМЕЧАНИЕ Как и раньше, при привязке параметров тела можно обойти эту ситуацию для `GET`-запросов, добавив явный атрибут [FromBody] к параметру обработчика, но не следует этого делать!

Мы рассмотрели привязку простых и сложных типов из URL-адреса и тела и даже рассмотрели некоторые случаи, в которых несоответствие между тем, чего вы ожидаете, и тем, что вы получаете, приводит к ошибкам. Но что, если ожидаемого значения нет? В разделе 7.5 мы рассмотрим, как можно выбрать поведение для такой ситуации.

7.5 Делаем параметры необязательными с помощью типов, допускающих значение NULL

Мы описали множество способов привязки параметров к конечным точкам минимального API. Если вы экспериментировали с примерами кода и отправкой запросов, то, возможно, заметили, что если конечная точка не может привязать параметр во время выполнения, вы получаете ошибку и ответ 400 Bad Request. Если у вас есть конечная точка, которая привязывает параметр к строке запроса, например

```
app.MapGet("/products", (int id) => $"Received {id}");
```

но вы отправляете запрос без строки запроса или с неправильным именем в строке, например запрос к /products?p=3, EndpointMiddleware выдает исключение, как показано на рис. 7.6. Параметр id является обязательным, поэтому если его нельзя привязать, вы получите сообщение об ошибке и ответ 400 Bad Request, а обработчик конечной точки не запустится.

Все параметры являются обязательными независимо от того, какой источник привязки они используют, будь то значение маршрута, значение строки запроса, заголовок или тело запроса. Но что, если вы хотите, чтобы параметр обработчика был необязательным? Если у вас есть конечная точка, подобная этой:

```
app.MapGet("/stock/{id?}", (int id) => $"Received {id}");
```

учитывая, что параметр маршрута помечен как необязательный, запросы к /stock/123 и /stock вызовут обработчик. Но в последнем случае значения id маршрута не будет, и вы получите ошибку, подобную той, что показана на рис. 7.6.

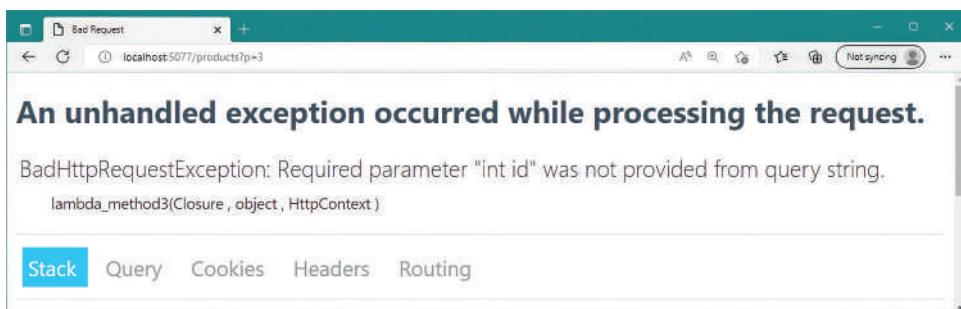


Рис. 7.6 Если параметр нельзя привязать из-за отсутствия значения, EndpointMiddleware выдаст исключение и возвращает ответ 400 Bad Request. Обработчик конечной точки не запускается.

Способ решения этой проблемы состоит в том, чтобы пометить *параметр обработчика* как необязательный, сделав его допускающим значение NULL. Равно как в шаблонах маршрутов символ ? означает необязательность, то же самое он обозначает и в параметрах обработ-

чика. Вы можете обновить обработчик, чтобы использовать `int?` вместо `int`, как показано в следующем листинге, и конечная точка будет обрабатывать `/stock/123` и `/stock` без ошибок.

Листинг 7.5. Использование необязательных параметров в обработчиках конечных точек

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
    Использует простой тип, допускающий значение NULL, чтобы указать, что значение является необязательным, поэтому при вызове /stock id имеет значение NULL
app.MapGet("/stock/{id?}", (int? id) => $"Received {id}"); ←
    → app.MapGet("/stock2", (int? id) => $"Received {id}");
    → app.MapPost("/stock", (Product? product) => $"Received {product}"); ←
app.Run();
```

В этом примере выполняется привязка к строке запроса. Идентификатор запроса /stock2 будет NULL

Сложный тип, допускающий значение NULL, привязывается к телу, если оно доступно; в противном случае он равен NULL

Если соответствующее значение маршрута или строка запроса не содержат требуемого значения, а параметр обработчика является необязательным, `EndpointHandler` использует значение `null` в качестве аргумента при вызове обработчика конечной точки. Аналогично для сложных типов, которые привязываются к телу запроса, если запрос ничего не содержит в теле и параметр является необязательным, обработчику будет передан аргумент `null`.

ВНИМАНИЕ! Если тело запроса содержит буквальное значение `null` в формате JSON и параметр обработчика помечен как необязательный, аргумент обработчика также будет иметь значение `null`. Если параметр не помечен как необязательный, вы получите ту же ошибку, как если бы у запроса не было тела.

Стоит отметить, что привязка сложных типов к телу запроса помечается как необязательная, с использованием аннотации *ссылочного типа, допускающей значение NULL*: `?.` Данные типы, представленные в C# 8, представляют собой попытку уменьшить проблему исключений с нулевыми ссылками в C#, в просторечии известную как «ошибка на миллиард долларов». См. <http://mng.bz/vneM>.

ASP.NET Core в .NET 7 создан с учетом того, что в вашем проекте активированы ссылочные типы, допускающие значение `NULL` (и они активированы по умолчанию во всех шаблонах), поэтому их стоит использовать везде, где только возможно. Если вы решите явно отключить их, вы можете обнаружить, что некоторые из ваших типов неожиданно помечены как необязательные, что может привести к ряду трудных для отладки ошибок.

COBET По возможности оставляйте ссылочные типы, допускающие значение NULL, активированными для конечных точек минимальных API. Если вы не можете использовать их для всего проекта, рассмотрите возможность их выборочной активации в файле Program.cs (или там, где вы добавляете конечные точки), добавив `#nullable enable` в начало файла.

Хорошая новость состоит в том, что ASP.NET Core включает в себя несколько встроенных в компилятор анализаторов для выявления проблем конфигурации, подобных описанным в этом разделе. Если у вас есть необязательный параметр маршрута, но вы забыли пометить соответствующий параметр обработчика как необязательный, например, такие интегрированные среды разработки (IDE), как Visual Studio, покажут подсказку, как видно на рис. 7.7, и вы получите предупреждение при сборке. Подробнее о встроенных анализаторах можно прочитать на странице <http://mng.bz/4DMV>.

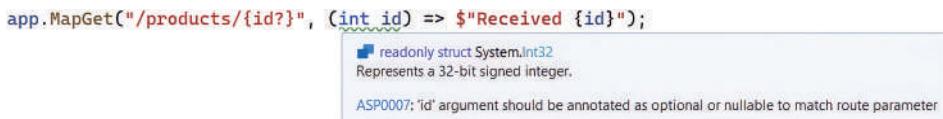


Рис. 7.7 Visual Studio и другие IDE используют анализаторы для обнаружения потенциальных проблем, связанных с конфигурацией

Сделать параметры обработчика необязательными – это один из подходов, который можно использовать независимо от того, привязаны ли они к параметрам маршрута, заголовкам или строке запроса. В качестве альтернативы можно указать значение по умолчанию для параметра как часть сигнатуры метода. В C# 11¹ невозможно указать значения по умолчанию для параметров в лямбда-функциях, поэтому в следующем листинге показано, как вместо этого использовать локальную функцию.

Листинг 7.6. Использование значений по умолчанию для параметров в обработчиках конечных точек

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/stock", StockWithValue); ← | Локальная функция
                                            | StockWithValue
                                            | является обработчиком
                                            | конечной точки

app.Run();      Параметр id привязывается к значению строки запроса, если
                | оно доступно; в противном случае он имеет значение 0
                | ← |
string StockWithValue(int id = 0) => $"Received {id}"; ← |
```

Мы подробно рассмотрели различия между простыми и сложными типами, а также то, как они связываются. В разделе 7.6 мы разберем специальные типы, которые не подчиняются этим правилам.

¹ Версия C# 12, которая будет выпущена вместе с .NET 8, должна включать поддержку значений по умолчанию в лямбда-выражениях. Подробнее об этом см. <http://mng.bz/AoRg>.

7.6 Привязка сервисов и специальных типов

В этом разделе вы узнаете, как использовать специальные типы, к которым можно выполнить привязку в обработчиках конечных точек. Говоря «специальные», я подразумеваю типы, которые вшиты в код или которые не создаются на основе деталей запроса, в отличие от привязки, которую вы видели до сих пор. В этом разделе рассматриваются три типа параметров:

- *хорошо известные типы* – т. е. вшитые в код типы, о которых известно ASP.NET Core, например `HttpContext` и `HttpRequest`;
- `IFormFileCollection` и `IFormFile` для работы с загрузкой файлов;
- сервисы приложений, зарегистрированные в `WebApplicationBuilder.Services`.

Мы начнем с рассмотрения известных типов, к которым можно выполнить привязку.

7.6.1 Внедрение хорошо известных типов

В этой книге вы видели примеры нескольких известных типов, которые можно внедрить в обработчики конечных точек, наиболее примечательным из которых является `HttpContext`. Остальные известные типы представляют ссылки для доступа к различным свойствам объекта `HttpContext`.

ПРИМЕЧАНИЕ Как описано в главе 3, `HttpContext` действует как хранилище всего, что связано с одним запросом. Он содержит доступ ко всем низкоуровневым сведениям о запросе и ответе, а также ко всем сервисам и функциям приложений, которые могут вам понадобиться.

Вы можете использовать известный тип в обработчике конечной точки, включив параметр соответствующего типа. Например, чтобы получить доступ к `HttpContext` в вашем обработчике, вы можете использовать

```
app.MapGet("/", (HttpContext context) => "Hello world!");
```

Можно применять в обработчиках конечных точек минимальных API следующие известные типы:

- `HttpContext` – этот тип содержит все сведения как о запросе, так и об ответе. Отсюда можно получить доступ ко всему, что вам нужно, но часто более простой способ получить доступ к общим свойствам – использовать один из других известных типов;
- `HttpRequest` – эквивалент свойства `HttpContext.Request`. Этот тип содержит все сведения только о запросе;
- `HttpResponse` – эквивалент свойства `HttpContext.Response`, этот тип содержит все сведения только об ответе;
- `CancellationToken` – эквивалент свойства `HttpContext.RequestAborted`. Этот токен срабатывает, если клиент прерывает запрос. Это полезно, если вам нужно отменить задачу с длительным временем выполнения, как описано в моем посте <http://mng.bz/QP2j>;

- `ClaimsPrincipal` – эквивалент свойства `HttpContext.User`. Этот тип содержит данные аутентификации пользователя. Подробнее об аутентификации вы узнаете в главе 23;
- `Stream` – эквивалент свойства `HttpRequest.Body`. Этот параметр является ссылкой на объект запроса `Stream`. Он может быть полезен в ситуациях, в которых необходимо эффективно обрабатывать большие объемы данных из запроса, не удерживая их все в памяти одновременно;
- `PipeReader` – эквивалент свойства `HttpContext.BodyReader`. `PipeReader` предоставляет API более высокого уровня по сравнению со `Stream`, но он полезен в аналогичных ситуациях. Подробнее о `PipeReader` и пространстве имен `System.IO.Pipelines` можно узнать на странице <http://mng.bz/XNY6>.

Вы можете получить доступ к каждому из последних хорошо известных типов через внедренный объект `HttpContext`, если хотите. Но внедрение именно того объекта, который вам нужен, обычно упрощает чтение кода.

7.6.2 Внедрение сервисов

В этой книге я несколько раз упоминал, что для работы с ASP.NET Core необходимо настроить различные основные сервисы. Многие сервисы регистрируются автоматически, но часто необходимо добавить дополнительные сервисы, чтобы использовать дополнительные функции, например когда мы вызывали метод `AddHttpLogging()` в главе 3, чтобы добавить журналирование запросов в конвейер.

ПРИМЕЧАНИЕ Добавление сервисов в приложение предполагает их регистрацию в контейнере внедрения зависимостей. Вы узнаете все о внедрении зависимостей и регистрации сервисов в главах 8 и 9.

Можно автоматически использовать любой зарегистрированный сервис в обработчиках конечных точек, а ASP.NET Core внедрит экземпляр сервиса из контейнера внедрения зависимостей. В главе 6 был показан пример использования сервиса `LinkGenerator` в обработчике конечной точки. `LinkGenerator` – один из основных сервисов, зарегистрированных `WebApplicationBuilder`, поэтому он всегда доступен, как показано в следующем листинге.

Листинг 7.7. Использование сервиса `LinkGenerator` в обработчике конечной точки

```
app.MapGet("/links", (LinkGenerator links) => {
    string link = links.GetPathByName("products");
    return $"View the product at {link}";
});
```

←
LinkGenerator мож-
но использовать
в качестве па-
раметра, постольку
он доступен в кон-
тейнере внедрения
зависимостей

Минимальные API могут автоматически определять, когда сервис доступен в контейнере внедрения зависимостей, но если вы хотите указать это явно, вы также можете пометить свои параметры атрибутом [FromServices]:

```
app.MapGet("/links", ([FromServices] LinkGenerator links) =>
```

[FromServices] может потребоваться в некоторых редких случаях, если вы используете собственный контейнер внедрения зависимостей, который не поддерживает API, используемые минимальными API. Но в целом я считаю, что могу обеспечить читабельность конечных точек, избегая атрибутов [From*] везде, где это возможно, и полагаясь на минимальные API для автоматического выполнения правильных действий.

7.6.3 Привязка загрузки файлов с помощью *IFormFile* и *IFormFileCollection*

Общей особенностью многих сайтов является возможность загрузки файлов. Это действие может быть относительно редким, например загрузка пользователем изображения профиля в свой профиль на Stack Overflow, или может быть неотъемлемой частью приложения, например загрузка фотографий в Facebook.

Разрешаем пользователям загружать файлы в наше приложение

Загрузка файлов на сайты – обычное дело, но следует тщательно подумать, нужна ли вашему приложению такая возможность. Всякий раз, когда пользователи могут загружать файлы, ситуация чревата опасностью.

Будьте осторожны и относитесь к входящим файлам как к потенциально вредоносным. Не доверяйте указанному имени файла, будьте осторожны с загружаемыми большими файлами и не позволяйте файлам выполняться на сервере.

Файлы также вызывают вопросы о том, где следует хранить данные: в базе данных, в файловой системе или в каком-то другом хранилище. Ни на один из этих вопросов нет однозначного ответа, и следует хорошенько подумать о последствиях выбора. Лучше не позволяйте пользователям загружать файлы, если в этом нет необходимости!

ASP.NET Core поддерживает отправку файлов, предоставляя интерфейс *IFormFile*. Вы можете использовать этот интерфейс в своих обработчиках конечных точек, и он будет заполняться деталями загрузки файла:

```
app.MapGet("/upload", (IFormFile file) => {});
```

Также можно использовать интерфейс *IFormFileCollection*, если нужно принять несколько файлов:

```
app.MapGet("/upload", (IFormFileCollection files) =>
{
    foreach (IFormFile file in files)
    {
    }
});
```

Объект `IFormFile` предоставляет ряд свойств и служебных методов для чтения содержимого из загруженного файла, некоторые из которых показаны здесь:

```
public interface IFormFile
{
    string ContentType { get; }
    long Length { get; }
    string FileName { get; }
    Stream OpenReadStream();
}
```

Как видите, этот интерфейс предоставляет свойство `FileName`, которое возвращает имя файла, с которым тот был загружен. Но вы ведь знаете, что пользователям нельзя доверять, верно? *Никогда* не следует использовать имя файла непосредственно в коде; пользователи могут применять его для атаки на ваш сайт и доступа к файлам, к которым им не следует иметь доступ. Всегда создавайте новое имя для файла, прежде чем сохранять его где-либо.

ВНИМАНИЕ! Существует множество потенциальных угроз, которые следует учитывать при принятии загрузки файлов от пользователей. Для получения дополнительной информации см. <http://mng.bz/yQ9q>.

Подход с интерфейсом `IFormFile` подойдет, если пользователи собираются загружать только небольшие файлы. Когда ваш метод принимает экземпляр `IFormFile`, все содержимое файла буферизуется в памяти и на диске перед его получением. Затем можно использовать метод `OpenReadStream`, чтобы считать данные.

Если пользователи публикуют на вашем сайте большие файлы, вам может начать не хватать места в памяти или на диске, поскольку ASP.NET Core буферизует каждый файл. В этом случае вам может потребоваться выполнить потоковую передачу файлов напрямую, чтобы избежать одновременного получения всех данных. К сожалению, в отличие от подхода с привязкой модели, потоковая передача больших файлов может быть сложной и подвержена ошибкам, поэтому данная тема выходит за рамки этой книги. Подробности см. в документации Microsoft на странице <http://mng.bz/MBgn>.

СОВЕТ Не используйте интерфейс `IFormFile` для обработки больших загрузок файлов, так как у вас может возникнуть проблема с производительностью. Имейте в виду, вы не можете рассчиты-

вать на то, что пользователи не будут загружать большие файлы, поэтому избегайте загрузки файлов, когда это возможно!

Для подавляющего большинства конечных точек минимальных API конфигурация привязки модели по умолчанию для простых и сложных типов работает отлично. Но вы можете столкнуться с ситуациями, в которых нужно взять на себя немного больше контроля.

7.7 Собственная привязка с помощью BindAsync

Привязка модели, которую вы получаете «из коробки» с минимальными API, охватывает большинство распространенных ситуаций, с которыми вы столкнетесь при создании HTTP API, но всегда есть несколько крайних случаев, в которых вы не сможете ее использовать.

Вы уже видели, что можно внедрить `HttpContext` в обработчики конечных точек, чтобы иметь прямой доступ к деталям запроса в обработчике, но часто вам все равно нужно инкапсулировать логику для извлечения необходимых данных. Вы можете получить лучшее из обоих миров в минимальных API, реализовав `BindAsync` в типах параметров обработчика конечной точки и используя преимущества полностью настраиваемой привязки модели. Чтобы добавить собственную привязку для типа параметра, необходимо реализовать в типе `T` один из двух статических методов `BindAsync`:

```
public static ValueTask<T?> BindAsync(HttpContext context);
public static ValueTask<T?> BindAsync(
    HttpContext context, ParameterInfo parameter);
```

Оба метода принимают `HttpContext`, поэтому вы можете извлечь из запроса все, что вам нужно. Но второй метод также предоставляет детали рефлексии привязываемого параметра. В большинстве случаев более простой сигнатуры должно быть достаточно, но кто знает!

В листинге 7.8 показан пример использования `BindAsync` для привязки записи к телу запроса с использованием собственного формата. Реализация, показанная в листинге, предполагает, что тело содержит два значения типа `double` с разрывом строки, и если это так, то он успешно создает объект `SizeDetails`. Если по пути возникнут какие-либо проблемы, то он возвращает `null`.

Листинг 7.8. Использование BindAsync для привязки собственной модели

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
```

```
app.MapPost("/sizes", (SizeDetails size) => $"Received {size}");
```

```
app.Run();
```

Для параметра `SizeDetails` не требуются дополнительные атрибуты, поскольку он имеет метод `BindAsync`

```

public record SizeDetails(double height, double width)
{
    public static async ValueTask<SizeDetails?> BindAsync(
        HttpContext context)
    {
        using var sr = new StreamReader(context.Request.Body);
        string? line1 = await sr.ReadLineAsync(context.RequestAborted);
        if (line1 is null) { return null; }

        string? line2 = await sr.ReadLineAsync(context.RequestAborted);
        if (line2 is null) { return null; }

        return double.TryParse(line1, out double height)
            && double.TryParse(line2, out double width)
            ? new SizeDetails(height, width)
            : null;
    }
}

```

Создает StreamReader для чтения тела запроса

Считывает строку текста из тела

Пытается разобрать две строки как значения double

Если какая-либо из строк равна null, что указывает на отсутствие содержимого, обработка прекращается

Если разбор прошел успешно, создаем модель SizeDetails и возвращаем ее...

...в противном случае возвращаем null

В листинге 7.8 мы возвращаем значение `null`, если парсинг не удался. Показанная конечная точка приведет к тому, что `EndpointMiddleware` выдаст исключение `BadHttpRequestException` и вернет ошибку 400, поскольку параметр размера в конечной точке является обязательным (не помечен как необязательный). Можно было бы создать исключение в `BindAsync`, но оно не было бы перехвачено `EndpointMiddleware` и привело бы к ответу 500.

7.8 Выбор источника привязки

Уф! Наконец-то мы рассмотрели все способы привязки запроса к параметрам в минимальных API. Во многих случаях все должно работать, как и ожидалось. Простые типы, например `int` и `string`, по умолчанию привязываются к значениям маршрута и значениям строки запроса, а сложные типы привязываются к телу запроса. Но добавление атрибутов `BindAsync` и `TryParse` может привести к путанице!

Когда инфраструктура минимальных API пытается привязать параметр, она проверяет по порядку все следующие источники привязки. Используется первый соответствующий источник привязки:

- 1 Если параметр определяет явный источник привязки с использованием таких атрибутов, как `[FromRoute]`, `[FromQuery]` или `[FromBody]`, параметр привязывается к этой части запроса.
- 2 Если параметр имеет общезвестный тип, например `HttpContext`, `HttpRequest`, `Stream` или `IFormFile`, параметр привязывается к соответствующему значению.
- 3 Если тип параметра имеет метод `BindAsync()`, он используется для привязки.

- 4 Если параметр является строкой или имеет соответствующий метод `TryParse()` (т. е. простой тип):
 - а если имя параметра совпадает с именем параметра маршрута, произойдет привязка к значению маршрута;
 - а в противном случае выполнится привязка к строке запроса.
 - 5 Если параметр представляет собой массив простых типов, `string[]` или `StringValues`, а запрос представляет собой метод `GET` или аналогичный HTTP-метод, который обычно не имеет тела запроса, выполнится привязка к строке запроса.
 - 6 Если параметр представляет собой известный тип сервиса из контейнера внедрения зависимостей, произойдет внедрение сервиса из контейнера.
 - 7 Наконец, выполнится привязка к телу путем десериализации из JSON.
- Инфраструктура минимального API следует этой последовательности для каждого параметра в обработчике и останавливается на первом соответствующем источнике привязки.

ВНИМАНИЕ! Если привязка для записи не удалась и параметр не является необязательным, то запрос завершается неудачей с ответом `400 Bad Request`. Минимальный API не пытается использовать другой источник привязки после сбоя одного источника.

Запоминание этой последовательности источников привязки – одна из самых сложных вещей в минимальных API, которые нужно усвоить. Если вы изо всех сил пытаетесь понять, почему запрос не работает так, как ожидалось, обязательно вернитесь и проверьте эту последовательность. Однажды у меня был параметр, который не был привязан к параметру маршрута, несмотря на то что у него был метод `TryParse`. Когда я проверил последовательность, то понял, что у параметра также есть метод `BindAsync`, который имеет более высокий приоритет!

7.9 Упрощение обработчиков с помощью `AsParameters`

Прежде чем двигаться дальше, мы кратко рассмотрим функцию .NET 7 для минимальных API, которая может упростить обработчики конечных точек: атрибут `[AsParameters]`. Рассмотрим следующую конечную точку `GET`, которая привязывается к значению маршрута, значению заголовка и значениям запроса:

```
app.MapGet("/category/{id}", (int id, int page, [FromHeader(Name = "sort")]
    => bool? sortAsc, [FromQuery(Name = "q")] string search) => { });
```

Думаю, вы согласитесь, что параметры обработчика этого метода довольно сложно читать. Параметры определяют ожидаемую форму запроса, которая далека от идеальной. Атрибут `[AsParameters]` позволяет объединить все эти аргументы в один класс или структуру, упрощая сигнатуру метода и делая все более читабельным.

В листинге 7.9 показан пример преобразования этой конечной точки для использования `[AsParameters]` путем замены ее параметров на `record struct`. Вы также можете использовать класс, запись или структуру и, если хотите, можете использовать свойства вместо параметров конструктора. См. документацию по всем перестановкам, доступную на странице <http://mng.bz/a1KB>.

Листинг 7.9. Использование атрибута `[AsParameters]` для упрощения параметров обработчика конечной точки

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/category/{id}",
    ([AsParameters] SearchModel model) => $"Received {model}"); ←
app.Run();                                     [AsParameters] указывает на то, что
                                                должен быть привязан конструктор
                                                или свойства типа, а не сам тип

record struct SearchModel(
    int id,
    int page,
    [FromHeader(Name = "sort")] bool? sortAsc,
    [FromQuery(Name = "q")] string search);           Каждый параметр при-
                                                вязан так, как если бы
                                                он был записан в обра-
                                                ботчике конечной точки
```

Те же атрибуты и правила применяются для привязки параметров конструктора типа `[AsParameters]` и привязки параметров обработчика конечной точки, поэтому можно использовать атрибуты `[From*]`, внедрять сервисы и общеизвестные типы, а также читать из тела. Этот подход может сделать ваши конечные точки более читабельными, если вы обнаружите, что они становятся громоздкими.

СОВЕТ В главе 16 вы узнаете о привязке модели в MVC и Razor Pages. Вам будет приятно узнать, что в таких случаях подход с использованием атрибута `[AsParameters]` работает «из коробки» без необходимости использования дополнительного атрибута.

На этом раздел о привязке модели подошел к концу. Если все прошло хорошо, то аргументы обработчика конечной точки созданы, и обработчик готов выполнить свою логику. Пришло время обработать запрос, верно? Не о чем беспокоиться.

Не так быстро! Как узнать, что полученные вами данные верны? Как узнать, что вам не прислали вредоносные данные для попытки SQL-инъекции или номер телефона, полный букв? Механизм привязки относительно слепо присваивает значения, отправленные в запросе, который вы с радостью подключите к своим собственным методам. Что мешает маленькому гнусному Джимми отправить вредоносные значения в ваше приложение? За исключением основных мер предосторожности, его ничто не останавливает, поэтому важно *всегда* проверять входные данные. ASP.NET Core предоставляет возможность сделать это декларативным образом «из коробки», чему посвящен раздел 7.10.

7.10 Обработка пользовательского ввода с помощью валидации модели

В этом разделе мы обсудим:

- что такое валидация модели и зачем она нужна;
- использование атрибутов `DataAnnotations` для описания ожидаемых данных;
- как проверить параметры обработчика конечной точки.

Валидация модели в целом – довольно большая тема, и ее необходимо будет учитывать в каждом создаваемом вами приложении. ASP.NET Core позволяет относительно легко добавить валидацию модели в ваши приложения, сделав ее неотъемлемой частью фреймворка.

Минимальные API не включают проверку по умолчанию, вместо этого они предпочтуют предоставлять необязательные точки подключения с помощью фильтров, о которых вы узнали в главе 5. Такая конструкция дает несколько вариантов добавления валидации в приложение; обязательно добавьте ее!

7.10.1 Необходимость валидации модели

Данные могут поступать из множества различных источников в веб-приложении – вы можете загружать их из файлов, читать из базы данных или принимать значения, которые пользователь передает в запросах. Хотя вы, возможно, склонны полагать, что данные, которые находятся на вашем сервере, уже являются допустимыми (но иногда это опасное предположение!), вам *определенно* не следует доверять данным, отправляемым как часть запроса.

СОВЕТ Подробнее о целях валидации, подходах реализации и потенциальных атаках можно прочитать на странице <http://mng.bz/gBxE>.

Следует проверять параметры обработчика конечной точки, прежде чем использовать их для каких-либо действий, касающихся вашего домена, инфраструктуры или всего, что может привести к утечке информации, которая может попасть к злоумышленнику. Обратите внимание, что это предупреждение намеренно расплывчато, поскольку в минимальных API нет определенной точки, где должна происходить проверка. Советую вам сделать это как можно ближе к началу конвейера фильтров минимального API.

Всегда проверяйте данные, предоставленные пользователями, прежде чем использовать их в своих методах. Вы понятия не имеете, что вам мог отправить браузер. Классический пример маленького Бобби Тейблса (<https://xkcd.com/327>) подчеркивает необходимость всегда проверять данные, отправленные пользователем.

Однако валидация используется не только для проверки на предмет наличия угроз безопасности. Также необходимо проверить наличие незлонамеренных ошибок:

- данные должны быть отформатированы правильно. Например, поля адресов электронной почты имеют допустимый формат;
- возможно, что числа должны идти в определенном диапазоне. Нельзя купить –1 копию этой книги!;
- некоторые значения могут быть обязательными, а другие – нет. Для профиля может потребоваться имя, но номер телефона указывать не обязательно;
- значения должны соответствовать вашим бизнес-требованиям. Нельзя конвертировать валюту в эту же валюту, ее нужно конвертировать в другую.

Как упоминалось ранее, фреймворк для создания минимальных API не включает в себя ничего конкретного, что могло бы помочь вам с этими требованиями, но можно использовать фильтры для реализации валидации, как вы увидите в разделе 7.10.3. .NET 7 также включает набор атрибутов, которые можно использовать для значительного упрощения кода валидации.

7.10.2 Использование атрибутов `DataAnnotations` для валидации

Атрибуты валидации, или, точнее, атрибуты `DataAnnotations`, позволяют указать правила, которым должны соответствовать ваши параметры. Они предоставляют метаданные о типе параметра, описывая *тип* данных, которые должна содержать модель привязки, в отличие от самих данных.

Можно применять атрибуты `DataAnnotations` непосредственно к типам параметров, чтобы указать допустимый тип данных. Это позволяет, например, проверить, заполнены ли обязательные поля, находятся ли числа в правильном диапазоне и что в полях электронной почты указаны допустимые адреса.

В качестве примера рассмотрим страницу оформления заказа для приложения конвертера валют. Вам необходимо собрать информацию о пользователе – его имя, адрес электронной почты и (необязательно) номер телефона, – поэтому вы создаете API для сбора этих данных. В следующем листинге показана схема этого API, который принимает параметр `UserModel`. Тип `UserModel` дополнен атрибутами валидации, которые представляют правила проверки для модели.

Листинг 7.10. Добавление `DataAnnotations` в тип для предоставления метаданных

С помощью оператора `using`
добавляется возможность
использования атрибутов валидации

```
using System.ComponentModel.DataAnnotations; <--
```

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
```

```

app.MapPost("/users", (UserModel user) => user.ToString()); <-- API принимает параметр UserModel и привязывает его к телу запроса

app.Run();

public record UserModel
{
    [Required] <-- Значения, отмеченные как Required, обязательно должны быть указаны
    [StringLength(100)] <-- StringLengthAttribute задает максимальную длину свойства
    [Display(Name = "Your name")]
    public string FirstName { get; set; } <-- Настраивает имя, используемое для описания свойства

    [Required]
    [StringLength(100)]
    [Display(Name = "Last name")]
    public string LastName { get; set; }

    [Required] <-- Проверяет, что значение Email может быть действительным адресом электронной почты
    [EmailAddress]
    public string Email { get; set; }

    [Phone] <-- Проверяет, что значение PhoneNumber имеет допустимый формат номера телефона
    [Display(Name = "Phone number")]
    public string PhoneNumber { get; set; }
}

```

Внезапно ваш тип параметра, который раньше был довольно скучным, содержит огромное количество информации. Вы указали, что свойство `FirstName` должно предоставляться всегда, его максимальная длина должна составлять 100 символов и что при его упоминании (например, в сообщениях об ошибках) его следует называть "Your name", а не "FirstName".

Самое замечательное в этих атрибутах то, что они четко декларируют *ожидаемое* состояние экземпляра типа. Глядя на эти атрибуты, вы знаете, что будут или по крайней мере *должны* содержать эти свойства. Затем можно написать код после привязки модели, чтобы подтвердить, что привязанный параметр действителен, как вы увидите в разделе 7.10.3.

У вас есть множество атрибутов на выбор при применении `DataAnnotations` к вашим типам. Я перечислил здесь некоторые распространенные атрибуты, но можно найти и другие в пространстве имен `System.ComponentModel.DataAnnotations`. Для получения более полного списка я рекомендую использовать `IntelliSense` в вашей IDE или обратиться к документации на странице <http://mng.bz/e1Mv>.

- `[CreditCard]` – проверяет, что свойство имеет допустимый формат номера кредитной карты;
- `[EmailAddress]` – проверяет, что свойство имеет допустимый формат адреса электронной почты;
- `[StringLength(max)]` – проверяет, что строка имеет не более `max` символов;
- `[MinLength(min)]` – проверяет, что коллекция имеет как минимум `min` элементов;

- `[Phone]` – проверяет, что свойство имеет допустимый формат номера телефона;
- `[Range(min, max)]` – проверяет, что свойство имеет значение от `min` до `max`;
- `[RegularExpression(regex)]` – проверяет, соответствует ли свойство шаблону регулярного выражения `regex`;
- `[Url]` – проверяет, что свойство имеет допустимый формат URL;
- `[Required]` – указывает, что свойство не должно иметь значение `null`;
- `[Compare]` – позволяет подтвердить, что два свойства имеют одинаковое значение (например, `Email` и `ConfirmEmail`).

ВНИМАНИЕ! Атрибуты `[EmailAddress]` и `[Phone]` подтверждают только то, что *формат* значения является потенциально правильным. Они не подтверждают существование адреса электронной почты или номера телефона. Пример более строгой проверки номера телефона см. в этом посте в блоге Twilio: <http://mng.bz/xmZe>.

Атрибуты `DataAnnotations` не являются новшеством – они были частью .NET Framework начиная с версии 3.5, – и в ASP.NET Core они используются почти так же, как и в предыдущей версии ASP.NET. Их также применяют и для других целей, не только для валидации моделей. Entity Framework Core (среди прочего) использует атрибуты `DataAnnotations` для определения типов столбцов и правил, которые будут применяться при создании таблиц базы данных из классов C#. Подробнее о Entity Framework Core можно узнать в главе 12 и в книге Джона П. Смита «*Entity Framework Core в действии*», 2-е издание (Manning, 2021).

Если атрибуты `DataAnnotation`, предоставляемые из коробки, не охватывают все, что вам нужно, также можно написать специальные атрибуты путем наследования от базового класса `ValidationAttribute`. В главе 32 вы узнаете, как создать специальный атрибут валидации.

Одним из распространенных ограничений атрибутов `DataAnnotation` является сложность валидации свойств, которые зависят от значений других свойств. Возможно, типу `UserModel` из листинга 7.10 требуется, чтобы вы указали либо адрес электронной почты, либо номер телефона, но не то и другое одновременно, чего сложно добиться с помощью атрибутов. В такого рода ситуациях можно реализовать в своих моделях интерфейс `IValidatableObject` вместо использования атрибутов или в дополнение к ним.

В листинге 7.11 к типу `UserModel` добавляется правило валидации, которое проверяет, что указан либо адрес электронной почты, либо номер телефона, но не то и другое одновременно. Если это не так, метод `Validate()` возвращает `ValidationResult`, описывающий проблему.

Листинг 7.11. Реализация интерфейса `IValidatableObject`

```
using System.ComponentModel.DataAnnotations;
public record CreateUserModel : IValidatableObject
```

Реализует интерфейс
`IValidatableObject`

```

{
    [EmailAddress]
    public string Email { get; set; }

    [Phone]
    public string PhoneNumber { get; set; }

    public IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext)
    {
        if(string.IsNullOrEmpty(Email)
            && string.IsNullOrEmpty(PhoneNumber))
        {
            yield return new ValidationResult(
                "You must provide an Email or a PhoneNumber",
                new[] { nameof(Email), nameof(PhoneNumber) });
        }
    }
}

```

Проверяет, является ли объект валидным...

Атрибуты DataAnnotation продолжают проверять основные требования к формату

Validate – единственная функция, реализующая IValidatableObject

...и если нет, возвращает результат, описывающий ошибку

IValidatableObject помогает охватить случаи, с которыми одни атрибуты не могут справиться, но это не всегда лучший вариант. Функция Validate не обеспечивает легкий доступ к сервисам приложения и выполняется только в том случае, если соблюдены все условия атрибутов DataAnnotation.

СОВЕТ DataAnnotations хорошо подходят для изолированной валидации ввода, но не очень полезны при проверке бизнес-правил. Скорее всего, вам потребуется выполнить эту проверку за пределами DataAnnotations.

В качестве альтернативы, если вы не являетесь поклонником подхода, основанного на использовании DataAnnotation и IValidatableObject, то можете использовать популярную библиотеку FluentValidation (<https://github.com/JeremySkinner/FluentValidation>) в своих минимальных API. Минимальные API достаточно гибкие, поэтому вы можете использовать любой подход, который вам больше нравится.

Атрибуты DataAnnotations предоставляют базовые метаданные для валидации, но ни одна часть листинга 7.10 или листинга 7.11 не использует добавленные вами атрибуты проверки. Вам все равно придется добавить код для чтения метаданных типа параметра, проверки достоверности данных и возврата ответа об ошибке, если они не являются допустимыми. ASP.NET Core не включает выделенный API валидации для этой задачи в минимальные API, но его легко добавить с помощью небольшого пакета NuGet.

7.10.3 Добавление фильтра валидации в минимальные API

Microsoft решила не включать какие-либо специальные API валидации в минимальные API. В отличие от Razor Pages и MVC, где валидации – это основная встроенная функция. Аргументация Microsoft

заключалась в том, что компания хотела предоставить пользователям гибкость и выбор для добавления валидации тем способом, который им подходит лучше всего, но не хотела влиять на производительность для тех, кто не хотел использовать их реализацию.

Следовательно, валидация в минимальных API обычно зависит от конвейера фильтров. Как классическая сквозная задача, валидация хорошо подходит для фильтра. Единственным недостатком является тот факт, что обычно нужно писать собственный фильтр, а не использовать существующий API. Положительная сторона заключается в том, что валидация дает вам полную гибкость, включая возможность использовать альтернативную библиотеку (например, FluentValidation), если захотите.

К счастью, у Дамиана Эдвардса, архитектора-менеджера проектов из группы ASP.NET Core из Microsoft, есть пакет NuGet под названием `MinimalApis.Extensions`, предоставляющий необходимый фильтр. Используя простую систему валидации, которая подключается к `DataAnnotations` в моделях, этот пакет NuGet предоставляет метод расширения `WithParameterValidation()`, который вы можете добавить в свои конечные точки. Чтобы добавить пакет, найдите `MinimalApis.Extensions` в диспетчере пакетов NuGet в вашей IDE (обязательно включите предварительные версии) или выполните следующую команду, используя интерфейс командной строки .NET:

```
dotnet add package MinimalApis.Extensions
```

После добавления пакета можно добавить валидацию в любую из ваших конечных точек, добавив фильтр с помощью метода `WithParameterValidation()`, как показано в листинге 7.12. После того как `UserModel` привязывается к телу запроса в формате JSON, фильтр валидации выполняется как часть конвейера фильтров. Если параметр `user` является допустимым, то выполнение передается обработчику конечной точки. В противном случае возвращается ответ `400 Bad Request Issue Details`, содержащий описание ошибок, как показано на рис. 7.8.

Листинг 7.12. Добавление валидации к минимальным API с помощью `MinimalApis.Extensions`

```
using System.ComponentModel.DataAnnotations;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/users", (UserModel user) => user.ToString())
    .WithParameterValidation(); ← Добавляет фильтр валидации
    в конечную точку

app.Run();

public record UserModel ← UserModel определяет требования
{                                         к валидации с помощью атрибутов
  DataAnnotations
}
```

```
[Required]
[StringLength(100)]
[Display(Name = "Your name")]
public string Name { get; set; }
[Required]
[EmailAddress]
public string Email { get; set; }
}
```

В этом примере в теле запроса отправляются недопустимые данные

The screenshot shows a Postman interface with a POST request to `https://localhost:7268/users`. In the 'Body' tab, the 'raw' section contains the JSON `{"Name": "!", "Email": "!"}`. The response status is `400 Bad Request`, and the response body is a JSON object with validation errors:

```

1: {
2:   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
3:   "title": "One or more validation errors occurred.",
4:   "status": 400,
5:   "errors": [
6:     {
7:       "name": [
8:         "The 'Name' field is required."
9:       ],
10:      "Email": [
11:        "The 'Email' field is not a valid e-mail address."
12:      ]
13:    }
14:  ]
15: }

```

Рис. 7.8 Если данные, отправленные в теле запроса, не являются допустимыми, фильтр валидации автоматически возвращает ответ 400 Bad Request, содержащий ошибки проверки, и обработчик конечной точки не выполняется

В листинге 7.12 показано, как проверить сложный тип, но в некоторых случаях может потребоваться проверка простых типов. Возможно, вы захотите проверить, что значение `id` в следующем обработчике должно находиться в диапазоне от 1 до 100:

```
app.MapGet("/user/{id}", (int id) => $"Received {id}")
    .WithParameterValidation();
```

К сожалению, это непросто сделать с атрибутами `DataAnnotations`. Фильтр валидации проверит тип `int`, увидит, что это тип, в свойствах которого нет `DataAnnotations`, и не будет его проверять.

ВНИМАНИЕ! Добавление атрибутов в обработчик, например `([Range(1, 100)] int id)`, не сработает. Атрибуты здесь добавляются к *параметру*, а не к свойствам типа `int`, поэтому валидатор их не найдет.

Есть несколько способов обойти эту проблему, но самый простой – использовать атрибут `[AsParameters]`, который вы видели в разделе 7.9, и применить аннотации к модели. В следующем листинге показано, как это сделать.

Листинг 7.13. Добавление валидации в минимальные API с помощью MinimalApis.Extensions

```
using System.ComponentModel.DataAnnotations;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/user/{id}",
    ([AsParameters] GetUserModel model) => $"Received {model.Id}") ◀
    .WithParameterValidation(); ◀
    | Добавляет фильтр
    | валидации в конеч-
    | ную точку
    | Использует
    | [AsParameters]
    | для создания
    | типа, который
    | можно провер-
    |ить
app.Run();

struct GetUserModel
{
    [Range(1, 10)]
    public int Id { get; set; } | Добавляет атрибуты валидации
} | в ваши простые типы
```

На этом мы завершаем обзор привязки модели в минимальных API. Вы увидели, как фреймворк ASP.NET Core использует привязку модели для упрощения процесса извлечения значений из запроса и превращения их в обычные объекты .NET, с которыми можно быстро работать. От большого количества способов привязки у вас может закружиться голова, но обычно можно придерживаться основ и возвращаться к более сложным типам по мере необходимости.

Хотя это и короткое обсуждение, наиболее важным аспектом этой главы является ее внимание к валидации – общей проблеме для всех веб-приложений. Независимо от того, решите ли вы использовать `DataAnnotations` или другой подход с валидацией, необходимо обязательно проверять все данные, которые вы получаете во всех ваших конечных точках.

В главе 8 мы оставим в стороне минимальные API, чтобы рассмотреть внедрение зависимостей в ASP.NET Core и увидеть, как оно помогает создавать слабосвязанные приложения. Вы узнаете, как зарегистрировать сервисы ASP.NET Core в контейнере, добавить свои собственные сервисы и управлять жизненным циклом сервисов.

Резюме

- Привязка модели – это процесс создания аргументов для обработчиков конечных точек на основе сведений HTTP-запроса. Привязка модели берет на себя извлечение и разбор строк в запросе, поэтому вам не нужно этого делать самим;
- простые значения, например `int`, `string` и `double`, могут быть привязаны к значениям маршрута, значениям строки запроса и заголовкам. Эти значения являются общими, и их легко извлечь из запроса без какого-либо ручного разбора;

- если простое значение не удается привязать, поскольку значение в запросе несовместимо с параметром обработчика, создается исключение `BadHttpRequestException` и возвращается ответ `400 Bad Request`;
- собственный тип можно превратить в простой, добавив метод `TryParse` с сигнатурой `bool TryParse(string value, out T result)`. Если вернуть `false` из этого метода, то минимальные API вернут ответ `400 Bad Request`;
- сложные типы по умолчанию привязываются к телу запроса путем десериализации из JSON;
- минимальные API могут привязываться только к телу в формате JSON; нельзя использовать привязку модели для доступа к значениям формы;
- по умолчанию нельзя привязать тело GET-запросов, поскольку это противоречит ожиданиям для такого рода запросов. Это вызовет исключение во время выполнения;
- массивы простых типов по умолчанию привязываются к значениям строки запроса для запросов методом GET и к телу запроса для запросов методом POST. Эта разница может вызвать путаницу, поэтому всегда думайте, является ли массив лучшим вариантом;
- все параметры обработчика должны быть привязаны правильно. Если параметр попытается привязаться к отсутствующему значению, вы получите исключение `BadHttpRequestException` и ответ `400 Bad Request`;
- в обработчиках конечных точек можно использовать известные типы, например `HttpContext` и любые сервисы из контейнера внедрения зависимостей. Минимальные API проверяют, зарегистрирован ли каждый сложный тип в вашем обработчике как сервис в контейнере DI; в противном случае они рассматривают его как сложный тип для привязки к телу запроса;
- можно читать файлы, отправленные в запросе, с помощью интерфейсов `IFormFile` и `IFormFileCollection` в обработчиках конечных точек. Будьте осторожны, принимая загрузку файлов с помощью этих интерфейсов, поскольку они могут сделать ваше приложение уязвимым для атак со стороны пользователей;
- можно полностью настроить способ привязки типа, используя собственную привязку. Создайте статическую функцию с сигнатурой `public static ValueTask<T?> BindAsync(HttpContext context)` и верните связанное свойство. Этот подход может быть полезен для обработки сложных сценариев, таких как произвольная загрузка в формате JSON;
- можно переопределить источник привязки по умолчанию для параметра, применив атрибуты `[From*]` к параметрам обработчика, например `[FromHeader]`, `[FromQuery]`, `[FromBody]` и `[FromServices]`. Эти параметры имеют приоритет над предположениями, основанными на соглашениях;
- можно инкапсулировать параметры обработчика конечной точки, создав тип, содержащий все параметры в качестве свойств

или аргумента конструктора, и декорировав параметр атрибутом `[AsParameters]`. Данный подход может помочь вам упростить сигнатуру метода конечной точки;

- валидация необходима для проверки на наличие угроз безопасности. Убедитесь, что данные отформатированы правильно, подтвердите, что они соответствуют ожидаемым значениям, и удостоверьтесь, что они соответствуют вашим бизнес-правилам;
- минимальные API не имеют встроенных API валидации, поэтому обычно применяется валидация через фильтр минимального API. Такой подход обеспечивает гибкость, поскольку вы можете реализовать проверку наиболее удобным для вас способом, хотя обычно это означает, что необходимо использовать сторонний пакет;
- пакет NuGet `MinimalApis.Extensions` предоставляет фильтр валидации, который использует атрибуты `DataAnnotations` для декларативного определения ожидаемых значений. Вы можете добавить фильтр с помощью метода расширения `WithParameterValidation()`;
- чтобы добавить собственную валидацию простых типов с помощью `MinimalApis.Extensions`, необходимо создать тип и использовать атрибут `[AsParameters]`.

Часть II

Создание полноценных приложений

В предыдущей части мы рассмотрели много вопросов. Мы увидели, что приложение ASP.NET Core состоит из промежуточного ПО, и сосредоточились на конечных точках минимальных API. Мы увидели, как использовать их для создания JSON API, как извлекать общий код с помощью фильтров и групп маршрутов, а также способы валидации входных данных.

Во второй части мы подробно изучим фреймворк, рассмотрев различные компоненты, которые нам неизбежно понадобятся для создания более сложных приложений. К концу этой части вы сможете создавать динамические приложения, сохраняющие данные в БД, и развертывать их в нескольких окружениях с различными настройками.

ASP.NET Core использует внедрение зависимостей (DI) во всех своих библиотеках, поэтому важно понимать, как работает этот паттерн проектирования. В главе 8 мы познакомимся с внедрением зависимостей и обсудим, для чего оно используется. В главе 9 вы узнаете, как настроить службы в своих приложениях для использования внедрения зависимостей.

В главе 10 рассматривается система конфигурации ASP.NET Core, которая позволяет передавать значения конфигурации в приложение из различных источников: файлов в формате JSON, переменных окружения и многих других. Вы узнаете, как настроить приложение для исполь-

зования разных значений в зависимости от окружения, в котором оно выполняется, и как привязать строго типизированные объекты к конфигурации, чтобы уменьшить количество ошибок во время выполнения.

В главе 11 вы узнаете, как документировать приложения с минимальным API, используя спецификацию OpenAPI. Добавление документа OpenAPI в приложение не только упрощает другим пользователям взаимодействие с ним, но и обладает другими преимуществами. Вы узнаете, как использовать Swagger UI для тестирования своего приложения из браузера и как применять генерацию кода для автоматического создания строго типизированных библиотек для взаимодействия с вашим API.

Большинству веб-приложений требуется какое-то хранилище данных, поэтому в главе 12 я расскажу об Entity Framework Core (EF Core). Эта кроссплатформенная библиотека упрощает подключение приложения к базе данных. EF Core сама по себе заслуживает отдельной книги, поэтому я предоставлю лишь краткое введение и порекомендую превосходную книгу Джона П. Сmita «Entity Framework Core в действии», 2-е изд. (Manning, 2021). Я также покажу вам, как создать базу данных и как вставлять, обновлять и запрашивать простые данные.



Введение во внедрение зависимостей

В этой главе:

- преимущества внедрения зависимостей;
- как ASP.NET Core использует внедрение зависимостей;
- получение сервисов из контейнера внедрения зависимостей.

В первой части этой книги вы познакомились с основами создания приложений с помощью ASP.NET Core. Вы узнали, как комбинировать промежуточное ПО для создания своего приложения и создавать конечные точки минимальных API для обработки HTTP-запросов. Это дало вам инструменты для создания простых приложений.

В данной главе вы увидите, как использовать *внедрение зависимостей* – паттерн проектирования, который помогает разрабатывать слабосвязанный код в приложениях ASP.NET Core. ASP.NET Core широко использует данный паттерн как внутри фреймворка, так и в создаваемых вами приложениях, поэтому вам нужно будет использовать его во всех приложениях, кроме самых тривиальных.

Возможно, вы слышали о внедрении зависимостей раньше и, вероятно, даже использовали его в собственных приложениях. Если это так, тогда данная глава не должна преподнести вам много сюрпризов. Если вы раньше не использовали внедрение зависимостей, не бой-

тесь. Я позабочусь, чтобы вы успели сделать это к тому времени, как глава будет пройдена!

Эта глава начинается со знакомства с внедрением зависимостей, с принципами, которыми оно руководствуется, и с объяснения причин, по которым вам следует использовать его. Вы увидите, как ASP.NET Core использует внедрение зависимостей повсюду в своей реализации и почему вы должны делать то же самое при написании собственных приложений. Наконец, вы узнаете, как получать сервисы из внедрения зависимостей в своем приложении.

Прочитав эту главу, вы получите четкое представление о концепции внедрения зависимостей. В главе 9 вы увидите, как применять внедрение зависимостей к собственным классам. Вы узнаете, как настроить приложение так, чтобы фреймворк ASP.NET Core мог создавать классы за вас, снимая с вас боль создания новых объектов вручную в коде. Вы узнаете, как контролировать продолжительность использования объектов, а также некоторые подводные камни, о которых следует знать при написании собственных приложений. В главе 31 мы рассмотрим некоторые продвинутые способы использования внедрения зависимостей, в том числе способы подключения стороннего контейнера внедрения зависимостей.

Однако сейчас вернемся к основам. Что такое внедрение зависимостей и почему стоит обратить на него внимание?

8.1 Преимущества внедрения зависимостей

Этот раздел призван дать вам общее представление о том, что такое внедрение зависимостей и почему следует знать о нем. Сама тема выходит далеко за рамки одной этой главы. Если вам нужна более подробная информация, настоятельно рекомендую вам ознакомиться со статьями Мартина Фаулера в Интернете. Например, эта статья 2004 г. является классической: <http://mng.bz/pPJ8>.

СОВЕТ Для более глубокого ознакомления с большим количеством примеров на C# рекомендую взять книгу Стивена ван Дойрсена и Марка Земанна «Принципы внедрения зависимостей, практики и паттерны».

ASP.NET Core изначально разрабатывался как модульный фреймворк, который должен придерживаться «хороших» практик разработки ПО. Как и в случае с любым другим программным обеспечением, то, что считается передовой практикой, со временем меняется, но принципы SOLID хорошо себя зарекомендовали в объектно ориентированном программировании.

ОПРЕДЕЛЕНИЕ SOLID – это мнемонический акроним для принципов единственной ответственности, открытости-закрытости, подстановки Лисков, разделения интерфейса и инверсии зависимостей. Этот курс Стива Смита знакомит с принципами использования C#: <http://mng.bz/Ox1R>.

Исходя из этого, в ASP.NET Core *внедрение зависимостей* (которое иногда называют *инверсией зависимости*, или *инверсией управления*) встроено в основу фреймворка. Независимо от того, хотите вы использовать его в коде своего приложения или нет, сами библиотеки фреймворка концептуально зависимы от него.

ПРИМЕЧАНИЕ Несмотря на то что внедрение зависимостей и инверсия зависимостей связаны, это две разные вещи. В этой главе я буду рассматривать оба принципа в общих чертах, но, чтобы лучше понимать различия между ними, см. пост Дерика Бейли под названием «Внедрение зависимостей НЕ то же самое, что принцип инверсии зависимостей»: <http://mng.bz/5jvB>.

Когда вы только начинали программировать, скорее всего, вы не сразу стали использовать внедрение зависимостей. В этом нет ничего удивительного и даже плохого; внедрение зависимостей добавляет определенное количество дополнительных вещей, которые часто не требуются в простых приложениях или когда вы только начинаете. Но когда все становится сложнее, внедрение зависимостей становится отличным инструментом, помогая контролировать эту сложность.

Рассмотрим простой пример, написанный без какого-либо внедрения зависимостей. Представьте, что пользователь зарегистрировался в вашем веб-приложении и вы хотите отправить ему электронное письмо. В этом листинге показано, как подойти к данной задаче на начальном этапе, используя обработчик конечной точки минимального API.

Листинг 8.1. Отправка электронного письма без внедрения зависимостей, когда зависимостей нет

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/register/{username}", RegisterUser); ← Конечная точка вызывается при создании нового пользователя

app.Run(); ← Функция RegisterUser является обработчиком конечной точки

string RegisterUser(string username) ← Создает новый экземпляр EmailSender
{
    var emailSender = new EmailSender(); ← Использует новый экземпляр для отправки электронного письма
    emailSender.SendEmail(username);
    return $"Email sent to {username}!";
}
```

В этом примере обработчик `RegisterUser` выполняется, когда новый пользователь регистрируется в приложении. Так создается новый экземпляр класса `EmailSender` и вызывается метод `SendEmail()` для отправки электронного письма. `EmailSender` – класс, выполняющий отправку электронного письма. Представим, что он выглядит как-то так:

```
public class EmailSender
{
    public void SendEmail(string username)
    {
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

`Console.WriteLine` имитирует реальный процесс отправки электронной почты.

ПРИМЕЧАНИЕ Хотя я использую отправку электронной почты в качестве простого примера, на практике вам может потребоваться полностью убрать этот код из метода обработчика. Данный тип асинхронной задачи хорошо подходит для использования очередей сообщений и фонового процесса. Подробнее см. <http://mng.bz/Y1AB>.

Если класс `EmailSender` такой же простой, как и в предыдущем примере, и у него нет зависимостей, возможно, вы и не увидите необходимости применять другой подход к созданию объектов. И в какой-то степени вы были бы правы. Но что, если позже вы обновите реализацию `EmailSender`, чтобы он сам не реализовал всю логику отправки электронной почты?

На практике для отправки электронной почты этому классу потребуется много действий. Ему пришлось бы:

- создать электронное сообщение;
- сконфигурировать настройки почтового сервера;
- отправить электронное письмо на почтовый сервер.

Если делать все это в одном классе, то это будет противоречить принципу единственной ответственности, поэтому в конечном итоге вы, вероятно, получите класс `EmailSender`, который зависит от других сервисов. На рис. 8.1 показано, как может выглядеть такая схема зависимостей. `RegisterUser` хочет отправить электронное письмо с помощью класса `EmailSender`, но для этого ей также необходимо создать объекты `MessageFactory`, `NetworkClient` и `EmailServerSettings`, от которых зависит класс `EmailSender`.

У каждого класса есть ряд зависимостей, поэтому «корневой» класс, в данном случае `RegisterUser`, должен знать, как создавать каждый класс, от которого он зависит, а также каждый класс, от которого зависят его зависимости. Иногда подобное называют *графом зависимостей*.

ОПРЕДЕЛЕНИЕ *Граф зависимостей* – это набор объектов, которые нужно создать для получения определенного запрашиваемого «корневого» объекта.

Класс `EmailSender` зависит от объектов `MessageFactory` и `NetworkClient`, поэтому они предоставляются через конструктор, как показано здесь.

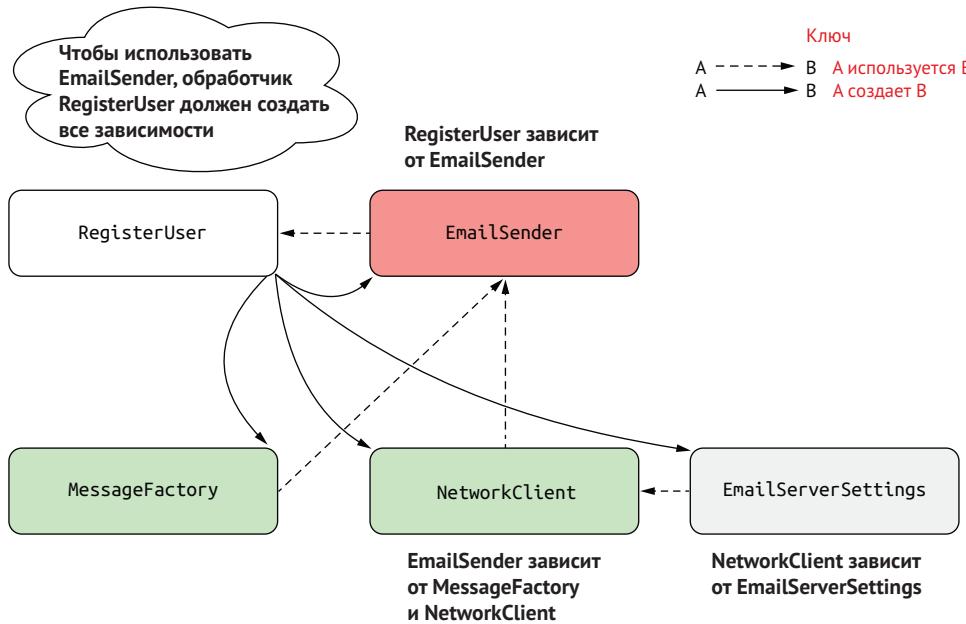


Рис. 8.1 Диаграмма зависимостей без внедрения зависимостей. RegisterUser косвенно зависит от всех остальных классов, поэтому должен создать их все

Листинг 8.2. Сервис с несколькими зависимостями

```
public class EmailSender
{
    private readonly NetworkClient _client;
    private readonly MessageFactory _factory;
    public EmailSender(MessageFactory factory, NetworkClient client)
    {
        _factory = factory;
        _client = client;
    }
    public void SendEmail(string username)
    {
        var email = _factory.Create(username);
        _client.SendEmail(email);
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

EmailSender теперь зависит от двух других классов

Экземпляры зависимостей предоставляются в конструкторе

EmailSender координирует зависимости для создания и отправки электронного письма

Кроме того, класс `NetworkClient`, от которого зависит класс `EmailSender`, также зависит от объекта `EmailServerSettings`:

```
public class NetworkClient
{
    private readonly EmailServerSettings _settings;
    public NetworkClient(EmailServerSettings settings)
```

```
{
    _settings = settings;
}
}
```

Этот пример может показаться немного надуманным, но такая цепочка зависимостей часто встречается. Фактически если в вашем коде этого *нет*, то, вероятно, это признак того, что ваши классы слишком велики и не следуют принципу единственной ответственности.

Итак, как это влияет на код в `RegisterUser?` В следующем листинге показано, как теперь нужно отправить электронное письмо, если вы придерживаетесь использования ключевого слова `new`:

Листинг 8.3. Отправка электронной почты без внедрения зависимостей при создании зависимостей вручную

```
string RegisterUser(string username)
{
    var emailSender = new EmailSender(
        new MessageFactory(),
        new NetworkClient(
            new EmailServerSettings(
                Host: "smtp.server.com",
                Port: 25
            )));
    emailSender.SendEmail(username);
    return $"Email sent to {username}!";
}
```

Все это превращается в какой-то грубый код. Мы улучшили внешний вид класса `EmailSender` для разделения обязанностей, и это превратило его вызов из `RegisterUser` в настоящую рутину. В этом коде есть несколько проблем:

- *неблюдение принципа единственной ответственности* – теперь наш код отвечает и за создание объекта `EmailSender`, и за использование его для отправки электронного письма;
- *слишком много церемоний* – церемония обозначает код, который вам нужно написать, но который напрямую не добавляет ценности. Из одиннадцати строк кода в методе `RegisterUser` только две последние делают что-то полезное. Это затрудняет чтение и усложняет понимание цели метода;
- *привязка к реализации* – если вы решите провести рефакторинг класса `EmailSender` и добавить еще одну зависимость, вам нужно будет обновить все места, где он используется. Точно так же, если какая-либо из зависимостей будет подвергнута рефакторингу, вам также потребуется обновить этот код;
- *трудности с повторным использованием экземпляра* – в примере кода мы создали новые экземпляры всех объектов. Но что, если созда-

ние нового экземпляра `NetworkClient` требует больших вычислительных затрат и мы хотели бы повторно использовать экземпляры? Для решения этой задачи нам пришлось бы добавить дополнительный код, что еще больше увеличило бы объем шаблонного кода.

`RegisterUser` неявно зависит от класса `EmailSender`, поскольку вручную создает сам объект. Единственный способ узнать, что `RegisterUser` использует `EmailSender`, – просмотреть его исходный код. Напротив, у `EmailSender` есть явные зависимости от `NetworkClient` и `MessageFactory`, которые должны быть предоставлены в конструкторе. Точно так же `NetworkClient` имеет явную зависимость от класса `EmailServerSettings`.

СОВЕТ В целом все зависимости в коде должны быть явными. Неявные зависимости сложно рассматривать и тестировать, поэтому следует избегать их везде, где это возможно. Внедрение зависимостей полезно тем, что направляет вас по этому пути.

Внедрение зависимостей направлено на решение проблемы создания графа зависимостей путем инвертирования цепочки зависимостей. Вместо того чтобы `RegisterUser` создавал свои зависимости вручную, глубоко внутри деталей реализации кода, уже созданный экземпляр `EmailSender` передается в качестве аргумента методу `RegisterUser`.

Очевидно, теперь нам нужно что-то для создания объекта, поэтому код для этого должен где-то находиться. Сервис, отвечающий за создание объекта, называется *контейнером внедрения зависимостей*, или *контейнером инверсии управления*, как показано на рис. 8.2.

ОПРЕДЕЛЕНИЕ Контейнер внедрения зависимостей, или контейнер инверсии управления, отвечает за создание экземпляров сервисов. Он знает, как создать экземпляр сервиса, создавая все его зависимости и передавая их конструктору. На протяжении всей книги я буду называть его *контейнер внедрения зависимостей*.

Термин *внедрение зависимостей* часто используется как синоним *инверсии управления*, но внедрение зависимостей – это частный случай более общего принципа инверсии управления. В контексте ASP.NET Core:

- без инверсии управления вам пришлось бы писать код для прослушивания запросов, проверять, какой обработчик использовать, а затем вызывать его. В инверсии управления поток управления происходит наоборот. Вы регистрируете обработчики во фреймворке, но сам фреймворк должен вызывать обработчик. Обработчик по-прежнему отвечает за создание зависимостей;
- внедрение зависимостей продвигает инверсию управления на шаг дальше. Помимо вызова обработчика, с помощью внедрения зависимостей фреймворк создает все зависимости обработчика.

Поэтому когда вы используете внедрение зависимостей, обработчик `RegisterUser` больше не отвечает за контроль над созданием экземпляра `EmailSender`. Вместо этого фреймворк предоставляет экземпляр `EmailSender` непосредственно обработчику.

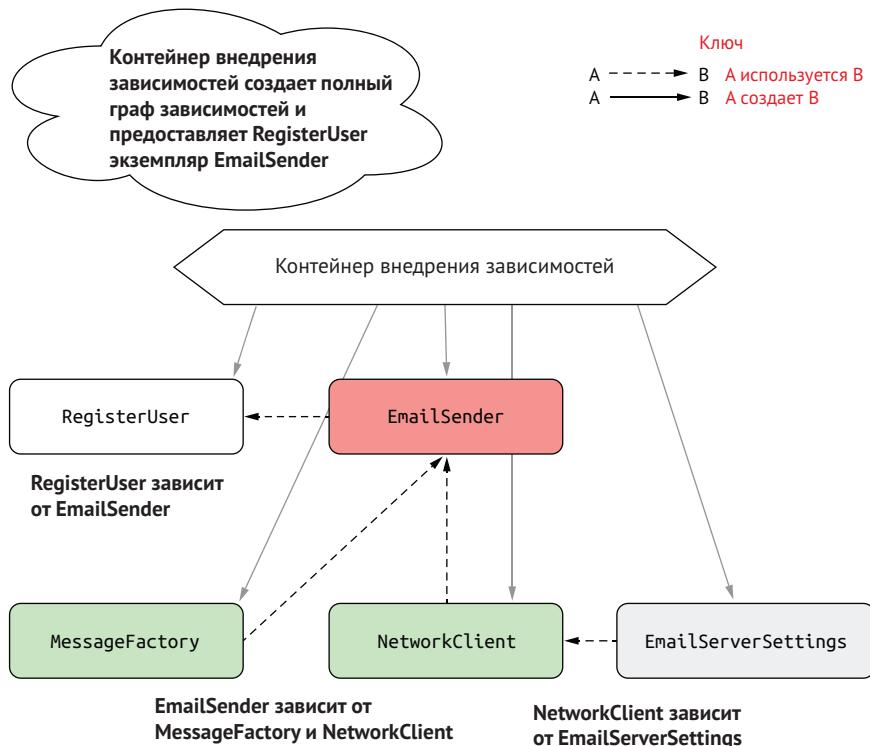


Рис. 8.2 Диаграмма зависимостей с использованием внедрения зависимостей. RegisterUser косвенно зависит от всех других классов, но ему не нужно знать, как создавать их. Он объявляет, что ему требуется EmailSender, и контейнер предоставляет его

ПРИМЕЧАНИЕ Для .NET доступно множество контейнеров внедрения зависимостей: Autofac, Lamar, Unity, Ninject, Simple Injector... Список можно продолжить! В главе 31 вы увидите, как заменить контейнер ASP.NET Core по умолчанию на одну из этих альтернатив.

Преимущество использования данного паттерна становится очевидным, когда вы видите, насколько он упрощает использование зависимостей. В следующем листинге показано, как выглядел бы обработчик RegisterUser, если бы вы использовали внедрение зависимостей для создания EmailSender, вместо того чтобы делать это вручную. Все эти ключевые слова new исчезли, и можно сосредоточиться исключительно на том, что делает контроллер – вызывает EmailSender и возвращает строковое сообщение.

Листинг 8.4. Отправка электронного письма с использованием внедрения зависимостей

```
string RegisterUser(string username, EmailSender emailSender) <--  
{  
    emailSender.SendEmail(username);  
    return $"Email sent to {username}!";  
}  
} Вместо неявного создания зависимостей они внедряются напрямую  
Обработчик снова легко читать и понимать
```

Одним из преимуществ контейнера внедрения зависимостей является тот факт, что он обладает единственной ответственностью – это создание объектов или сервисов. Вы запрашиваете у контейнера экземпляр сервиса, и он заботится о том, чтобы выяснить, как создать график зависимостей, основываясь на том, как вы его настроили.

ПРИМЕЧАНИЕ Когда говорят о контейнерах внедрения зависимостей, часто имеют в виду *сервисы*, что немножко прискорбно, поскольку это один из самых перегруженных терминов в разработке программного обеспечения! В данном контексте под сервисом подразумевается любой класс или интерфейс, который контейнер внедрения зависимостей создает при необходимости.

Прелесть данного подхода состоит в том, что, используя явные зависимости, вам больше не придется писать беспорядочный код, показанный в листинге 8.3. Контейнер внедрения зависимостей может проверить конструктор вашего сервиса и решить, как писать большую часть кода. Эти контейнеры всегда можно сконфигурировать, поэтому если вы хотите описать, как создать экземпляра сервиса вручную, это можно сделать, но по умолчанию в этом нет необходимости.

СОВЕТ ASP.NET Core поддерживает внедрение конструктора и внедрение в методы обработчика конечной точки «из коробки». Технически можно внедрить зависимости в сервис другими способами, например с помощью внедрения свойств, но эти методы не поддерживаются встроенным контейнером внедрения зависимостей.

Я надеюсь, что данный пример продемонстрировал преимущества использования внедрения зависимостей в вашем коде, но во многих отношениях эти преимущества вторичны по сравнению с основным преимуществом использования внедрения зависимостей. В частности, этот принцип помогает сделать так, чтобы ваш код оставался слабосвязанным, путем программирования на уровне интерфейса.

8.2 Создание слабосвязанного кода

Связанность – важная концепция объектно ориентированного программирования. Она означает, как данный класс зависит от других классов при выполнении своей функции. Слабосвязанный код не должен знать много деталей о конкретном компоненте, чтобы его использовать.

Первоначальный пример с RegisterUser и классом EmailSender – образец сильной связанности; мы создавали объект EmailSender напрямую и должны были знать, как именно его подключить. Кроме того, код было сложно протестировать. Любые попытки протестировать RegisterUser приведут к отправке электронного письма. Если бы вы тестирували контроллер с помощью набора модульных тестов, это был бы верный способ внести ваш почтовый сервер в черный список из-за спама!

Использование EmailSender в качестве параметра и снятие ответственности за создание объекта помогает уменьшить связанность в системе. Если реализация EmailSender изменится так, что у него появится еще одна зависимость, вам больше не придется одновременно обновлять RegisterUser.

Остается одна проблема: RegisterUser по-прежнему привязан к *реализации*, а не к *абстракции*. Программирование на уровне абстракции (часто интерфейса) – распространенный паттерн проектирования, который помогает еще больше уменьшить связанность системы, поскольку в этом случае вы не привязаны к одной реализации. Это особенно полезно, если вы делаете классы тестируемыми, поскольку вы можете создавать «фиктивные» реализации зависимостей в целях тестирования, как показано на рис. 8.3.

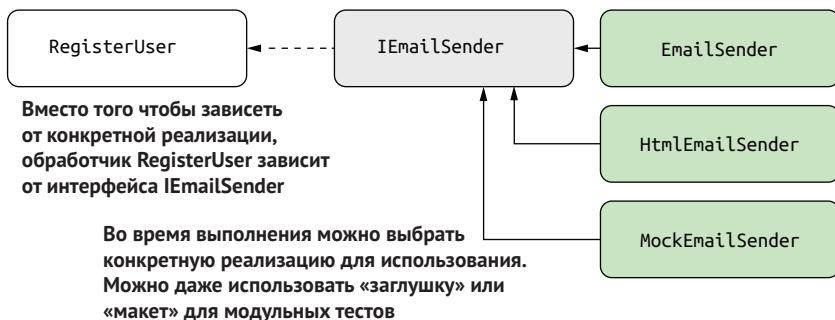


Рис. 8.3 Программируя на уровне интерфейса вместо явной реализации, можно использовать различные реализации IEmailSender в разных сценариях, например MockEmailSender в модульных тестах

СОВЕТ Вы можете выбирать из множества различных фреймворков, используемых для предоставления имитированной реализации зависимостям. Мой любимый фреймворк – Moq, но NSubstitute и FakeItEasy также популярны.

В качестве примера можно создать интерфейс IEmailSender, который будет реализовывать EmailSender:

```
public interface IEmailSender
{
    public void SendEmail(string username);
}
```

Тогда RegisterUser может зависеть от этого интерфейса вместо конкретной реализации EmailSender, как показано в следующем листинге, что позволит вам использовать другую реализацию во время модульных тестов, например DummyEmailSender.

Листинг 8.5. Использование интерфейсов с внедрением зависимостей

```
string RegisterUser(string username, IEmailSender emailSender) ←  
{  
    emailSender.SendEmail(username); ←  
    return $"Email sent to {username}!";  
}  
Теперь вы зависите от IEmailSender,  
а не от конкретной реализации EmailSender
```

Вас не волнует, какова реализация, если она реализует IEmailSender

Ключевым моментом здесь является тот факт, что клиентский код, RegisterUser, не заботится о том, как реализована зависимость, а только о том, что он реализует интерфейс IEmailSender и предоставляет метод SendEmail. Код приложения теперь не зависит от реализации.

Надеюсь, принципы, лежащие в основе внедрения зависимостей, кажутся разумными: имея слабосвязанный код, легко полностью изменить или заменить реализации. Но по-прежнему остается вопрос: как приложение узнает, что в промышленном окружении нужно использовать EmailSender вместо DummyEmailSender? Процесс сообщения контейнеру внедрения зависимостей «когда вам нужен IEmailSender, используйте EmailSender» называется *регистрацией*.

ОПРЕДЕЛЕНИЕ Мы *регистрируем* сервисы в контейнере внедрения зависимостей, чтобы он знал, какую реализацию использовать для каждого запрашиваемого сервиса. Обычно это выглядит так: «для интерфейса X используйте реализацию Y».

То, как именно вы регистрируете свои интерфейсы и типы в контейнере, может варьироваться в зависимости от конкретной реализации контейнера, но принципы, как правило, одинаковы. В ASP.NET Core есть простой контейнер внедрения зависимостей «из коробки». Посмотрим, как он используется при типичном запросе.

8.3 Использование внедрения зависимостей в ASP.NET Core

ASP.NET Core с самого начала разрабатывался как модульный и компонуемый фреймворк, с архитектурой, почти повторяющей стиль плагинов, дополненной внедрением зависимостей. Следовательно, в ASP.NET Core есть простой контейнер внедрения зависимостей, который используют все библиотеки фреймворка для регистрации себя и своих зависимостей.

Этот контейнер используется, например, для регистрации инфраструктуры минимальных API – форматеров, веб-сервера Kestrel и т. д.

Это всего лишь базовый контейнер, поэтому он предоставляет только несколько методов регистрации сервисов, но его также можно заменить на сторонний. Это может дать дополнительные возможности, среди которых авторегистрация или внедрение с помощью свойств. Контейнер внедрения зависимостей встроен в модель размещения ASP.NET Core, как показано на рис. 10.4.

Модель размещения извлекает зависимости из контейнера, когда они необходимы. Если фреймворк определяет, что из-за входящего URL-адреса /route требуется вызвать RegisterHandler, отвечающий за создание минимальных API, RequestDelegateFactory запросит у контейнера реализацию IEmailSender.

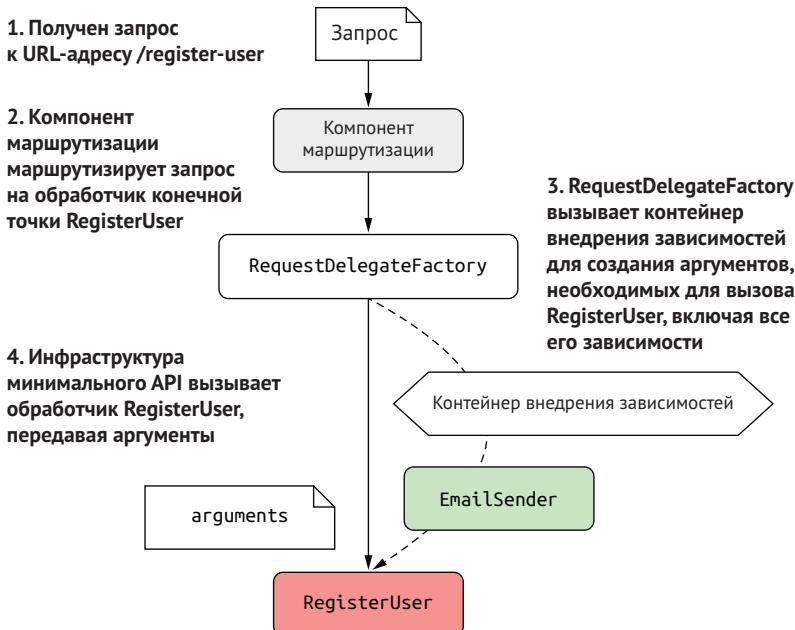


Рис. 8.4 Модель размещения ASP.NET Core использует контейнер внедрения зависимостей, чтобы разрешить зависимости при создании обработчиков конечных точек минимальных API

ПРИМЕЧАНИЕ RequestDelegateFactory – это часть минимального API, которая отвечает за вызов обработчиков минимальных API. Вы не будете использовать его или взаимодействовать с ним напрямую, но он скрытно взаимодействует с контейнером внедрения зависимостей. В моем блоге есть подробная серия, посвященная этому типу: <http://mng.bz/Gy6v>. Но будьте осторожны: этот пост содержит гораздо больше деталей, чем когда-либо понадобится (или захочется знать) большинству разработчиков!

Контейнеру внедрения зависимостей необходимо знать, что создавать при запросе IEmailSender, поэтому необходимо зарегистрировать в контейнере реализацию, например EmailSender. Когда реализация за-

регистрирована, контейнер может внедрить ее куда угодно, а это означает, что вы можете внедрить сервисы, связанные с фреймворком (например, LinkGenerator из главы 6), в свои собственные специальные сервисы. Это также означает, что вы можете зарегистрировать альтернативные версии сервисов, и фреймворк автоматически будет использовать их вместо версий по умолчанию.

Другая инфраструктура ASP.NET Core, например Model-View-Controller (MVC) и Razor Pages (о которых вы узнаете в третьей части), использует внедрение зависимостей аналогично минимальным API. Эти фреймворки применяют контейнер внедрения зависимостей для создания зависимостей, необходимых их собственным обработчикам, например для страницы Razor (рис. 8.5).

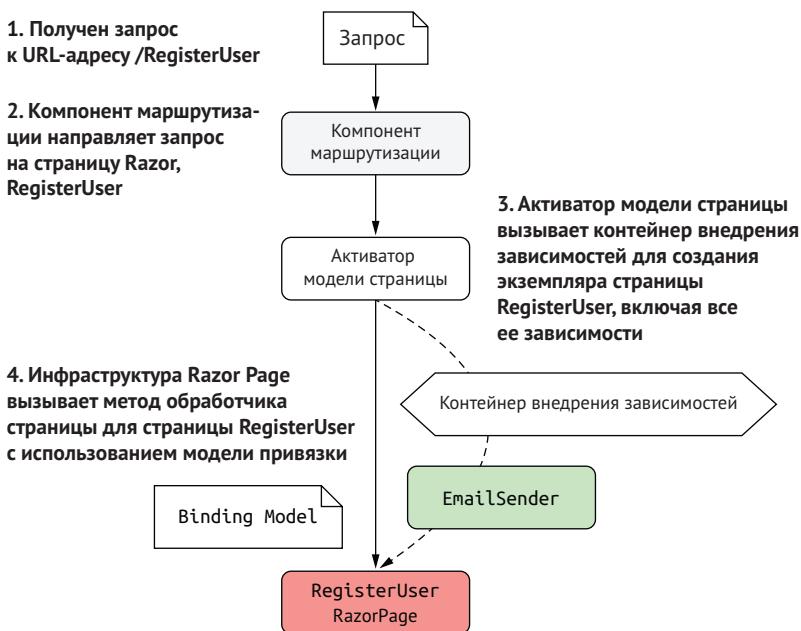


Рис. 8.5 Модель размещения ASP.NET Core использует контейнер внедрения зависимостей, чтобы разрешить зависимости при создании Razor Pages

Гибкость выбора того, как и какие компоненты вы сочетаете в своих приложениях, является одним из преимуществ внедрения зависимостей. В следующем разделе вы узнаете, как сконфигурировать внедрение зависимостей в собственном приложении ASP.NET Core, используя встроенный контейнер.

8.4 Добавление сервисов ASP.NET Core в контейнер

До ASP.NET Core использование внедрения зависимостей было необязательным. Сейчас, напротив, для создания всех приложений ASP.

NET Core, кроме самых тривиальных, требуется некоторая степень внедрения зависимостей. Как я уже упоминал, от этого зависит базовый фреймворк, поэтому такие функции, как Razor Pages и аутентификация, требуют настройки необходимых сервисов. В этом разделе вы увидите, как зарегистрировать эти сервисы во встроенным контейнере. В главе 9 вы узнаете, как зарегистрировать *собственные* сервисы в контейнере внедрения зависимостей.

ASP.NET Core использует внедрение зависимостей для настройки как своих внутренних компонентов, таких как веб-сервер Kestrel, так и дополнительных функций, таких как Razor Pages. Чтобы использовать эти компоненты во время выполнения, контейнеру внедрения зависимостей необходимо знать обо всех классах, которые ему потребуются. Эти сервисы регистрируются с помощью свойства `Services` в экземпляре `WebApplicationBuilder` в файле `Program.cs`.

ПРИМЕЧАНИЕ Свойство `Services` объекта `WebApplicationBuilder` имеет тип `IServiceCollection`. Здесь регистрируется набор сервисов, о которых знает контейнер.

Если вы думаете: «Подождите, мне придется самому настраивать все внутренние компоненты?», – не паникуйте. Большинство основных сервисов регистрируются `WebApplicationBuilder` автоматически, и вам больше ничего делать не нужно. Чтобы использовать другие функции, например Razor Pages или аутентификацию, необходимо явно зарегистрировать компоненты в контейнере своего приложения, но это не так сложно, как кажется. Все общие библиотеки, которые вы используете, предоставляют удобные методы расширения, чтобы по-заботиться о мельчайших деталях. Эти методы настраивают все, что вам нужно, одним махом, вместо того чтобы подключать все вручную.

Например, Razor Pages предоставляет метод расширения `AddRazorPages()`, который добавляет в ваше приложение все необходимые сервисы фреймворка. Вызовите метод расширения свойства `Services` объекта `WebApplicationBuilder` в файле `Program.cs`, как показано в следующем листинге.

Листинг 8.6. Регистрация сервисов Razor Pages в контейнере внедрения зависимостей

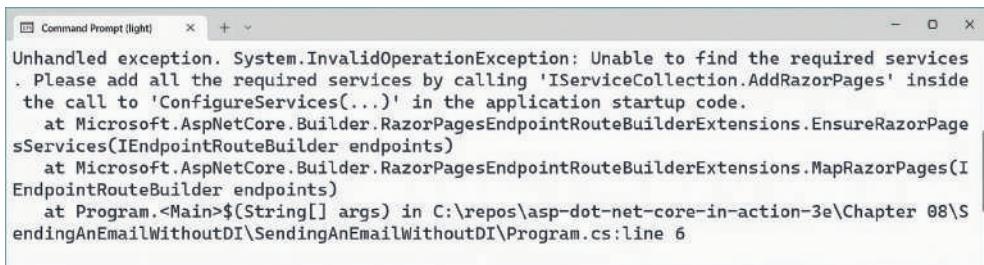
```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages(); ← Метод расширения
                                         AddRazorPages добавляет
                                         все необходимые сервисы
                                         в IServiceCollection
WebApplication app = builder.Build();
app.MapRazorPages(); ← Регистрирует все страницы Razor в вашем
                     приложении в качестве
                     конечных точек
app.Run();
```

Проще некуда. Под капотом этот вызов регистрирует несколько компонентов в контейнере внедрения зависимостей, используя те же самые API. Вы это увидите, когда будете регистрировать собственные сервисы в главе 9.

ПРИМЕЧАНИЕ Не беспокойтесь об аспекте этого кода, связанном с Razor Pages; вы узнаете, как работает Razor Pages, в третьей части. В листинге 8.6 важно показать, как регистрировать и активировать различные функции в ASP.NET Core.

Большинство нетривиальных библиотек, которые вы добавляете в свое приложение, будут иметь сервисы, которые нужно добавить в контейнер. По соглашению каждая библиотека, имеющая необходимые сервисы, должна предоставить метод расширения `Add*`(), который можно вызвать в методе `ConfigureServices`.

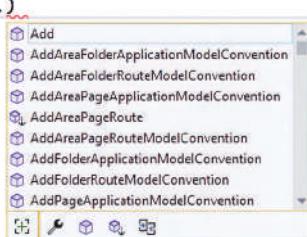
Невозможно точно узнать, какие библиотеки потребуют от вас добавления сервисов в контейнер; обычно для этого нужно обратиться к документации тех библиотек, которые вы используете. Если вы забудете добавить их, то можете обнаружить, что функции не работают, или можете получить исключение, подобное тому, что показано на рис. 8.6. Следите за этим и обязательно регистрируйте все сервисы, которые вам нужны.



```
Unhandled exception. System.InvalidOperationException: Unable to find the required services.
Please add all the required services by calling 'IServiceCollection.AddRazorPages()'
inside the call to 'ConfigureServices(...)' in the application startup code.
   at Microsoft.AspNetCore.Builder.RazorPagesEndpointRouteBuilderExtensions.EnsureRazorPage
sServices(IEndpointRouteBuilder endpoints)
   at Microsoft.AspNetCore.Builder.RazorPagesEndpointRouteBuilderExtensions.MapRazorPages(I
EndpointRouteBuilder endpoints)
   at Program.<Main>$<String[] args> in C:\repos\asp-dot-net-core-in-action-3e\Chapter 08\S
endingAnEmailWithoutDI\SendingAnEmailWithoutDI\Program.cs:line 6
```

Рис. 8.6 Если вам не удастся вызвать метод `AddRazorPages()` в приложении, использующем Razor Pages, вы получите исключение при попытке запуска приложения

Также стоит отметить, что некоторые методы расширения `Add*`() позволяют указывать дополнительные параметры при их вызове, часто с помощью лямбда-выражения. Можно рассматривать это как настройку установки сервиса в приложение. Например, если вы хотите погрузиться в детали, метод `AddRazorPages` предоставляет множество возможностей для тонкой настройки своего поведения, как показано в снippetе IntelliSense на рис. 8.7.



```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options => options.Conventions.)
```

```
var app = builder.Build();

app.MapGet("/", () => "Hello world!");

app.Run();
```

Рис.8.7 Настройка сервисов при добавлении их в коллекцию сервисов. Функция `AddRazorP
ages()` позволяет настроить множество внутренних компонентов фреймворка

Регистрация сервисов в контейнере внедрения зависимостей – это очень хорошо, но важный вопрос заключается в том, как использовать контейнер для получения экземпляра зарегистрированного сервиса. В разделе 8.5 мы рассмотрим два возможных способа доступа к этим сервисам и обсудим, когда вам следует отдать предпочтение тому или иному варианту.

8.5 Использование сервисов из контейнера внедрения зависимостей

В минимальном API у вас есть два основных способа доступа к сервисам из контейнера внедрения зависимостей:

- внедрить сервисы в обработчик конечной точки;
- получить доступ к контейнеру непосредственно в файле Program.cs.

Первый подход – внедрение сервисов в обработчик конечной точки – является наиболее распространенным способом доступа к корню графа зависимостей. Вам следует использовать этот подход почти во всех случаях в ваших минимальных API. Вы можете внедрить сервис в обработчик конечной точки, добавив ее в качестве параметра в свой метод обработчика конечной точки, как показано в главах 6 и 7, когда вводили экземпляр `LinkGenerator` в свой обработчик.

Листинг 8.7. Внедрение сервиса LinkGenerator в обработчик конечной точки

```
app.MapGet("/links", (LinkGenerator links) => {
    string link = links.GetPathByName("products");
    return $"View the product at {link}";
});
```

Контейнер DI создает экземпляр `LinkGenerator` и передает его в качестве аргумента обработчику

Инфраструктура минимального API видит, что вам нужен экземпляр `LinkGenerator`, который представляет собой сервис, зарегистрированный в контейнере, и просит контейнер внедрения зависимостей предоставить экземпляр сервиса. Контейнер создает новый экземпляр `LinkGenerator` (или повторно использует существующий) и возвращает его в инфраструктуру минимального API. Затем `LinkGenerator` передается в качестве аргумента для вызова обработчика конечной точки.

ПРИМЕЧАНИЕ Создает ли контейнер внедрения зависимостей новый экземпляр или повторно использует существующий экземпляр, зависит от жизненного цикла, использованного для регистрации сервиса. Вы узнаете о жизненных циклах в главе 9.

Как уже упоминалось, контейнер внедрения зависимостей создает целый граф зависимостей. Реализация `LinkGenerator`, зарегистрированная в контейнере, объявляет необходимые ей зависимости, используя параметры в своем конструкторе, точно так же как тип `EmailSender` из раздела 8.1 объявлял свои зависимости. Когда контейнер создает `LinkGenerator`, он сначала создает все зависимости сервиса и использует их для создания окончательного экземпляра `LinkGenerator`.

Внедрение сервисов в обработчики – это канонический подход с внедрением зависимостей для обработчиков конечных точек минимальных API, но иногда необходимо получить доступ к сервису вне контекста запроса. У вас может быть множество причин сделать это, но некоторые из наиболее распространенных связаны с работой с базой данных или журналированием. Возможно, вы захотите выполнить некий код, когда ваше приложение, например, начинает обновлять схему базы данных до того, как приложение начнет обрабатывать запросы. Если вам необходимо получить доступ к сервисам в файле `Program.cs` вне контекста запроса, можно получить сервисы из контейнера внедрения зависимостей напрямую, используя свойство `WebApplication.Services`, которое предоставляет контейнер как `IServiceProvider`.

ПРИМЕЧАНИЕ Сервисы регистрируются с помощью `IServiceCollection`, предоставленного в `WebApplicationBuilder.Services`, запрашиваются с помощью `IServiceProvider`, предоставленного в `WebApplication.Services`.

`IServiceProvider` действует как локатор сервисов, поэтому можно напрямую запрашивать сервисы у него, используя методы `GetService()` и `GetRequiredService()`:

- `GetService<T>()` – возвращает запрашиваемый сервис `T`, если он доступен в контейнере внедрения зависимостей; в противном случае возвращает `null`;
- `GetRequiredService<T>()` – возвращает запрашиваемый сервис `T`, если он доступен в контейнере; в противном случае выдается исключение `InvalidOperationException`.

Я обычно отдаю предпочтение `GetRequiredService`, а не `GetService`, поскольку он сразу сообщает, есть ли у вас проблемы с конфигурацией контейнера внедрения зависимостей, выдавая исключение, и вам не нужно обрабатывать значения `null`.

Вы можете использовать любой из этих методов в файле `Program.cs` для получения сервиса. В следующем листинге показано, как получить `LinkGenerator` из контейнера, но здесь можно получить доступ к любому сервису, зарегистрированному в контейнере.

Листинг 8.8. Получение сервиса из контейнера внедрения зависимостей с помощью WebApplication.Services

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/", () => "Hello World!");

LinkGenerator links =
    app.Services.GetRequiredService<LinkGenerator>(); <-- Извлекает сервис из контейнера внедрения зависимостей с помощью метода расширения GetRequiredService<T>()

app.Run(); <-- Вы должны настроить сервисы перед app.Run(), поскольку этот вызов блокируется до тех пор, пока ваше приложение не завершило работу
```

Этот подход, при котором вы напрямую вызываете контейнер внедрения зависимостей для запроса класса, называется паттерн *Локатор сервисов*. Вообще говоря, следует избегать этого паттерна в своем коде; включайте свои зависимости напрямую в качестве аргументов конструктора или обработчика конечной точки и позвольте контейнеру предоставить их вам. Данный паттерн – единственный способ получить доступ к сервисам внедрения зависимостей в основном цикле приложения в файле Program.cs, поэтому не беспокойтесь об их использовании здесь. Тем не менее следует по возможности избегать доступа к WebApplication.Services из обработчиков конечных точек или других типов.

ПРИМЕЧАНИЕ Об антипаттерне «Локатор сервисов» можно прочитать в книге «Принципы, практики и шаблоны внедрения зависимостей» Стивена ван Дёrsена и Марка Зееманна (Manning, 2019).

В этой главе мы рассмотрели причины использования внедрения зависимостей в приложениях, включение дополнительных функций ASP.NET Core путем добавления сервисов в контейнер внедрения зависимостей и получение доступа к сервисам из этого контейнера с помощью внедрения в обработчики конечных точек. В главе 9 вы узнаете о жизненных циклах сервисов и о том, как регистрировать свои собственные сервисы в контейнере.

Резюме

- Внедрение зависимостей встроено во фреймворк ASP.NET Core. Вам необходимо убедиться, что ваше приложение добавляет все необходимые зависимости для дополнительных функций фреймворка в файле Program.cs; в противном случае вы получите исключения во время выполнения, когда контейнер внедрения зависимостей не сможет найти нужные сервисы;
- граф зависимостей – это набор объектов, которые необходимо создать для формирования определенного запрошенного корневого объекта. Контейнер внедрения зависимостей создает все эти зависимости за вас;

- в большинстве случаев следует стремиться использовать явные зависимости вместо неявных. ASP.NET Core использует аргументы конструктора и аргументы обработчика конечной точки для объявления явных зависимостей;
- при обсуждении внедрения зависимостей термин «сервис» используется для описания любого класса или интерфейса, зарегистрированного в контейнере;
- мы регистрируем сервисы в контейнере внедрения зависимостей, чтобы контейнер знал, какую реализацию использовать для каждого запрашиваемого сервиса. Эта регистрация обычно имеет форму «Для интерфейса X используйте реализацию Y»;
- вы должны регистрировать сервисы в контейнере, вызвав методы расширения `Add*` в коллекции `IServiceCollection`, представленной как `WebApplicationBuilder.Services` в файле `Program.cs`. Если вы забудете зарегистрировать сервис, который используется фреймворком или собственным кодом, вы получите исключение `InvalidOperationException` во время выполнения;
- вы можете получить сервисы из контейнера внедрения зависимостей в обработчиках конечных точек, добавив параметр требуемого типа;
- можно получить сервисы из контейнера внедрения зависимостей в файле `Program.cs` с помощью паттерна «Локатор сервисов», вызвав `GetService<T>()` или `GetRequiredService<T>()` в `IServiceProvider`, представленном как `WebApplication.Services`. «Локатор сервисов» обычно считается антипаттерном; как правило, не следует использовать его в методах-обработчиках, но его можно использовать непосредственно в файле `Program.cs`;
- `GetService<T>()` возвращает значение `null`, если запрошенный сервис не зарегистрирован в контейнере внедрения зависимостей.



Регистрация сервисов с помощью внедрения зависимостей

В этой главе:

- конфигурирование сервисов для работы с внедрением зависимостей;
- выбор правильного жизненного цикла сервисов.

В предыдущей главе вы узнали о внедрении зависимостей, о том, почему она полезна в качестве паттерна для разработки слабосвязанного кода, и о его центральном месте в ASP.NET Core. В этой главе мы воспользуемся этими знаниями, чтобы применить внедрение зависимостей к своим собственным классам.

Мы начнем с изучения того, как настроить свое приложение так, чтобы фреймворк ASP.NET Core мог создавать за вас ваши классы, избавляя вас от необходимости создавать новые объекты вручную в коде. Мы рассмотрим различные паттерны, которые вы можете использовать для регистрации своих сервисов, а также некоторые ограничения встроенного контейнера внедрения зависимостей.

Далее вы узнаете, как настроить сервис, имеющий несколько реализаций. Вы узнаете, как внедрить несколько версий сервиса, переопре-

делить регистрацию сервиса по умолчанию и как условно зарегистрировать сервис, если неизвестно, зарегистрирован ли он.

В разделе 9.4 мы рассмотрим, как контролировать продолжительность использования объектов, то есть их жизненный цикл. Мы исследуем различия между тремя вариантами жизненного цикла и некоторые подводные камни, о которых следует знать при написании собственных приложений.

Наконец, в разделе 9.5 вы узнаете, почему жизненный цикл важен при использовании сервисов вне контекста HTTP-запроса.

Мы начнем с повторного рассмотрения сервиса `EmailSender` из главы 8, чтобы увидеть, как зарегистрировать граф зависимостей в контейнере внедрения зависимостей.

9.1 Регистрация собственных сервисов в контейнере внедрения зависимостей

В этом разделе вы узнаете, как зарегистрировать свои собственные сервисы в контейнере внедрения зависимостей. Мы исследуем разницу между сервисом и его реализацией и узнаем, как зарегистрировать иерархию `EmailSender`, представленную в главе 8.

В главе 8 я описал систему отправки писем при регистрации нового пользователя в приложении. Первоначально обработчик конечной точки минимального API `RegisterUser` вручную создал экземпляр `EmailSender`, используя код, аналогичный тому, что приведен в следующем листинге (который вы видели в главе 8).

Листинг 9.1. Создание экземпляра `EmailSender` без внедрения зависимостей

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
app.MapGet("/register/{username}", RegisterUser); ← Конечная точка вызывается при создании нового пользователя
app.Run();

string RegisterUser(string username)
{
    IEmailSender emailSender = new EmailSender( ← Чтобы создать EmailSender, необходимо создать все его зависимости
        new MessageFactory(), ← Вам нужен экземпляр MessageFactory
        new NetworkClient( ← Получилось уже два уровня вложенности, но их вполне возможно, что их может стать больше
            new EmailServerSettings
            (
                Host: "smtp.server.com",
                Port: 25
            )
        );
    emailSender.SendEmail(username); ← Наконец, можно отправить электронное письмо
    return $"Email sent to {username}!";
}
```

NetworkClient также имеет зависимости

Впоследствии мы провели рефакторинг этого кода, чтобы вместо этого внедрить в обработчик экземпляр `IEmailSender`, как показано в листинге 9.2. Интерфейс `IEmailSender` отделяет обработчик конечной точки от реализации `EmailSender`, что упрощает изменение реализации `EmailSender` (или ее замену) без необходимости переписывать `RegisterUser`.

Листинг 9.2. Использование `IEmailSender` с внедрением зависимостей в обработчик конечной точки

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/register/{username}", RegisterUser); ← Конечная точка вызывается при создании нового пользователя

app.Run();

string RegisterUser(string username, IEmailSender emailSender)
{
    emailSender.SendEmail(username); ← Обработчик использует экземпляр IEmailSender
    return $"Email sent to {username}!";
}
```

Последним шагом в проведении рефакторинга является настройка сервисов с помощью контейнера внедрения зависимостей. Эта конфигурация позволяет контейнеру знать, что использовать, когда ему необходимо реализовать зависимость `IEmailSender`. Если вы не зарегистрируете свои сервисы, то во время выполнения вы получите исключение, подобное изображенному на рис. 9.1. Это исключение описывает проблему привязки модели; инфраструктура минимального API пытается привязать параметр `emailSender` к телу запроса, поскольку `IEmailSender` не является известным сервисом в контейнере внедрения зависимостей.

Чтобы полностью настроить приложение, необходимо зарегистрировать реализацию `IEmailSender` и все ее зависимости в контейнере внедрения зависимостей, как показано на рис. 9.2.

Конфигурирование внедрения зависимостей состоит из ряда утверждений о сервисах в вашем приложении. Например:

- если сервису требуется `IEmailSender`, используйте экземпляр `EmailSender`;
- если сервису требуется `NetworkClient`, используйте экземпляр `NetworkClient`;
- если сервису требуется `MessageFactory`, используйте экземпляр `MessageFactory`.

ПРИМЕЧАНИЕ Нам также необходимо будет зарегистрировать в контейнере объект `EmailServerSettings` – в следующем разделе мы сделаем это немного по-другому.

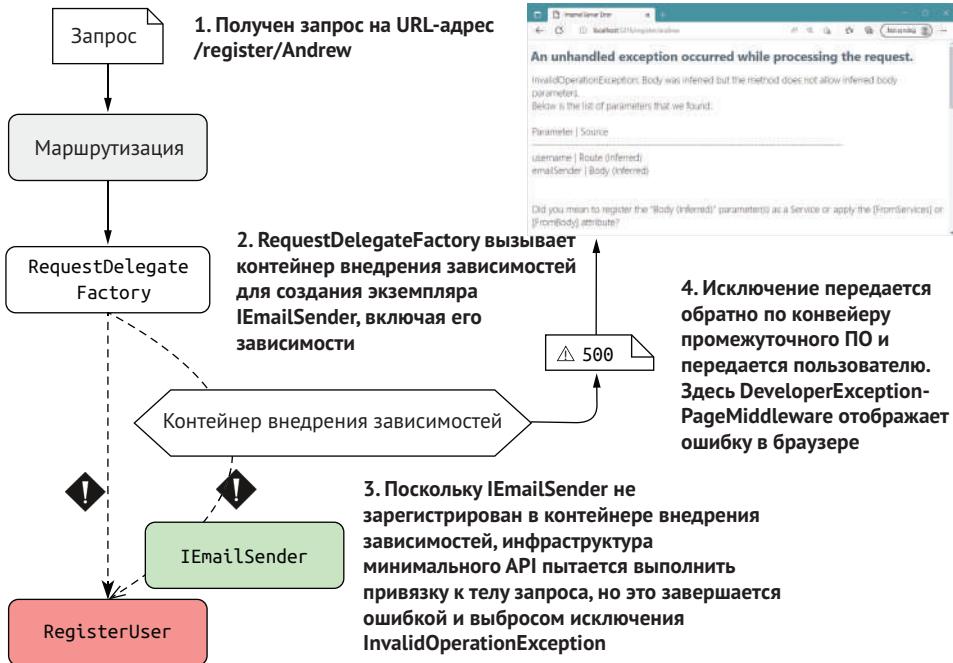


Рис. 9.1 Если вы не зарегистрируете все необходимые зависимости в контейнере внедрения зависимостей, то получите исключение во время выполнения, сообщающее, какой сервис не был зарегистрирован

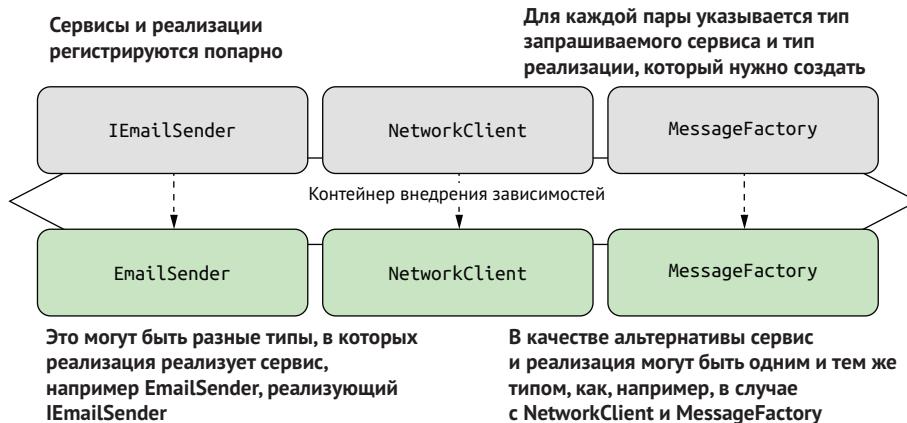


Рис. 9.2 Конфигурирование контейнера внедрения зависимостей в приложении включает в себя сообщение ему, какой тип использовать при запросе данного сервиса, например «Используйте EmailSender, когда требуется IEmailSender»

Данные утверждения выполняются путем вызова различных методов `Add*` в `IServiceCollection` в методе `ConfigureServices`. Каждый метод предоставляет контейнеру три составляющие регистрации:

- *тип сервиса* – `TService`. Это класс или интерфейс, который будет запрошен в качестве зависимости. Часто это интерфейс, например `IEmailSender`, но иногда это может быть и конкретный тип, например `NetworkClient` или `MessageFactory`;
- *тип реализации* – `TService` или `TImplementation`. Это класс, который контейнер должен создать, чтобы получить реализацию зависимости. Это должен быть конкретный тип, например `EmailSender`. Типы реализаций и сервиса могут совпадать, например как у `NetworkClient` и `MessageFactory`;
- *жизненный цикл* – `transient`, `singleton` или `scoped`. Определяет, как долго контейнер внедрения зависимостей должен использовать экземпляр сервиса. Подробнее о жизненном цикле я расскажу в разделе 9.4.

ОПРЕДЕЛЕНИЕ Конкретный тип – это тип, который можно создать, например стандартный класс или структура. Он контрастирует с такими типами, как интерфейс или абстрактный класс, которые невозможно создать напрямую.

В следующем листинге показано, как сконфигурировать `EmailSender` и его зависимости в своем приложении с помощью трех разных методов: `AddScoped<TService>`, `AddSingleton<TService>` и `AddScoped<TService, TImplementation>`. Так вы говорите контейнеру, как создать каждый из экземпляров `TService`, когда они потребуются и какой жизненный цикл использовать.

Листинг 9.3. Регистрация сервисов в контейнере внедрения зависимостей

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IEmailSender, EmailSender>();
builder.Services.AddScoped<NetworkClient>(); ←
builder.Services.AddSingleton<MessageFactory>(); ←

WebApplication app = builder.Build();

app.MapGet("/register/{username}", RegisterUser); ←
app.Run(); ←

string RegisterUser(string username, IEmailSender emailSender)
{
    emailSender.SendEmail(username);
    return $"Email sent to {username}!";
}
```

Всякий раз, когда требуется `IEmailSender`, использовать `EmailSender`

Всякий раз, когда требуется `NetworkClient`, использовать `NetworkClient`

Вот и все, что нужно для внедрения зависимостей! Это может немного походить на волшебство, но вы просто даете контейнеру инструкции о том, как собрать все составные части. Вы даете ему рецепт, как приготовить перец чили, измельчить салат и натереть сыр, чтобы,

когда вы попросите буррито, он мог собрать все ингредиенты воедино и подать вам еду!

ПРИМЕЧАНИЕ Под капотом встроенный контейнер внедрения зависимостей ASP.NET Core использует оптимизированную рефлексию для создания зависимостей, но сторонние контейнеры могут использовать иные подходы. API с методами `Add*` – единственный способ зарегистрировать зависимости во встроенном контейнере; например, здесь нет поддержки использования файлов внешней конфигурации для настройки контейнера.

Тип сервиса и тип реализации одинаковы для `NetworkClient` и `MessageFactory`, поэтому нет необходимости указывать один и тот же тип дважды в методе `AddScoped`. Отсюда и более простая сигнатура.

ПРИМЕЧАНИЕ Экземпляр `EmailSender` регистрируется только как `IEmailSender`, поэтому его нельзя получить, запросив конкретную реализацию `EmailSender`; вы должны использовать интерфейс `IEmailSender`.

Эти обобщенные методы – не единственный способ зарегистрировать сервисы в контейнере. Вы также можете предоставлять объекты напрямую или с помощью лямбда-функций, как вы увидите в следующем разделе.

9.2 Регистрация сервисов с использованием объектов и лямбда-функций

Как я упоминал ранее, я зарегистрировал не все сервисы, необходимые для `EmailSender`. Во всех моих предыдущих примерах `NetworkClient` зависит от `EmailServerSettings`, который вам также потребуется зарегистрировать в контейнере, чтобы ваш проект работал без ошибок.

Я избегал регистрации этого объекта в предыдущем примере, потому что необходимо использовать несколько иной подход. Предыдущие методы `Add*` применяют обобщенные типы для указания типа регистрируемого класса, но не дают никаких указаний на то, *как* создать экземпляр этого типа. Контейнер делает ряд предположений, которых вы должны придерживаться:

- это должен быть класс конкретного типа;
- у класса должен быть только один релевантный конструктор, который может использовать контейнер;
- чтобы конструктор был релевантным, все аргументы конструктора должны быть зарегистрированы в контейнере, или они должны быть аргументами со значением по умолчанию.

ПРИМЕЧАНИЕ Эти ограничения применяются к простому встроенному контейнеру. Если вы решите использовать в своем приложении сторонний контейнер, у него может быть другой набор ограничений.

Запись `EmailServerSettings` не соответствует этим требованиям, так как требует, чтобы вы указали в конструкторе `Host` и `Port`, которые являются строкой и целым числом без значений по умолчанию:

```
public record EmailServerSettings(string Host, int Port);
```

Нельзя регистрировать эти примитивные типы в контейнере; было бы странно, если бы кто-то сказал: «Для каждого аргумента конструктора `string` в любом типе используйте значение».

Вместо этого вы можете самостоятельно создать экземпляр объекта `EmailServerSettings` и передать его контейнеру, как показано ниже. Контейнер использует предварительно сконструированный объект всякий раз, когда требуется экземпляр объекта `EmailServerSettings`.

Листинг 9.4. Предоставление экземпляра объекта при регистрации сервисов

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IEmailSender, EmailSender>();
builder.Services.AddScoped<NetworkClient>();
builder.Services.AddSingleton<MessageFactory>();
builder.Services.AddSingleton(
    new EmailServerSettings
    (
        Host: "smtp.server.com",
        Port: 25
    ));
```

Этот экземпляр `EmailServerSettings` будет использоваться всякий раз, когда требуется экземпляр

```
WebApplication app = builder.Build();

app.MapGet("/register/{username}", RegisterUser);

app.Run();
```

Это отлично работает, если вам нужен только один экземпляр `EmailServerSettings` в приложении – один и тот же объект будет использоваться везде. Но что, если вам нужно создавать *новый* объект каждый раз, когда он запрашивается?

ПРИМЕЧАНИЕ Когда один и тот же объект используется при каждом запросе, он называется *одиночным объектом*, или *синглтоном*. Если вы создаете объект и передаете его в контейнер, он всегда регистрируется как одиночный объект. Вы также можете зарегистрировать любой класс с помощью метода `AddSingleton<T>()`, и контейнер будет использовать только один экземпляр во всем приложении. Подробное обсуждение одиночных объектов наряду с другими видами жизненного цикла приводится в разделе 9.4. Жизненный цикл – это время, в течение которого контейнер должен использовать данный объект, чтобы реализовывать зависимости сервиса.

Вместо предоставления одного экземпляра, который контейнер будет использовать, можно также предоставить функцию, которую контейнер вызывает, когда ему нужен экземпляр типа, как показано на рис. 9.3.

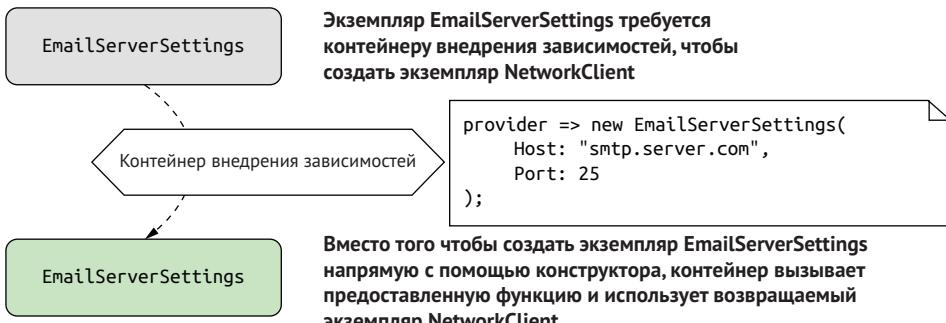


Рис. 9.3 Можно зарегистрировать функцию в контейнере, которая будет вызываться всякий раз, когда потребуется новый экземпляр сервиса

ПРИМЕЧАНИЕ На рис. 9.3 показан пример паттерна «Фабрика», в котором мы определяем, как создается тип. Обратите внимание, что фабричные функции должны быть синхронными; нельзя создавать типы асинхронно, (например) используя `async`.

Самый простой способ зарегистрировать службу с использованием паттерна «Фабрика» – использовать лямбда-функцию (анонимный делегат), в которой контейнер создает новый объект `EmailServerSettings` всякий раз, когда это необходимо, как показано в следующем листинге.

Листинг 9.5. Использование фабричной лямбда-функции для регистрации зависимости

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddScoped<IEmailSender, EmailSender>();
```

```
builder.Services.AddScoped<NetworkClient>();
```

```
builder.Services.AddSingleton<MessageFactory>();
```

```
builder.Services.AddScoped(provider =>
```

Лямбда-функции предоставляет-
ся экземпляр IServiceProvider

```
    new EmailServerSettings
```

```
(
```

```
    Host: "smtp.server.com",
```

```
    Port: 25
```

```
);
```

Поскольку вы предо-
ставляете функцию для
создания объекта, вы не
ограничены синглтоном

Конструктор вызывается
каждый раз, когда требуется
объект `EmailServerSettings`,
а не единожды

```
WebApplication app = builder.Build();
```

```
app.MapGet("/register/{username}", RegisterUser);
```

```
app.Run();
```

В этом примере я изменил жизненный цикл созданного объекта `EmailServerSettings` на `scoped` и предоставил фабричную лямбда-функцию, которая возвращает новый объект `EmailServerSettings`. Каждый раз, когда контейнеру требуется новый объект `EmailServerSettings`, он выполняет функцию и использует новый объект, который она возвращает.

Когда вы используете лямбда-функцию для регистрации своих сервисов, вам предоставляется экземпляр `IServiceProvider` во время выполнения, который в листинге 9.5 называется `provider`. Это экземпляр открытого API самого контейнера внедрения зависимостей, предоставляющий методы расширения `GetService<T>()` и `GetRequiredService<T>()`, которые мы видели в главе 8. Если вам нужно получить зависимости для создания экземпляра вашего сервиса, можно обратиться к контейнеру во время выполнения таким образом, но по возможности этого следует избегать.

СОВЕТ По возможности избегайте вызова функции `GetService()` и `GetRequiredService<T>` в фабричных методах. Лучше отдайте предпочтение внедрению через конструктор – это практично и более эффективно.

Открытые обобщенные типы и внедрение зависимостей

Как уже упоминалось, нельзя использовать обобщенные методы регистрации с `EmailServerSettings`, потому что он использует примитивные типы (в данном случае `int` и `string`) в своем конструкторе. Также нельзя использовать эти методы для регистрации открытых обобщенных типов.

Открытые обобщенные типы – это типы, содержащие параметр обобщенного типа, например `Repository<T>`. Обычно этот тип используется для определения базового поведения, которое можно использовать с несколькими обобщенными типами. Например, в `Repository<T>` можно внедрить `IRepository<Customer>` в свои сервисы, которые должны внедрить, например, экземпляр `DbRepository<Customer>`.

Чтобы зарегистрировать эти типы, необходимо использовать другой перегруженный вариант методов . Например:

```
services.AddScoped(typeof(IRepository<>), typeof(DbRepository<>));
```

Это гарантирует, что всякий раз, когда конструктору сервиса требуется `IRepository<T>`, контейнер внедряет экземпляр `DbRepository<T>`.

На данном этапе все ваши зависимости зарегистрированы. Но класс `Program.cs` выглядит немного хаотично, не так ли? Все полностью зависит от личных предпочтений, но мне нравится группировать свои сервисы в логические группы и создавать для них методы расширения, как в следующем листинге. Здесь создается эквивалент метода расширения `AddControllers()` – приятный и простой API для регистрации. По мере того как вы будете добавлять все больше и больше функциональных возможностей в свое приложение, я думаю, вы тоже его оцените.

Листинг 9.6. Создание метода расширения для упрощения добавления нескольких сервисов

```
public static class EmailSenderServiceCollectionExtensions
{
    public static IServiceCollection AddEmailSender(
        this IServiceCollection services) ← Создаем метод расширения для типа IServiceCollection с помощью ключевого слова «this»
    {
        services.AddScoped<IEmailSender, EmailSender>();
        services.AddSingleton<NetworkClient>();
        services.AddScoped<MessageFactory>();
        services.AddSingleton(
            new EmailServerSettings
            (
                host: "smtp.server.com",
                port: 25
            ));
        return services; ← Вырезаем и вставляем код регистрации из ConfigureServices
    }
}
```

Согласно конвенции «цепочка методов», возвращаем IServiceCollection

После создания предыдущего метода расширения в следующем листинге показано, что наш стартовый код гораздо проще понять!

Листинг 9.7. Использование метода расширения для регистрации сервисов

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddEmailSender(); ← Метод расширения регистрирует все сервисы, связанные с EmailSender
WebApplication app = builder.Build();

app.MapGet("/register/{username}", RegisterUser);

app.Run();
```

До сих пор вы видели, как регистрировать простые внедрения зависимостей, когда существует одна-единственная реализация сервиса. В некоторых ситуациях вы можете обнаружить, что у вас есть несколько реализаций интерфейса. В следующем разделе вы увидите, как зарегистрировать их в контейнере, чтобы они соответствовали вашим требованиям.

9.3 Многократная регистрация сервиса в контейнере

Одним из преимуществ программирования на уровне интерфейса является тот факт, что вы можете создавать несколько реализаций сервиса. Например, представьте, что вы хотите создать более обобщенную версию IEmailSender, чтобы иметь возможность отправлять сообщения через SMS или Facebook, так же, как и по электронной почте. Вы создаете для этого интерфейс:

```
public interface IMessageSender
{
    public void SendMessage(string message);
}
```

а также несколько реализаций: `EmailSender`, `SmsSender` и `FacebookSender`. Но как зарегистрировать эти реализации в контейнере? И как внедрить их в обработчик `RegisterUser`? Ответ может отличаться в зависимости от того, хотите ли вы использовать все реализации в своем потребителе или только одну из них.

9.3.1 Внедрение нескольких реализаций интерфейса

Представьте, что вы хотите отправить сообщение, используя каждую из реализаций `IMessageSender` всякий раз, когда регистрируется новый пользователь, так чтобы он получал электронное письмо, SMS и сообщение в Facebook, как показано на рис. 9.4.



Рис. 9.4 Когда пользователь регистрируется в вашем приложении, он вызывает обработчик `RegisterUser`, который отправляет ему электронное письмо, SMS и сообщение в Facebook с помощью классов `IMessageSender`

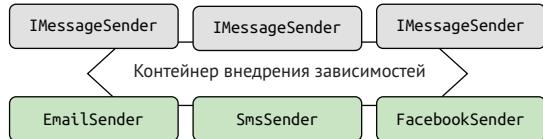
Самый простой способ сделать это – зарегистрировать все реализации сервисов в своем контейнере внедрения зависимостей, чтобы он внедрял по одному экземпляру каждого типа в обработчик конечной точки `RegisterUser`. Затем `RegisterUser` может использовать простой цикл `foreach` для вызова метода `SendMessage()` для каждой реализации, как показано на рис. 9.5.

Мы регистрируем несколько реализаций одного и того же сервиса в контейнере внедрения зависимостей, точно так же, как и для отдельных реализаций, используя методы расширения `Add*`. Например:

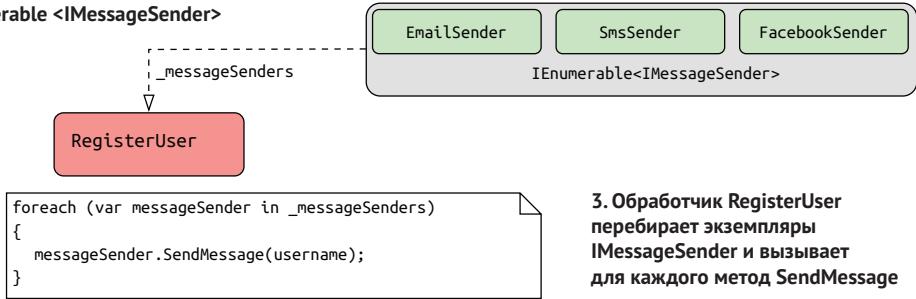
```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddScoped<IMessageSender, EmailSender>();
builder.Services.AddScoped<IMessageSender, SmsSender>();
builder.Services.AddScoped<IMessageSender, FacebookSender>();
```

Затем можно внедрить `IEnumerable<IMessageSender>` в `RegisterUser`, как показано в следующем листинге. Контейнер внедряет массив `IMessageSender`, содержащий по одной из зарегистрированных нами реализаций в том же порядке, в каком мы их зарегистрировали. Затем можно использовать стандартный цикл `foreach` для вызова метода `SendMessage()` для каждой реализации.

1. Несколько реализаций `IMessageSender` регистрируются в контейнере DI с использованием обычных методов `Add*`



2. Контейнер создает по одному экземпляру каждой реализации `IMessageSender` и внедряет их в `RegisterUser` как `IEnumerable<IMessageSender>`



3. Обработчик `RegisterUser` перебирает экземпляры `IMessageSender` и вызывает для каждого метод `SendMessage`

Рис. 9.5 Можно зарегистрировать несколько реализаций сервиса в контейнере, как, например, `IEmailSender` в этом примере. Вы можете получить экземпляр каждой из этих реализаций, затребовав `IEnumerable<IMessageSender>` в обработчике `RegisterUser`

Листинг 9.8. Внедрение нескольких реализаций сервиса в конечную точку

```

string RegisterUser(
    string username,
    IEnumerable<IMessageSender> senders)
{
    foreach(var sender in senders)
    {
        Sender.SendMessage($"Hello {username}!");
    }

    return $"Welcome message sent to {username}";
}

```

При запросе `IEnumerable` будет внедрен массив `IMessageSender`

Каждый `IMessageSender` в `IEnumerable` – это отдельная реализация

ВНИМАНИЕ! Вы должны использовать `IEnumerable<T>` в качестве аргумента конструктора для внедрения всех зарегистрированных типов сервиса, `T`. Несмотря на то что внедрение будет сделано путем создания массива `T[]`, нельзя использовать `T[]` или `ICollection<T>` в качестве аргумента конструктора. Это вызовет исключение `InvalidOperationException`, подобное тому, что вы видели на рис. 9.1.

Достаточно просто внедрить все зарегистрированные реализации сервиса, но что, если вам нужна только одна? Как контейнер узнает, какую из них использовать?

9.3.2 Внедрение одной реализации при регистрации нескольких сервисов

Представьте, что вы уже зарегистрировали все реализации `IMessageSender`; что происходит, если у вас есть сервис, которому требуется только одна из них? Например:

```
public class SingleMessageSender
{
    private readonly IMessageSender _messageSender;
    public SingleMessageSender(IMessageSender messageSender)
    {
        _messageSender = messageSender;
    }
}
```

Контейнер должен выбрать одну реализацию `IMessageSender` для внедрения в этот сервис, из трех доступных. Для этого используется последняя зарегистрированная реализация – `FacebookSender` из предыдущего примера.

ПРИМЕЧАНИЕ Контейнер внедрения зависимостей будет использовать последнюю зарегистрированную реализацию сервиса при разрешении одного экземпляра сервиса.

Это может быть особенно полезно для замены встроенных регистраций собственными сервисами. Если у вас есть собственная реализация сервиса, которая, как вы знаете, зарегистрирована в библиотечном методе расширения `Add*`, можно переопределить эту регистрацию, зарегистрировав собственную. Контейнер внедрения зависимостей будет использовать вашу реализацию всякий раз, когда запрашивается единственный экземпляр сервиса.

Основным недостатком данного подхода является тот факт, что вы по-прежнему получаете несколько зарегистрированных реализаций – вы можете внедрить `IEnumerable<T>`, как и раньше. Иногда нужно условно зарегистрировать сервис, чтобы у вас была только одна зарегистрированная реализация.

9.3.3 Условная регистрация сервисов с помощью `TryAdd`

Иногда нужно добавить реализацию сервиса, только если она еще не была добавлена. Это особенно полезно для авторов библиотек; они могут создать реализацию интерфейса по умолчанию и зарегистрировать ее только в том случае, если пользователь еще не зарегистрировал собственную реализацию.

В пространстве имён `Microsoft.Extensions.DependencyInjection.Extensions` можно найти несколько методов расширения для условной регистрации, например `TryAddScoped`. Он проверяет, не зарегистрирован ли сервис в контейнере, перед тем как вызвать метод `AddScoped` для реализации. В следующем листинге показано, как условно добавить `SmsSender`, только если нет существующих реализаций `IMessageSender`. Поскольку вы ранее зарегистрировали `EmailSender`, контейнер будет игнорировать регистрацию `SmsSender`, поэтому она не будет доступна в вашем приложении.

Листинг 9.9. Условное добавление сервиса с помощью метода TryAddScoped

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddScoped<IMessageSender, EmailSender>();
builder.Services.TryAddScoped<IMessageSender, SmsSender>();
```

EmailSender
зарегистри-
рован в кон-
тейнере

Уже существует реализация IMessageSender,
поэтому SmsSender не зарегистрирован

Такой код часто не имеет большого смысла на уровне приложения, но может быть полезен, если вы создаете библиотеки для использования в нескольких приложениях. Например, ASP.NET Core использует TryAdd* во многих местах, что позволяет легко регистрировать альтернативные реализации внутренних компонентов в собственном приложении, если вы этого хотите.

Вы также можете заменить ранее зарегистрированную реализацию с помощью метода расширения Replace(). К сожалению, API этого метода не так удобен, как методы TryAdd. Чтобы заменить ранее зарегистрированный IMessageSender на SmsSender, нужно использовать

```
builder.Services.Replace(new ServiceDescriptor(
    typeof(IMessageSender), typeof(SmsSender), ServiceLifetime.Scoped
));
```

СОВЕТ При использовании метода Replace() вы должны указать тот же жизненный цикл, который использовался для регистрации заменяемого сервиса.

Мы довольно подробно рассмотрели регистрацию зависимостей, но лишь в общих чертах коснулись одного важного аспекта: жизненного цикла. Понимание жизненного цикла имеет решающее значение при работе с контейнерами внедрения зависимостей, поэтому важно уделять им пристальное внимание при регистрации сервисов в контейнере.

9.4 Жизненный цикл: когда создаются сервисы?

Каждый раз, когда в контейнере запрашивается конкретный зарегистрированный сервис, например экземпляр IMessageSender, он может сделать одно из двух:

- создать и вернуть новый экземпляр сервиса;
- вернуть существующий экземпляр сервиса.

Жизненный цикл сервиса управляет поведением контейнера по отношению к этим параметрам. Он определяется во время регистрации сервиса. Жизненный цикл определяет, когда контейнер внедрения зависимостей повторно использует существующий экземпляр сервиса для выполнения зависимостей сервиса, а когда создает новый.

ОПРЕДЕЛЕНИЕ Жизненный цикл сервиса – это продолжительность существования экземпляра сервиса в контейнере, прежде чем он создаст новый экземпляр.

Важно понять последствия использования разных жизненных циклов, применяемых в ASP.NET Core, поэтому в этом разделе рассматривается каждый доступный вариант цикла и то, когда необходимо его использовать. В частности, вы увидите, как жизненный цикл влияет на то, как часто контейнер внедрения зависимости создает новые объекты. В разделе 9.4.4 я покажу вам паттерн, на который следует обратить внимание, когда зависимость с коротким жизненным циклом «захватывается» зависимостью с длительным циклом. Данный антипаттерн может вызвать некоторые трудности с отладкой, поэтому важно помнить об этом при конфигурировании приложения.

В ASP.NET Core можно указать три разных типа жизненного цикла при регистрации сервиса во встроенным контейнере:

- **Transient** – каждый раз, когда запрашивается сервис, создается новый экземпляр. Это означает, что потенциально у вас могут быть разные экземпляры одного и того же класса в одном и том же графе зависимостей;
- **Scoped** – в пределах области применения все запросы сервиса будут предоставлять вам один и тот же объект. Для разных областей вы получите разные объекты. В ASP.NET Core каждый веб-запрос получает свою область применения;
- **Singleton** – вы всегда будете получать один и тот же экземпляр сервиса, независимо от области видимости.

ПРИМЕЧАНИЕ Эти концепции хорошо согласуются с концепциями в большинстве других контейнеров, но терминология часто отличается. Если вы имеете дело со сторонним контейнером, убедитесь, что вы понимаете, как концепции жизненного цикла согласуются со встроенным контейнером ASP.NET Core.

Чтобы проиллюстрировать поведение каждого цикла, я воспользуюсь простым примером. Представьте, что у вас есть класс `DataContext`, у которого есть подключение к базе данных, как показано в листинге 9.10. У него есть одно-единственное свойство `RowCount`, отображающее число строк в таблице базы данных `Users`. Для этого примера мы эмулируем вызов базы данных, задав количество строк в конструкторе, поэтому вы будете получать одно и то же значение каждый раз, когда будете вызывать свойство `RowCount` для данного экземпляра `DataContext`. Разные экземпляры будут возвращать разное значение `RowCount`.

Листинг 9.10. `DataContext` генерирует случайный `RowCount` при создании

```
class DataContext
{
    public int RowCount { get; } ←
        = Random.Shared.Next(1, 1_000_000_000);
}
```

Это свойство только для чтения, поэтому оно всегда возвращает одно и то же значение

Генерирует случайное число от 1 до 1 000 000 000

У вас также есть класс `Repository`, который зависит от класса `DataContext`, как показано в следующем листинге. Он также предоставляет свойство `RowCount`, но делегирует вызов экземпляру `DataContext`. Независимо от значения, с которым был создан экземпляр `DataContext`, `Repository` отображает то же значение.

Листинг 9.11. Класс Repository, зависящий от экземпляра класса `DataContext`

```
public class Repository
{
    private readonly DataContext _dataContext;
    public Repository(DataContext dataContext)
    {
        _dataContext = dataContext;
    }
    public int RowCount => _dataContext.RowCount;
```

Экземпляр `DataContext` предоставляется с использованием внедрения зависимостей

RowCount возвращает то же значение, что и текущий экземпляр `DataContext`

}

Наконец, у вас есть обработчик конечной точки `RowCounts`, который напрямую зависит от обоих этих классов. Когда инфраструктура минимального API создает аргументы, необходимые для вызова `RowCounts`, контейнер внедряет экземпляры классов `DataContext` и `Repository`. Чтобы создать экземпляр `Repository`, он также создает второй экземпляр `DataContext`. В ходе двух запросов в общей сложности потребуетсяся четыре экземпляра класса `DataContext`, как показано на рис. 9.6.

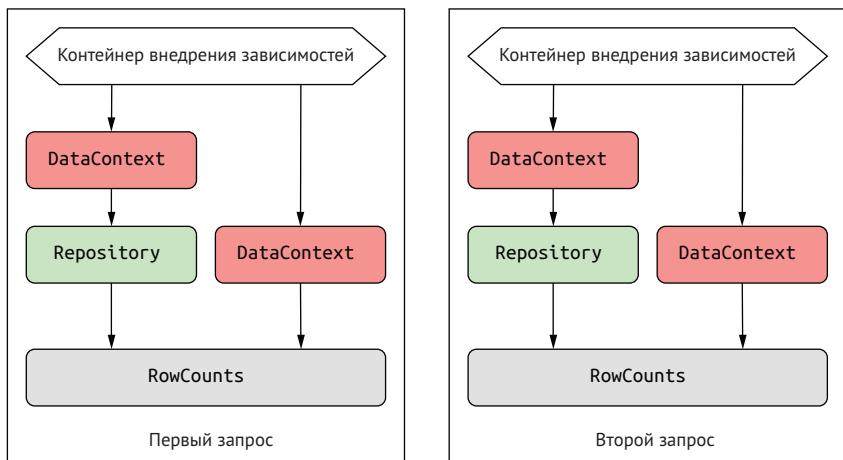


Рис. 9.6 Контейнер внедрения зависимостей использует два экземпляра класса `DataContext` для каждого запроса. В зависимости от жизненного цикла, в котором зарегистрирован тип `DataContext`, контейнер может создать один, два или четыре разных экземпляра `DataContext`

Обработчик RowCounts извлекает значение RowCount, возвращенное как из Repository, так и из DataContext, а затем возвращает его в виде строки, аналогично коду в листинге 9.12. Пример кода, связанный с этой книгой, также записывает и отображает значения предыдущих запросов, поэтому можно легко отслеживать, как значения изменяются с каждым запросом.

Листинг 9.12. Обработчик RowCounts зависит от классов DataContext и Repository

```
static string RowCounts(DataContext и Repository создаются с использованием внедрения зависимостей
    DataContext db,
    Repository repository)
{
    int dbCount = db.RowCount;
    int repositoryCount = repository.RowCount;
    return: $"DataContext: {dbCount}, Repository: {repositoryCount}"; При вызове обработчик страницы извлекает и записывает RowCount из обеих зависимостей
}
```

Счетчики возвращаются в ответе

Цель этого примера – изучить взаимосвязь между четырьмя экземплярами класса DataContext в зависимости от жизненных циклов, которые вы используете для регистрации сервисов в контейнере. Я генерирую случайное число в DataContext как способ однозначной идентификации экземпляра DataContext, но вы можете рассматривать это как текущее количество пользователей, вошедших на ваш сайт, или, например, количество товара на складе на определенный момент времени.

Я начну с самого короткого типа жизненного цикла, transient, перейду к распространенному типу scoped, а затем мы рассмотрим синглтоны. Наконец, я покажу серьезную ловушку, на которую следует обратить внимание при регистрации сервисов в своих приложениях.

9.4.1 Transient: уникален каждый

В контейнере ASP.NET Core кратковременные (transient) сервисы всегда создаются заново, когда они необходимы, чтобы реализовать зависимости. Вы можете зарегистрировать свои сервисы, используя методы расширения AddTransient:

```
builder.Services.AddTransient<DataContext>();
builder.Services.AddTransient<Repository>();
```

При такой регистрации каждый раз, когда требуется зависимость, контейнер будет создавать новый экземпляр. Такое поведение контейнера для кратковременных сервисов применимо как между запросами, так и внутри запросов; экземпляр класса DataContext, внедренный в Repository, будет отличаться от того, что внедрен в обработчик RowCounts.

ПРИМЕЧАНИЕ Кратковременные зависимости могут приводить к появлению разных экземпляров одного и того же типа внутри одного графа зависимостей.

На рис. 9.7 показаны результаты, которые вы получаете от многократного вызова API, когда используете кратковременный жизненный цикл для обоих сервисов. Видно, что каждое значение отличается как внутри запроса, так и между запросами. Обратите внимание, что рис. 9.7 был создан с использованием исходного кода для этой главы, который основан на листингах данной главы, но также отображает результаты предыдущих запросов, чтобы облегчить наблюдение за поведением.

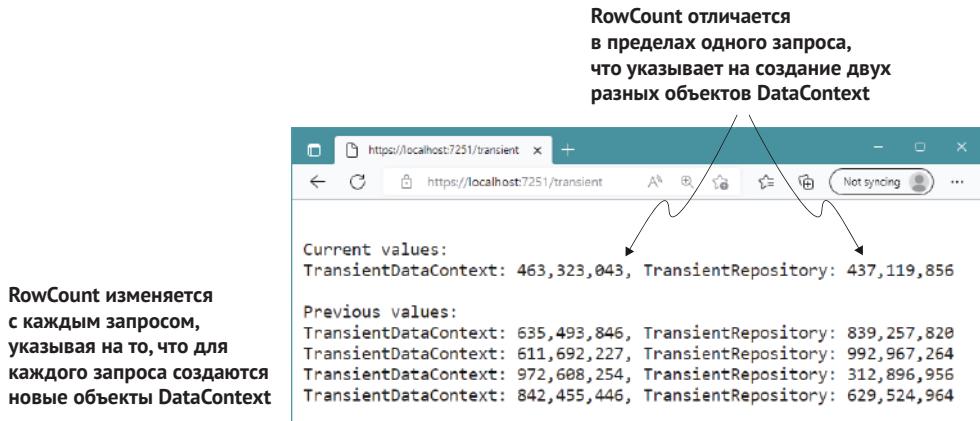


Рис. 9.7 При регистрации с использованием кратковременного жизненного цикла все четыре объекта DataContext различаются. Это видно, потому что все значения внутри и между запросами разные

Кратковременные жизненные циклы могут привести к созданию большого количества объектов, поэтому они имеют смысл для легковесных сервисов без состояния. Это равносильно вызову `new` каждый раз, когда вам понадобится новый объект, поэтому имейте это в виду. Вы вряд ли будете слишком часто применять данный подход; большинство ваших сервисов, вероятно, будут использовать жизненный цикл типа `scoped`.

9.4.2 **Scoped: держимся вместе**

Жизненный цикл типа `scoped` указывает на то, что один-единственный экземпляр объекта будет использоваться в пределах заданной области, но разные экземпляры будут использоваться между разными областями. В ASP.NET Core область применения соответствует запросу, поэтому в рамках одного запроса контейнер будет использовать один и тот же объект, чтобы удовлетворять всем зависимостям.

В предыдущем примере это означает, что в рамках одного запроса (одна область применения) во всем графе зависимостей будет использоваться один и тот же экземпляр `DataContext`, внедренный в `Repository`, будет тем же экземпляром, что и внедренный в обработчик `RowCountModel`.

В следующем запросе вы будете находиться в другой области, поэтому контейнер создаст новый экземпляр `DataContext`, как показано на рис. 9.8. Как видно, другой экземпляр означает другой `RowCount` для каждого запроса. Как и раньше, на рис. 9.8 также показано количество предыдущих запросов.

RowCount изменяется с каждым запросом, указывая на то, что для каждого запроса создается новый объект **DataContext**

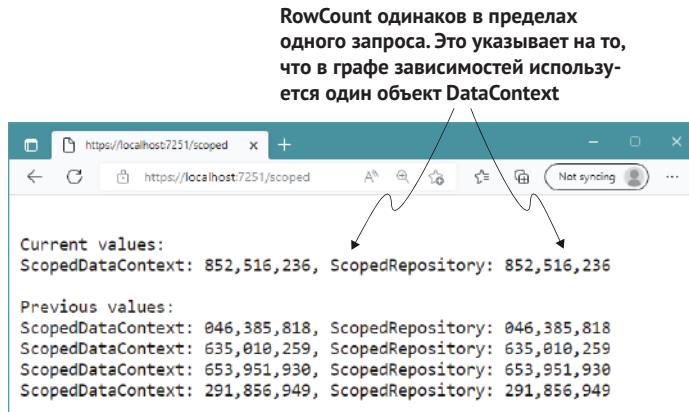


Рис. 9.8 Зависимости, зарегистрированные как scoped, используют один и тот же экземпляр **DataContext** в рамках одного запроса. Поэтому значения **RowCount** одинаковы в рамках запроса

Вы можете зарегистрировать зависимости как scoped с помощью методов расширения `AddScoped`. В этом примере я зарегистрировал `DataContext` как scoped и оставил `Repository` как transient, но вы бы в этом случае получили те же результаты, если бы оба они были scoped:

```
builder.Services.AddScoped<DataContext>();
```

Из-за природы веб-запросов часто можно встретить сервисы, регистрируемые как scoped-зависимости в ASP.NET Core. Контексты базы данных и сервисы аутентификации – распространенные примеры сервисов, которые должны быть ограничены запросом, – все, что вы хотите совместно использовать в своих сервисах в рамках одного запроса, но необходимо изменить между запросами.

ПРИМЕЧАНИЕ Если ваши сервисы с жизненными циклами scoped или transient реализуют интерфейс `IDisposable`, контейнер внедрения зависимостей автоматически освобождает ресурсы, когда область действия заканчивается.

В целом вы встретите множество сервисов, зарегистрированных с использованием данного жизненного цикла, особенно это касается всего, что использует базу данных, зависит от деталей HTTP-запроса или применяет сервис с жизненным циклом scoped. Но некоторые сервисы не нуждаются в изменении между запросами, например сервис, вычисляющий площадь круга или возвращающий текущее время в разных часовых поясах. Для них более подходящим может быть жизненный цикл singleton.

9.4.3 Singleton: может быть только один

Синглтон, или одиночка, – это паттерн, появившийся еще до внедрения зависимостей; контейнер обеспечивает его надежную и простую

в использовании реализацию. Синглтон концептуально прост: экземпляр сервиса создается при первой необходимости (или во время регистрации, как в разделе 9.2), и все. Вы всегда будете получать один и тот же экземпляр, внедренный в ваши сервисы.

Данный паттерн особенно полезен для объектов, создание которых связано с большими затратами, которые содержат данные для совместного использования в запросах, или объектов, не имеющих состояния. Последние два момента важны – любой сервис, зарегистрированный как синглтон, должен быть потокобезопасным.

ВНИМАНИЕ! Сервисы-одиночки должны быть потокобезопасными в веб-приложении, поскольку обычно они будут использоваться несколькими потоками во время конкурентных запросов.

Посмотрим, что означает использование синглтонов на примере подсчета строк. Я могу обновить регистрацию `DataContext`, чтобы он стал одиночкой:

```
builder.Services.AddSingleton<DataContext>();
```

Затем можно вызвать обработчик `RowCounts` и понаблюдать за результатами, показанными на рис. 9.9. Видно, что каждый экземпляр вернул одно и то же значение. Это указывает на то, что один и тот же экземпляр `DataContext` используется в каждом запросе, как при внедрении непосредственно в обработчик конечной точки, так и при транзитивной ссылке из `Repository`.

RowCount одинаков в пределах одного запроса.
Это указывает на то, что в графе зависимостей
используется один объект DataContext

RowCount одинаков для
всех запросов, что
указывает на то, что для
каждого запроса
используется один и тот
же DataContext

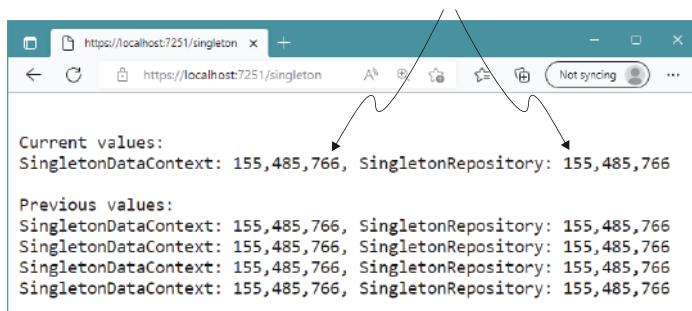


Рис. 9.9 Любой сервис, зарегистрированный как синглтон, всегда будет возвращать один и тот же экземпляр. Следовательно, все вызовы обработчика `RowCounts` возвращают одно и то же значение как внутри запроса, так и между запросами

Синглтоны удобны для объектов, которые необходимо использовать совместно или которые являются неизменяемыми и их затратно создавать. Сервис кеширования должен быть синглтоном, поскольку всем запросам нужно использовать его совместно. Однако он должен быть потокобезопасным. Точно так же можно зарегистрировать объ-

ект настроек, загружаемый с удаленного сервера как синглтон, если вы загружаете настройки один раз при запуске и повторно используете их в течение всего жизненного цикла вашего приложения.

На первый взгляд, выбор жизненного цикла сервиса может показаться не слишком сложным, но есть одна серьезная «ловушка», которая может поджидать вас, и в следующем разделе вы в этом убедитесь.

9.4.4 Следите за захваченными зависимостями

Представьте, что вы настраиваете жизненный цикл для примеров с классами `DataContext` и `Repository`. Вы отталкиваетесь от моих рекомендаций и выбираете следующие циклы:

- `DataContext` – scoped, поскольку он должен использоваться совместно для одного запроса;
- `Repository` – singleton, поскольку он не имеет собственного состояния и является потокобезопасным, так почему бы и нет?

ВНИМАНИЕ! Данная конфигурация жизненного цикла предназначена для изучения ошибки – не используйте ее в коде, иначе вы столкнетесь с аналогичной проблемой!

К сожалению, вы создали захваченную зависимость, потому что внедряете scoped-объект, `DataContext`, в синглтон, `Repository`. Поскольку это синглтон, один и тот же экземпляр класса `Repository` используется на протяжении всего жизненного цикла приложения, поэтому внедренный в него объект `DataContext` также будет существовать весь жизненный цикл приложения, хотя при каждом запросе следует использовать новый. На рис. 9.10 показан сценарий, в котором новый экземпляр `DataContext` создается для каждой области, но экземпляр внутри `Repository` находится там в течение всего жизненного цикла приложения.

Захваченные зависимости могут вызывать небольшие ошибки, которые трудно устраниить, поэтому следует всегда следить за ними. Такие зависимости относительно легко внедрить, поэтому тщательно все обдумайте перед регистрацией синглтона.

ВНИМАНИЕ! Сервис должен использовать только те зависимости, жизненный цикл которых превышает или эквивалентен жизненному циклу сервиса. Сервис, зарегистрированный как синглтон, может безопасно использовать только singleton-зависимости. Сервис, зарегистрированный как scoped, может безопасно использовать scoped- или singleton-зависимости. Кратковременный сервис может использовать зависимости с любым жизненным циклом.

Здесь я должен упомянуть, что в этом предупреждении есть проблеск надежды. ASP.NET Core автоматически проверяет эти типы захваченных зависимостей и выбросит исключение при запуске приложения, если обнаружит их, как показано на рис. 9.11.

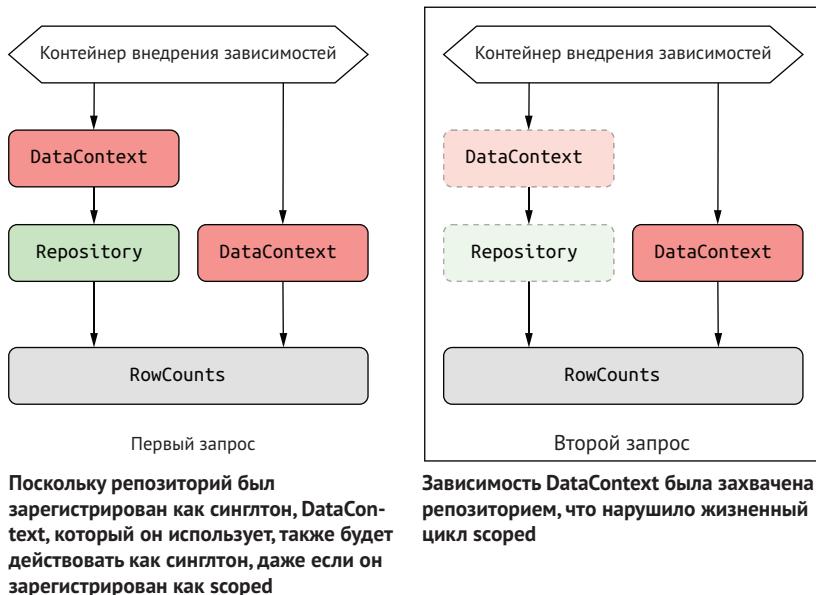


Рис. 9.10 **DataContext** зарегистрирован как scoped-зависимость, но **Repository** – это синглтон. Даже если вы ожидаете новый **DataContext** для каждого запроса, **Repository** захватывает внедренный **DataContext** и заставляет вас повторно использовать его на протяжении жизненного цикла приложения

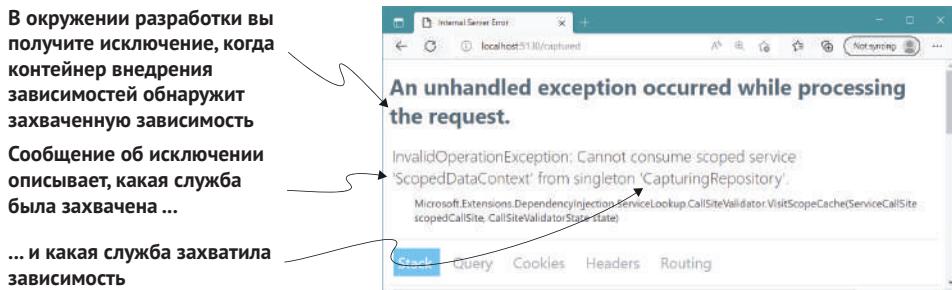


Рис. 9.11 При активации **ValidateScopes** контейнер внедрения зависимостей выбросит исключение, когда будет создавать сервис с захваченной зависимостью. По умолчанию эта проверка включена только для окружения разработки

Такая проверка влияет на производительность, поэтому по умолчанию она активируется, только когда приложение работает в окружении разработки, но это должно помочь обнаружить большинство проблем такого рода. Можно включить или отключить эту проверку независимо от окружения, задав параметр **ValidateScopes** для **WebApplicationBuilder** в файле **Program.cs**, как показано в следующем листинге:

Листинг 9.13. Задаем свойство ValidateScopes для постоянной проверки областей

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args); <-- Компоновщик по умолчанию задает ValidateScopes для проверки только в окружении разработки

builder.Host.UseDefaultServiceProvider(o => <-- Вы можете переопределить проверку с помощью расширения UseDefaultServiceProvider
{
    o.ValidateScopes = true;
    o.ValidateOnBuild = true; <-- ValidateOnBuild проверяет, что для каждого зарегистрированного сервиса зарегистрированы все его зависимости
});
```

Если для этого параметра установлено значение true, то области будут проверяться во всех окружениях, что влияет на производительность

В листинге 9.13 показан еще один параметр, который можно активировать, `ValidateOnBuild`. Он идет еще дальше. Когда он активирован, контейнер внедрения зависимостей при запуске приложения проверяет наличие зависимостей, зарегистрированных для каждого сервиса, который необходимо создать. Если этого не происходит, он генерирует исключение, как показано на рис. 9.12, сообщая о неправильной конфигурации. Это также влияет на производительность, поэтому по умолчанию доступно только в окружении разработки, но очень полезно, чтобы указать на все пропущенные регистрации сервисов.

ВНИМАНИЕ! К сожалению, контейнер не может отловить все ошибки. Список предостережений и исключений см. в посте моего блога: <http://mng.bz/QmwG>.

Мы уже почти все рассказали о внедрении зависимостей, и остается рассмотреть только один момент: как разрешить сервисы с жизненным циклом scoped при запуске приложения в файле `Program.cs`.

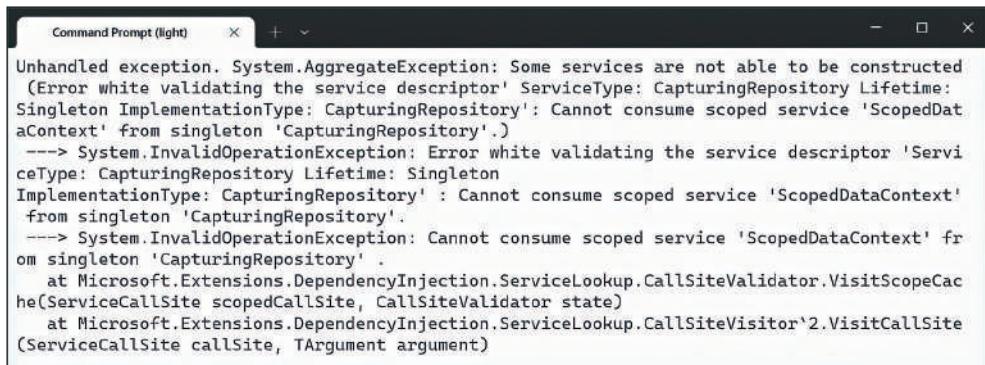


Рис. 9.12 Если `ValidateOnBuild` активирован, при запуске приложения контейнер проверит, может ли он создать все зарегистрированные сервисы. Если он находит сервис, который не может создать, то выбрасывает исключение. По умолчанию такая проверка включена только для окружения разработки

9.5 Разрешение сервисов с жизненным циклом scoped за пределами запроса

В главе 8 я говорил, что существует два основных способа разрешения сервисов из контейнера внедрения зависимостей для минимальных API:

- внедрение сервисов в обработчик конечной точки;
- обращение к контейнеру внедрения зависимостей непосредственно в файле Program.cs.

В этой главе вы уже несколько раз видели первый подход. В главе 8 было показано, что можно получить доступ к сервисам в файле Program.cs, вызвав метод `GetRequiredService<T>()` для `WebApplication.Services`:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
var settings = app.Services.GetRequiredService<EmailServerSettings>();
```

Однако важно, чтобы таким образом запрашивались только сервисы с жизненным циклом `singleton`. `IServiceProvider`, представленный как `WebApplication.Services`, является корневым контейнером внедрения зависимостей для вашего приложения. Сервисы, запрошенные таким образом, действуют в течение всего жизненного цикла приложения, что подходит для сервисов с жизненным циклом `singleton`, но обычно это не то поведение, которое нужно для сервисов с жизненными циклами `scoped` или `transient`.

ВНИМАНИЕ! Не запрашивайте сервисы с жизненными циклами `scoped` или `transient` непосредственно из `WebApplication.Services`. Такой подход может привести к утечке памяти, поскольку объекты остаются активными до выхода из приложения и не подвергаются сборке мусора.

Вместо этого следует запросить сервисы только с жизненными циклами `scoped` или `transient` из активной области. Новая область создается автоматически для каждого HTTP-запроса, но, когда вы запрашиваете сервисы из контейнера внедрения зависимостей непосредственно в файле `Program.cs` (или где-либо еще, вне контекста HTTP-запроса), необходимо создавать (и удалять) область вручную.

Можно создать новую область, вызвав методы `CreateScope()` или `CreateAsyncScope()` для `IServiceProvider`, который возвращает одноразовый объект `IServiceScope`, как показано на рис. 9.13. `IServiceScope` также предоставляет свойство `IServiceProvider`, но любые сервисы, разрешенные от этого поставщика, освобождаются автоматически при удалении `IServiceScope`, гарантируя, что все ресурсы, удерживаемые сервисами с жизненными циклами `scoped` и `transient`, будут высвобождены правильно.

В следующем листинге показано, как разрешить сервис с жизненным циклом `scoped` в файле `Program.cs`, используя шаблон, показанный на рис. 9.13. Этот шаблон гарантирует, что объект `DataContext` с жизненным циклом `scoped` будет правильно освобожден перед вызовом метода `app.Run()`.

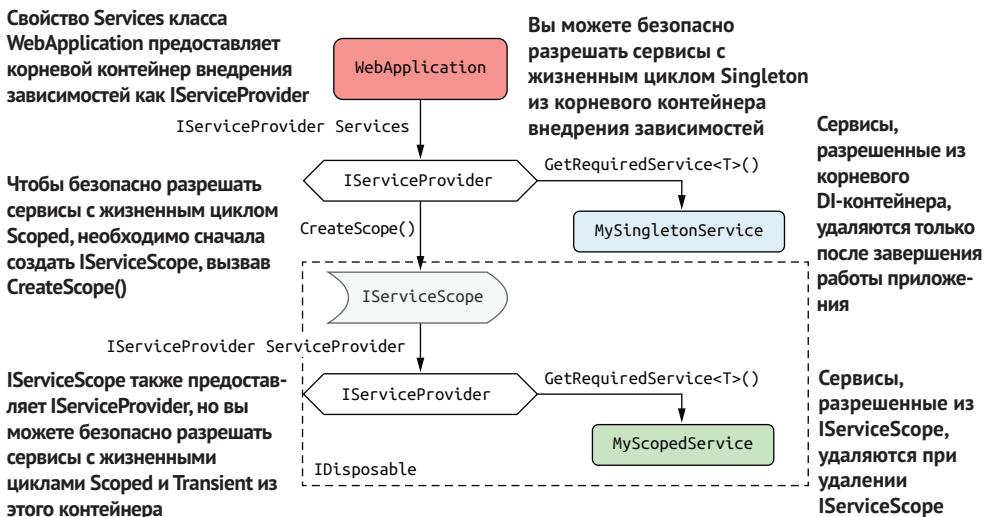


Рис. 9.13 Чтобы запросить сервисы с жизненными циклами scoped или transient вручную, необходимо создать объект IServiceScope, вызвав метод CreateScope() для WebApplication.Services. Любые сервисы с жизненными циклами scoped или transient, запрошенные из контейнера внедрения зависимостей, представленного как IServiceScope.ServiceProvider, освобождаются автоматически при удалении объекта IServiceScope.

Листинг 9.14. Разрешение сервиса с жизненным циклом scoped с помощью IServiceScope в файле Program.cs

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<DataContext>(); <-- DataContext зарегистрирован как scoped, поэтому его не следует разрешать непосредственно из app.Services

WebApplication app = builder.Build();

await using (var scope = app.Services.CreateAsyncScope()) <-- Создает IServiceScope
{
    var dbContext =
        scope.ServiceProvider.GetRequiredService<DataContext>();
    Console.WriteLine($"Retrieved scope: {dbContext.RowCount}");
}

app.Run();
```

При удалении IServiceScope все разрешенные сервисы также освобождаются

Запрашивает сервис с жизненным циклом scoped из scoped-контейнера

В этом примере используется асинхронная форма CreateAsyncScope() вместо CreateScope(), которой обычно следует отдавать предпочтение, когда это возможно. Метод CreateAsyncScope был представлен в .NET 6, чтобы исправить частный случай, связанный с IAsyncDisposable (представленный в .NET Core 3.0). Подробнее об этом сценарии можно прочитать в моем блоге на странице <http://mng.bz/zXGB>.

На этом наше знакомство с внедрением зависимостей в ASP.NET Core подошло к концу. Теперь вы знаете, как регистрировать соб-

ственными сервисами с помощью контейнера внедрения зависимостей, и в идеале должны хорошо понимать три жизненных цикла сервисов, используемых в .NET. Внедрение зависимостей встречается в .NET повсеместно, поэтому важно постараться разобраться в этом.

В следующей главе мы рассмотрим модель конфигурации ASP.NET Core. Вы увидите, как загружать настройки из файла во время выполнения, как безопасно хранить конфиденциальные настройки и как заставить приложение вести себя по-разному в зависимости от того, на какой машине оно работает. Мы даже немного применим внедрение зависимостей; оно присутствует в ASP.NET Core повсюду!

Резюме

- Регистрируя сервисы, вы описываете: тип сервиса, тип реализации и жизненный цикл. Тип сервиса определяет, какой класс или интерфейс будет запрошен в качестве зависимости. Тип реализации – это класс, который контейнер должен создать для выполнения зависимости. Жизненный цикл – это период, в течение которого следует использовать экземпляр сервиса;
- можно зарегистрировать сервис с помощью обобщенных методов, если класс является конкретным и все аргументы его конструктора зарегистрированы в контейнере или имеют значения по умолчанию;
- можно предоставить экземпляр сервиса во время регистрации, которая регистрирует этот экземпляр как синглтон. Данный подход может быть полезен, если у вас уже есть доступный экземпляр сервиса;
- можно предоставить фабричную лямбда-функцию, которая описывает, как создать экземпляр сервиса с любым выбранным вами жизненным циклом. Данный подход можно использовать, если ваши сервисы зависят от других сервисов, которые доступны только во время работы приложения;
- по возможности избегайте вызова методов `GetService()` или `GetRequiredService()` в фабричных функциях. Вместо этого отдайте предпочтение внедрению через конструктор – это более эффективно;
- можно зарегистрировать несколько реализаций для сервиса. Затем вы можете внедрить `IEnumerable<T>`, чтобы получить доступ ко всем реализациям во время выполнения;
- если вы внедряете один экземпляр многократно зарегистрированного сервиса, контейнер внедряет последнюю зарегистрированную реализацию;
- вы можете использовать методы расширения `TryAdd*`, чтобы гарантировать регистрацию реализации только в том случае, если не было зарегистрировано никакой другой реализации сервиса. Такой подход может быть полезен авторам библиотек для добавления сервисов по умолчанию, в то же время позволяя потребителям переопределять зарегистрированные сервисы;

- мы определяем жизненный цикл сервиса во время регистрации сервиса, чтобы определить, когда контейнер внедрения зависимостей будет повторно использовать существующий экземпляр сервиса для выполнения зависимостей службы, а когда он создаст новый;
- жизненный цикл transient означает, что каждый раз, когда запрашивается сервис, создается новый экземпляр;
- жизненный цикл scoped означает, что внутри области все запросы к сервису дадут вам один и тот же объект. Для разных областей вы получите разные объекты. В ASP.NET Core каждый веб-запрос получает собственную область;
- вы всегда будете получать один и тот же экземпляр singleton-сервиса независимо от области;
- сервис должен использовать только зависимости с жизненным циклом, превышающим или равным жизненному циклу сервиса. По умолчанию ASP.NET Core выполняет валидацию области для проверки ошибок, подобных этой, и выдает исключение при их обнаружении, но эта функция активирована только в окружении разработки, поскольку влияет на производительность;
- чтобы получить доступ к сервисам с жизненным циклом в файле Program.cs, необходимо сначала создать объект IServiceScope, вызвав методы CreateScope() или CreateAsyncScope() для WebApplication.Services. Вы можете разрешить сервисы из свойства ServiceProvider. При удалении IServiceScope также удаляются все сервисы с жизненными циклами scoped или transient, разрешенные из области действия.

10

Конфигурирование приложения ASP. Net Core

В этой главе:

- загрузка настроек от нескольких поставщиков конфигурации;
- безопасное хранение конфиденциальных настроек;
- использование строго типизированных объектов настроек;
- использование собственных настроек для разных окружений размещения.

В первой части этой книги вы познакомились с основами создания и запуска приложения ASP.NET Core и узнали, как использовать конечные точки минимального API для создания HTTP API. Как только вы начнете создавать реальные приложения, то быстро поймете, что вам нужно настраивать различные параметры во время развертывания, без необходимости повторной компиляции приложения. В этой главе рассматривается, как этого добиться в ASP.NET Core, используя конфигурацию.

Знаю. «Конфигурация» звучит скучно, правда? Но должен признаться, что модель конфигурации – одна из моих любимых частей ASP.NET Core. Ее очень легко использовать, и она намного элегант-

нее, чем в предыдущих версиях ASP.NET. В разделе 10.2 вы узнаете, как загружать значения из множества источников – файлов JSON, переменных окружения и аргументов командной строки – и объединять их в единый объект конфигурации.

В добавок ко всему ASP.NET Core дает возможность легко привязать эту конфигурацию к строго типизированным объектам параметров. Это простые РОСО-классы (plain old CLR object), которые формируются из объекта конфигурации и которые можно внедрять в свои сервисы, как вы увидите в разделе 10.3. Это позволяет красиво инкапсулировать настройки для различных функций в своем приложении. В последнем разделе этой главы вы узнаете об окружении размещения ASP.NET Core. Часто нужно, чтобы приложение работало по-разному в разных ситуациях, например при запуске на компьютере разработчика или при развертывании на рабочем сервере. Такие ситуации называются *окружениями*. Если приложение будет знать, в каком окружении оно работает, то сможет загружать нужную конфигурацию и соответственно изменять свое поведение.

Прежде чем начать, рассмотрим основы: что такое конфигурация, для чего она нам и как ASP.NET Core справляется с этими требованиями?

10.1 Представляем модель конфигурации ASP.NET Core

В этом разделе я кратко опишу, что мы понимаем под конфигурацией и для чего ее можно использовать в приложениях ASP.NET Core. Конфигурация – это набор внешних параметров, предоставляемых приложению, который так или иначе контролирует его поведение. Обычно он состоит из комбинации настроек и секретов, которые приложение будет загружать во время выполнения.

ОПРЕДЕЛЕНИЕ *Настройка* – это любое значение, изменяющее поведение вашего приложения. *Секрет* – это особый тип настройки, содержащий конфиденциальные данные, такие как пароль, API-ключ для стороннего сервиса или строка подключения.

Перед тем как мы начнем, есть очевидный вопрос. Нужно подумать, для чего вам нужна конфигурация приложения и что нужно настроить. Как правило, нужно выносить из кода приложения в конфигурацию все, что можно рассматривать как настройку или секрет. Таким образом, можно легко изменить эти значения во время развертывания, без необходимости повторно компилировать приложение.

Например, у вас может быть приложение, которое показывает расположение ваших магазинов. У вас может быть настройка строки подключения к базе данных, в которой вы храните подробную информацию о магазинах, а также такие настройки, как местоположение по умолчанию для отображения на карте, используемый по умолчанию уровень масштабирования и API-ключ для доступа к API Карт Google, как показано на рис. 10.1. Если вы храните эти настройки и секреты

за пределами своего кода, то это хорошая практика, так как позволяет легко настраивать их без необходимости повторной компиляции.

Здесь также присутствует и аспект безопасности; вам не нужно вшивать в код секретные значения, такие как API-ключи или пароли, где они могут быть переданы в систему контроля версий и стать общедоступными. Даже значения, встроенные в скомпилированное приложение, можно извлечь, поэтому по возможности лучше вынести их за его пределы кода.

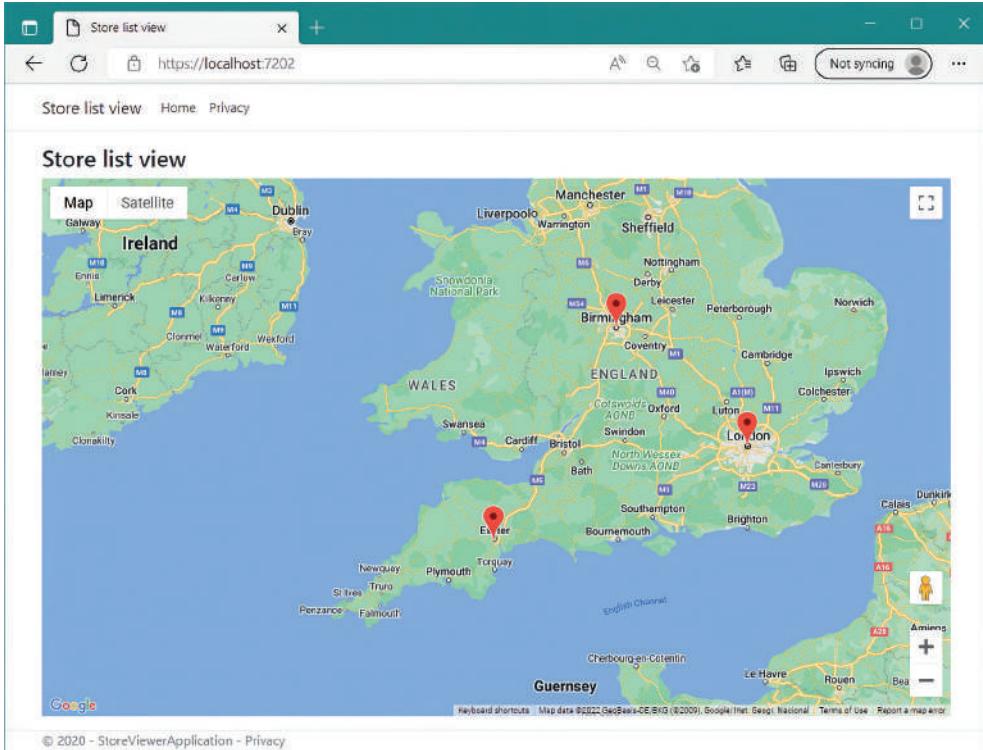


Рис. 10.1 Можно хранить местоположение по умолчанию, уровень масштабирования и API-ключ сопоставления в конфигурации и загружать их во время выполнения. Важно хранить секреты, такие как API-ключи, в конфигурации за пределами кода

Практически каждый веб-фреймворк предоставляет механизм для загрузки конфигурации, и предыдущая версия ASP.NET в этом ничем не выделялась. В ней использовался элемент `<appsettings>` из файла `web.config` для хранения пар конфигурации «ключ–значение». Во время выполнения вы использовали статический (*вам уже страшно?*) класс `ConfigurationManager` для загрузки значения определенного ключа из файла. Можно было делать и более сложные вещи, используя настраиваемые разделы конфигурации, но это было неудобно и, по моему опыту, использовалось редко.

ASP.NET Core предоставляет вам полностью обновленный интерфейс. На самом базовом уровне вы по-прежнему наблюдаете пару «ключ–значение» как строки, но вместо того, чтобы получать эти значения из одного файла, теперь вы можете загружать их из нескольки-

ких источников. Можно загружать значения из файлов, но теперь они могут иметь любой формат: JSON, XML, YAML и т. д. Кроме того, вы можете загружать значения из переменных окружения, аргументов командной строки, базы данных или из удаленной службы. Или можете создать собственного поставщика конфигурации.

ОПРЕДЕЛЕНИЕ ASP.NET Core использует *поставщиков конфигурации* для загрузки пар «ключ–значение» из множества источников. Приложения могут использовать множество различных поставщиков конфигурации.

У модели конфигурации ASP.NET Core также есть концепция переопределения настроек. Каждый поставщик конфигурации может определять собственные настройки или переопределять настройки от предыдущего поставщика. Вы увидите эту невероятно полезную функцию в действии в разделе 10.2.

ASP.NET Core упрощает привязку пар «ключ–значение», которые определены как строки, `strings`, к классам настройки РОСО, которые вы определяете в своем коде. Такая модель сильно типизированной конфигурации позволяет легко логически сгруппировать настройки по функциям приложения и хорошо поддается модульному тестированию.

Прежде чем перейти к деталям загрузки конфигурации от поставщиков, мы сделаем шаг назад и посмотрим, где происходит этот процесс, – мы рассмотрим, как вы загружаете настройки и секреты для своего приложения, хранятся ли они в файлах JSON, переменных окружения или аргументах командной строки.

10.2 Создание объекта конфигурации для приложения

В этом разделе мы познакомимся с самой сутью системы конфигурации. Вы узнаете, как загружать настройки из нескольких источников, как они хранятся внутри ASP.NET Core и как могут переопределять другие значения, создавая уровни конфигурации. Вы также узнаете, как безопасно хранить секреты, обеспечивая их доступность при запуске приложения.

Модель конфигурации ASP.NET Core практически не изменилась со времен .NET Core 1.0, но в .NET 6 в ASP.NET Core появился класс `ConfigurationManager`. `ConfigurationManager` упрощает общие шаблоны работы с конфигурацией, реализуя оба основных интерфейса, связанных с конфигурацией: `IConfigurationBuilder` и `IConfigurationRoot`.

ПРИМЕЧАНИЕ `IConfigurationBuilder` описывает, как построить окончательное представление конфигурации для приложения, а `IConfigurationRoot` содержит сами значения конфигурации.

Конфигурация описывается путем добавления объектов `IConfigurationProvider` в `ConfigurationManager`. Поставщики конфигурации описывают, как загрузить пары «ключ–значение» из определенного источника,

например JSON-файла или переменных окружения, как показано на рис. 10.2. Когда вы добавляете поставщика, ConfigurationManager запрашивает его и добавляет все значения, возвращаемые в реализацию интерфейса IConfigurationRoot.

ПРИМЕЧАНИЕ При добавлении поставщика в ConfigurationManager значения конфигурации добавляются в экземпляр IConfigurationRoot, реализующий интерфейс IConfiguration. Обычно вы будете работать с этим интерфейсом.

В состав ASP.NET Core входят поставщики конфигурации для загрузки данных из распространенных местоположений:

- файлы JSON;
- XML-файлы;
- переменные окружения;
- аргументы командной строки;
- файлы INI.

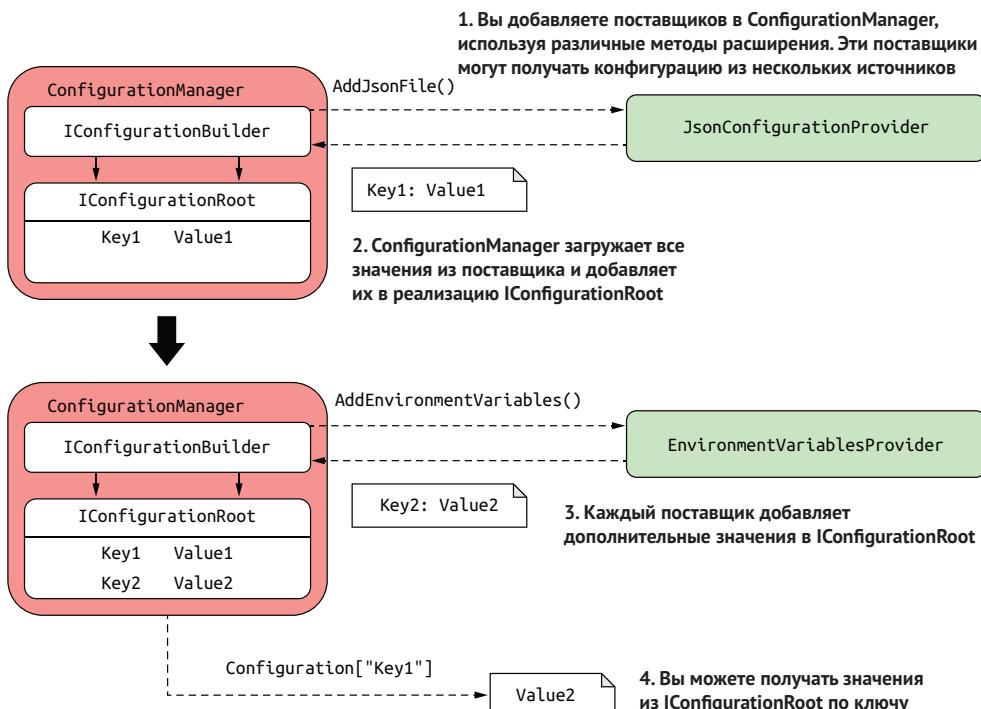


Рис. 10.2 Использование ConfigurationManager для заполнения IConfiguration. Поставщики конфигурации добавляются в ConfigurationManager с помощью методов расширения. Менеджер запрашивает поставщика и добавляет все возвращаемые значения в IConfigurationRoot, который реализует IConfiguration

Если они не соответствуют вашим требованиям, то можно найти множество альтернатив на GitHub и NuGet или написать свой (это не-

сложно). Например, можно использовать официальный поставщик Azure Key Vault или поставщик файлов YAML, который написал я.

ПРИМЕЧАНИЕ Поставщик Azure Key Vault доступен в NuGet на странице <http://mng.bz/OKrN>, а поставщика файлов YAML можно найти на GitHub: <http://mng.bz/Yqdj>.

Во многих случаях достаточно поставщиков по умолчанию. В частности, большинство шаблонов начинаются с файла appsettings.json, который содержит различные настройки в зависимости от выбранного вами шаблона. В следующем листинге показан файл по умолчанию, созданный шаблоном ASP.NET Core 7.0 Empty без аутентификации.

Листинг 10.1. Файл appsettings.json по умолчанию, созданный с помощью шаблона Empty

```
{  
    "Logging": {  
        "LogLevel": {  
            "Default": "Information",  
            "Microsoft.AspNetCore": "Warning"  
        }  
    },  
    "AllowedHosts": "*"  
}
```

Как видите, этот файл содержит в основном настройки для управления журналированием, но вы также можете добавить сюда дополнительную конфигурацию для своего приложения.

ВНИМАНИЕ! Не храните в этом файле конфиденциальные значения, например пароли, ключи API и строки подключения. Вы увидите, как безопасно хранить эти значения, в разделе 10.2.3.

Добавление собственных значений конфигурации предполагает добавление пары «ключ–значение» в JSON-файл. Хорошой идеей будет использовать пространство имён для ваших настроек, создав корневой объект для связанных настроек, как в объекте `MapSettings`, показанном в следующем листинге.

Листинг 10.2. Добавление значений конфигурации в файл appsettings.json

```
{  
    "Logging": {  
        "LogLevel": {  
            "Default": "Information",  
            "Microsoft": "Warning",  
            "Microsoft.Hosting.Lifetime": "Information"  
        }  
    },  
    "AllowedHosts": "*",  
}
```

```

    "MapSettings": {
        "DefaultZoomLevel": 9,
        "DefaultLocation": {
            "latitude": "50.500",
            "longitude": "-4.000"
        }
    }
}

Вкладываем всю конфигурацию в ключ MapSettings
Значения могут быть числами в файле JSON,
но при чтении они будут преобразованы
в строки
Можно создавать глубоко вложенные структуры
для лучшей организации конфигурации

```

Я вложил новую конфигурацию в родительский ключ `MapSettings`, чтобы создать «секцию», которая будет полезна позже, когда дело дойдет до привязки значений к объекту POCO. Я также вложилключи `latitude` и `longitude` в ключ `DefaultLocation`. Вы можете создать любую структуру, которая вам нравится; поставщик конфигурации прочитает их без проблем. Кроме того, можно хранить их как данные любого типа – в данном случае это числа, – но имейте в виду, что поставщик будет читать и хранить их как строки.

СОВЕТ Ключи конфигурации в вашем приложении *не* чувствительны к регистру, поэтому имейте это в виду при загрузке из поставщиков, у которых ключи чувствительны к регистру. Например, если у вас есть файл YAML с ключами `name` и `NAME`, в окончательной версии `IConfiguration` появится только один ключ.

Теперь, когда у вас есть файл конфигурации, пора нашему приложению загрузить его в `ConfigurationManager`.

10.2.1 Добавление поставщика конфигурации в файле `Program.cs`

Как уже было показано в этой книге, ASP.NET Core (начиная с версии .NET 6) использует класс `WebApplicationBuilder` для загрузки приложения. В рамках процесса начальной загрузки `WebApplicationBuilder` создает экземпляр `ConfigurationManager` и предоставляет его как свойство `Configuration`.

СОВЕТ Доступ к `ConfigurationManager` можно получить непосредственно в `WebApplicationBuilder.Configuration` и `WebApplication.Configuration`. Оба свойства ссылаются на один и тот же экземпляр `ConfigurationManager`.

`WebApplicationBuilder` добавляет в `ConfigurationManager` несколько поставщиков конфигурации по умолчанию, которые мы рассмотрим более подробно в этой главе:

- *поставщик файлов JSON* – загружает настройки из файла `appsettings.json`, а также загружает настройки из необязательного файла для конкретного окружения, `appsettings.ENVIRONMENT.json`. Я покажу, как использовать эти файлы, в разделе 10.4;
- *пользовательские секреты* – загружает секреты, которые надежно хранятся во время разработки;

- *переменные окружения* – загружает переменные окружения в качестве переменных конфигурации. Они отлично подходят для хранения секретов в промышленном окружении;
- *аргументы командной строки* – использует значения, переданные в качестве аргументов при запуске приложения.

`ConfigurationManager` автоматически настраивается со всеми этими источниками, но вы можете легко добавить дополнительных поставщиков. Вы также можете начать с нуля и очистить поставщиков по умолчанию, как показано в следующем листинге, который полностью определяет, откуда загружается конфигурация.

Листинг 10.3. Загрузка файла `appsettings.json` путем очистки источников конфигурации

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args); ←
    Очищает поставщиков,
    настроенных по умолчанию
builder.Configuration.Sources.Clear(); ←
    в WebApplicationBuilder
builder.Configuration.AddJsonFile("appsettings.json", optional: true); ←
    Добавляет поставщика кон-
    фигурации JSON, задавая
    имя файла конфигурации
WebApplication app = builder.Build(); ←
    Возвращает все пары конфигурации
app.MapGet("/", () => app.Configuration.AsEnumerable()); ←
    «ключ–значение» для отображения
app.Run();
```

В этом примере добавлен один поставщик конфигурации JSON путем вызова метода расширения `AddJsonFile()` и предоставления имени файла. Он также задает для параметра `optional` значение `true`, веля поставщику конфигурации пропускать файлы, которые он не может найти во время выполнения, вместо того чтобы выбросить исключение `FileNotFoundException`. При добавлении поставщика `ConfigurationManager` запрашивает все доступные значения у поставщика и добавляет их в реализацию `IConfiguration`.

ConfigurationBuilder и ConfigurationManager

До .NET 6 и появления `ConfigurationManager` конфигурация в ASP.NET Core реализовывалась с помощью `ConfigurationBuilder`. Поставщики конфигурации добавлялись к типу построителя так же, как это делалось с `ConfigurationManager`, но значения конфигурации не загружались до тех пор, пока не вызывали метод `Build()`, который создавал окончательный объект `IConfigurationRoot`.

Напротив, в .NET 6 и .NET 7 `ConfigurationManager` действует как построитель, так и окончательный `IConfigurationRoot`. При добавлении нового поставщика конфигурации значения конфигурации незамедлительно добавляются в `IConfigurationRoot` без необходимости предварительно го вызова метода `Build()`.

Подход с ConfigurationBuilder с использованием паттерна построителя в некотором смысле чище, поскольку в нем присутствует более четкое разделение задач, но паттерны общего использования для конфигурации означают, что новый подход с ConfigurationManager зачастую проще в использовании.

При желании вы все равно можете использовать паттерн построителя, открав WebApplicationBuilder.Host.ConfigureAppConfiguration. О некоторых из этих шаблонов и различиях между двумя подходами можно прочитать в моем блоге: <http://mng.bz/Ke4j>.

Можно получить доступ к объекту IConfiguration непосредственно в файле Program.cs, как показано в листинге 10.3, но ConfigurationManager также регистрируется как IConfiguration в контейнере внедрения зависимостей, поэтому вы можете внедрить его в свои классы и обработчики конечных точек. Можно переписать обработчик конечной точки из листинга 10.3 следующим образом, и объект IConfiguration будет внедрен в обработчик с помощью внедрения зависимостей:

```
app.MapGet("/", (IConfiguration config) => config.AsEnumerable());
```

ПРИМЕЧАНИЕ ConfigurationManager реализует IConfigurationRoot, который также реализует IConfiguration. ConfigurationManager регистрируется в контейнере внедрения зависимостей как IConfiguration, а не IConfigurationRoot.

Вы видели, как добавлять значения в ConfigurationManager с помощью таких поставщиков, как поставщик файлов JSON, а в листинге 10.3 показан пример перебора каждого значения конфигурации, но обычно нужно получить конкретное значение конфигурации.

IConfiguration хранит конфигурацию как набор пар строк «ключ–значение». Вы можете получить доступ к любому значению по его ключу, используя стандартный синтаксис словаря. Вы могли бы использовать следующий код:

```
var zoomLevel = builder.Configuration["MapSettings:DefaultZoomLevel"];
```

чтобы получить настроенный уровень масштабирования для своего приложения (используя настройки, показанные в листинге 10.2). Обратите внимание, что я использовал двоеточие (:) для обозначения отдельного раздела. Аналогично, чтобы получить ключ широты, можно использовать

```
var lat = builder.Configuration["MapSettings:DefaultLocation:Latitude"];
```

Доступ к значениям настроек таким способом полезен в файле Program.cs при определении приложения. Например, при настройке приложения для подключения к базе данных строка подключения

часто загружается из объекта `IConfiguration`. Конкретный пример вы увидите в главе 12, посвященной Entity Framework Core.

Если вам нужен доступ к объекту конфигурации в других местах, кроме файла `Program.cs`, можно использовать внедрение зависимостей, чтобы внедрить его как зависимость в конструктор сервиса. Но доступ к конфигурации с помощью строковых ключей таким способом не особенно удобен; вместо этого следует попытаться использовать строго типизированную конфигурацию, как вы увидите в разделе 10.3.

Пока что, вероятно, все это выглядит немного запутанным и заурядным для загрузки настроек из файла JSON, и я согласен, что так оно и есть. Система конфигурации ASP.NET Core лучше всего проявляет себя в том случае, если у вас есть несколько поставщиков.

10.2.2 Использование нескольких поставщиков для переопределения значений конфигурации

Вы видели, как добавить поставщика конфигурации в `ConfigurationManager` и получить значения конфигурации, но пока мы сконфигурировали только одного поставщика. При добавлении поставщиков важно учитывать порядок, в котором вы их добавляете, поскольку он определяет порядок, в котором значения конфигурации добавляются в базовый словарь. Значения конфигурации от более поздних поставщиков будут перезаписывать значения с тем же ключом от более ранних поставщиков.

ПРИМЕЧАНИЕ Следует повторить: важен порядок, в котором вы добавляете поставщиков конфигурации в `ConfigurationManager`. Поставщики, зарегистрированные позже, могут перезаписывать значения более ранних поставщиков.

Рассматривайте поставщиков конфигурации как добавление «слоев» значений конфигурации к стеку, где каждый слой может перекрываться несколькими или всеми вышележащими слоями, как показано на рис. 10.3. Если новый поставщик содержит какие-либо ключи, которые уже известны диспетчеру конфигураций, они перезаписывают старые значения, чтобы создать окончательный набор значений конфигурации, хранящийся в `IConfiguration`.

СОВЕТ Вместо того чтобы думать о слоях, можно рассматривать `ConfigurationManager` как простой словарь. Когда вы добавляете поставщика, вы задаете пары «ключ–значение». Когда вы добавляете второго поставщика, он может добавлять новые ключи или перезаписывать значения существующих ключей.

Обновите свой код, чтобы загрузить конфигурацию от трех разных поставщиков конфигурации – двух поставщиков JSON и поставщика переменных окружения, – добавив их в `ConfigurationManager`, как показано в следующем листинге.

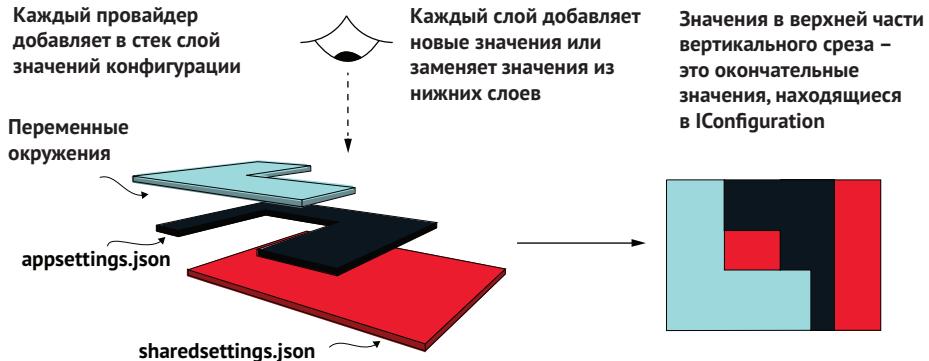


Рис. 10.3 Каждый поставщик конфигурации добавляет в ConfigurationBuilder «слой» значений. Вызов метода Build() сворачивает эту конфигурацию. Более поздние поставщики перезапишут значения конфигурации с теми же ключами, что и предыдущие поставщики

Листинг 10.4. Загрузка от нескольких поставщиков в файле Program.cs

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Configuration.Sources.Clear();           Загружает конфигурацию из другого
                                                файла конфигурации JSON перед
builder.Configuration                           файлом appsettings.json
    .AddJsonFile("sharedSettings.json", optional: true);
builder.Configuration.AddJsonFile("appsettings.json", optional: true);   Добавляет переменные
builder.Configuration.AddEnvironmentVariables();    окружения компьютера
                                                    в качестве поставщика
                                                    конфигурации

WebApplication app = builder.Build();
app.MapGet("/", () => app.Configuration.AsEnumerable());
app.Run();
```

Такой многослойный дизайн может быть полезен для ряда вещей. По сути, он позволяет объединить значения конфигурации из нескольких разных источников в один связный объект. Чтобы закрепить это на практике, рассмотрим значения конфигурации, приведенные на рис. 10.4.

Большинство настроек у каждого поставщика уникальны и добавляются в финальный IConfiguration. Но ключ "MyAppConnString" присутствует и в файле appsettings.json, и как переменная окружения. Поскольку поставщик переменных окружения добавлен после поставщиков JSON, значение конфигурации переменной окружения используется в IConfiguration.

Возможность сопоставлять конфигурацию от нескольких поставщиков – удобная черта сама по себе, но такой дизайн особенно полезен, когда речь идет о работе с конфиденциальными значениями конфигурации, такими как строки подключения и пароли. В следующем разделе показано, как решить эту проблему локально в окружении разработки и на рабочих серверах.

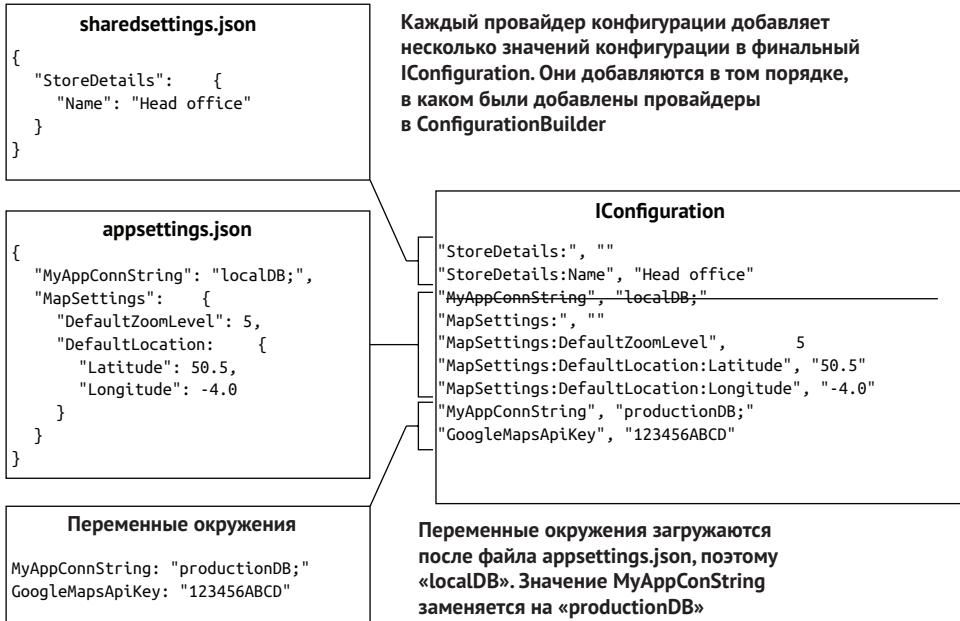


Рис. 10.4 Финальный `IConfiguration` включает в себя значения от каждого поставщика. И файл `appsettings.json`, и переменные окружения содержат ключ `MyAppConnString`. Поскольку переменные окружения добавляются позже, используется это значение конфигурации

10.2.3 Безопасное хранение секретов конфигурации

Как только вы создадите нетривиальное приложение, вы обнаружите, что вам нужно где-то хранить некоторые конфиденциальные данные. Это может быть пароль, строка подключения или, например, API-ключ для удаленной службы.

Хранить эти значения в файле `appsettings.json` – как правило, плохая идея, поскольку вы ни при каких обстоятельствах не должны передавать секреты в систему управления версиями; количество секретных ключей API, которые фиксируют в GitHub, просто ужасно! Вместо этого гораздо лучше хранить эти значения за пределами папки своего проекта, куда они не попадут случайно.

Это можно сделать несколькими способами, но самые простые и часто используемые подходы состоят в использовании переменных окружения для секретов на рабочем сервере и User Secrets, если вы делаете это локально. Ни один из подходов не является по-настоящему безопасным, поскольку они не хранят значения в зашифрованном виде. Если ваша машина скомпрометирована, злоумышленники смогут прочитать сохраненные значения, потому что они хранятся в виде открытого текста. Они предназначены для того, чтобы помочь вам избежать передачи секретов системе управления версиями.

СОВЕТ Azure Key Vault – безопасная альтернатива, поскольку хранит зашифрованные значения в Azure. Но вам все равно потребуется

использовать User Secrets и переменные окружения для хранения сведений о подключении к Azure Key Vault. Еще один популярный вариант – Vault от компании Hashicorp (www.vaultproject.io/), который можно запустить локально или в облаке.

Какой бы подход вы ни использовали для хранения секретов своего приложения, убедитесь, если это возможно, что вы не храните их в системе управления версиями. Даже частные репозитории могут не оставаться частными вечно, поэтому лучше проявлять осторожность.

Сохранение секретов в переменных окружения в промышленном окружении

Можно добавить поставщика конфигурации переменной окружения с помощью метода расширения `AddEnvironmentVariables`, как вы уже видели в листинге 10.4. Он добавляет все переменные окружения на вашем компьютере в виде пар «ключ–значение» в `ConfigurationManager`.

ПРИМЕЧАНИЕ По умолчанию `WebApplicationBuilder` добавляет переменные окружения в `ConfigurationManager`.

Можно создавать те же иерархические разделы в переменных окружения, которые вы обычно видите в файлах JSON, используя двоеточие (:) или двойное подчеркивание (__) для разграничения секций, например `MapSettings:MaxNumberofPoints` или `MapSettings__MaxNumberofPoints`.

СОВЕТ В некоторых окружениях, например в Linux, двоеточие в переменных окружения не допускается, и вместо него нужно использовать двойное подчеркивание. Двойное подчеркивание в переменных окружения будет преобразовано в двоеточие, когда те будут импортированы в объект `IConfiguration`. При извлечении значений из `IConfiguration` в свое приложение всегда следует использовать двоеточие.

Подход с использованием переменных окружения особенно полезен, когда вы публикуете свое приложение в автономное окружение, например выделенный сервер, Azure или контейнер Docker. Вы можете задать переменные окружения на своей промышленной машине или контейнере Docker, и поставщик прочитает их во время выполнения, переопределив значения по умолчанию, указанные в файлах `appsettings.json`.

СОВЕТ Инструкции по установке переменных окружения для своей операционной системы см. на странице <http://mng.bz/d4OD>.

Для компьютера, используемого при разработке, переменные окружения менее полезны, так как все ваши приложения будут применять те же значения. Например, если задать переменную окружения `ConnectionStrings_DefaultConnectionString`, она будет добавляться для каждого приложения, которое вы запускаете локально. Похоже, от этого больше хлопот, чем пользы!

СОВЕТ Чтобы избежать коллизий, можно добавлять только те переменные окружения, которые имеют заданный префикс, например `AddEnvironmentVariables("SomePrefix")`. Префикс удаляется из ключа перед его добавлением в `ConfigurationManager`, поэтому переменная `SomePrefix_MyValue` добавляется в конфигурацию как `MyValue`.

Для разработки можно использовать диспетчер пользовательских секретов (User Secrets Manager). Он эффективно добавляет переменные окружения для каждого приложения, поэтому у вас могут быть разные настройки для каждого приложения, но хранить вы их будете в другом месте, отличном от самого приложения.

Хранение секретов с помощью менеджера User Secrets в окружении разработки

Идея пользовательских секретов (User Secrets) состоит в том, чтобы упростить хранение секретов за пределами дерева проекта приложения. Это похоже на переменные окружения, но используется уникальный ключ для каждого приложения, чтобы хранить секреты отдельно.

ВНИМАНИЕ! Секреты не зашифрованы, поэтому их нельзя считать безопасными. Тем не менее этот способ лучше хранения в папке проекта.

Настройка User Secrets требует немного больше усилий, чем использование переменных окружения, поскольку вам нужно настроить инструмент для их чтения и записи, добавить поставщика конфигурации User Secrets и определить уникальный ключ для своего приложения.

- 1 ASP.NET Core по умолчанию включает поставщика User Secrets. Набор средств разработки .NET также включает в себя глобальный инструмент для работы с секретами из командной строки.
- 2 Если вы используете Visual Studio, щелкните проект правой кнопкой мыши и выберите **Manage User Secrets**. После этого откроется редактор файла `secrets.json`, в котором можно сохранить свои пары «ключ–значение», как если бы это был файл `appsettings.json`, как показано на рис. 10.5.
- 3 Добавьте уникальный идентификатор в свой файл с расширением `.csproj`. Visual Studio делает это автоматически, когда вы нажимаете **Manage User Secrets**, но если вы используете командную строку, то нужно будет добавить его самостоятельно. Обычно используется уникальный идентификатор, например GUID:

```
<PropertyGroup>
  <UserSecretsId>96eb2a39-1ef9-4d8e-8b20-8e8bd14038aa</UserSecretsId>
</PropertyGroup>
```

Вы также можете создать свойство `UserSecretsId` со случайным значением с помощью интерфейса командной строки .NET, выполнив следующую команду из папки проекта:

```
dotnet user-secrets init
```

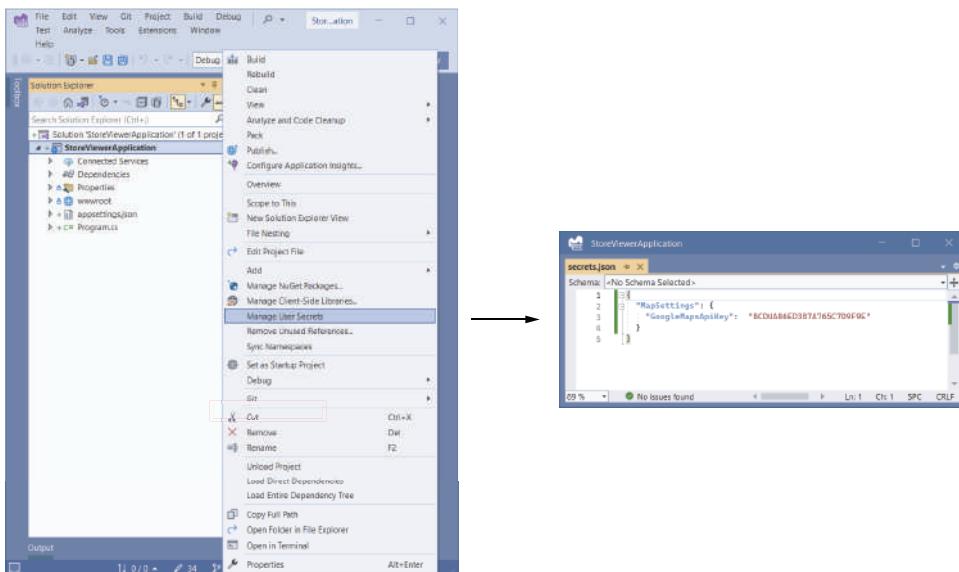


Рис. 10.5 Выберите *Manage User Secrets*, чтобы открыть редактор для приложения *User Secrets*. Можно использовать этот файл для хранения секретов при локальной разработке вашего приложения. Они хранятся за пределами папки вашего проекта и не попадут в систему управления версиями

4 Добавьте секреты пользователя с помощью командной строки

```
dotnet user-secrets set "MapSettings:GoogleMapsApiKey" F5RJT9GFHKR7
```

или отредактируйте файл *secret.json* напрямую с помощью своего любимого редактора. Точное расположение этого файла зависит от вашей операционной системы и может отличаться. Обратитесь к документации для получения подробной информации.

Уф, как много всего, а если вы настраиваете *ConfigurationManager*, то это еще не все! Вам необходимо обновить приложение, чтобы загрузить *User Secrets* во время выполнения с помощью метода расширения *AddUserSecrets*:

```
if (builder.Environment.IsDevelopment())
{
    builder.Configuration.AddUserSecrets<Program>();
}
```

ПРИМЕЧАНИЕ Рекомендуется использовать поставщика *User Secrets* только в окружении разработки, а не в промышленном окружении, поэтому в предыдущем фрагменте вы условно добавляете поставщика в *ConfigurationManager*. В промышленном окружении нужно использовать переменные окружения или Azure Key Vault, как уже обсуждалось ранее. Все это настраивается должным образом по умолчанию, если вы используете *WebApplicationBuilder*.

У метода `AddUserSecrets` есть несколько перегруженных вариантов, но самый простой – это обобщенный метод, который можно вызвать, передавая класс приложения `Program` в качестве обобщенного аргумента, как показано в предыдущем примере. Поставщик User Secrets должен прочитать свойство `UserSecretsId`, которое вы (или Visual Studio) добавили в файл с расширением `.csproj`. Класс `Program` работает как ссылка, чтобы указать, какая сборка содержит это свойство.

ПРИМЕЧАНИЕ Если вам интересно, то .NET SDK использует свойство `UserSecretsId` в файле с расширением `.csproj` для создания атрибута `UserSecretsIdAttribute` на уровне сборки. Затем поставщик считывает этот атрибут во время выполнения, чтобы определить `UserSecretsId` приложения, и, таким образом, генерирует путь к файлу `secrets.json`.

И вот оно – безопасное хранение ваших секретов за пределами папки проекта во время разработки. Это может показаться излишним, но если у вас есть нечто, что вы считаете конфиденциальным, к нему можно получить доступ удаленно и вам нужно загрузить это в конфигурацию, то настоятельно рекомендую вам использовать переменные окружения или User Secrets.

На этом пора оставить поставщиков конфигурации, но, прежде чем это сделать, я бы хотел показать вам один трюк системы конфигурации ASP.NET Core: перезагрузку файлов на лету.

10.2.4 Перезагрузка значений конфигурации при их изменении

Помимо безопасности, отсутствие необходимости перекомпилировать приложение каждый раз, когда вы хотите настроить значение, является одним из преимуществ использования конфигурации и настроек. В предыдущих версиях ASP.NET изменение настройки путем редактирования файла `web.config` приведет к тому, что ваше приложение необходимо будет перезапустить. Ждать, пока приложение запустится, прежде чем оно сможет обслуживать запросы, было немного затруднительно.

В ASP.NET Core вы, наконец, получаете возможность редактировать файл и автоматически обновлять конфигурацию приложения без перекомпиляции или перезапуска.

Часто упоминаемый сценарий, в котором вы, возможно, могли бы найти это полезным, – это когда вы пытаетесь отладить приложение в промышленном окружении. Обычно журналирование настраивается на один из этих уровней:

- ошибка (Error);
- предупреждение (Warning);
- информация (Information);
- отладка (Debug).

Каждая из этих настроек более подробна, чем предыдущая, но также предоставляет больше контекста. По умолчанию можно настроить свое приложение так, чтобы оно регистрировало только записи предупреж-

дений и ошибок в промышленном окружении, дабы не создавать слишком много лишних записей в журнале. Наоборот, если вы пытаетесь отладить какую-то проблему, вам нужно как можно больше информации, поэтому, возможно, вы захотите использовать уровень журнала Debug.

Возможность изменять конфигурацию во время выполнения означает, что вы можете легко включить дополнительные уровни журналирования, когда сталкиваетесь с проблемой, а затем переключить их обратно, отредактировав файл appsettings.json.

ПРИМЕЧАНИЕ Как правило, перезагрузка доступна только для файловых поставщиков конфигурации, таких как поставщик JSON, в отличие, например, от поставщика переменных окружения.

Можно активировать перезагрузку файлов конфигурации при добавлении любого из файловых поставщиков в ConfigurationManager. Методы расширения Add*File включают в себя перегруженный вариант с параметром reloadOnChange. Если для него задано значение true, то приложение будет отслеживать файловую систему для поиска изменений в файле и, если понадобится, инициирует полную повторную сборку IConfiguration. В следующем листинге показано, как добавить перезагрузку конфигурации в файл appsettings.json, добавленный вручную в ConfigurationManager.

Листинг 10.5 Перезагрузка appsettings.json при изменении файла

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Configuration.Sources.Clear();
builder.Configuration
    .AddJsonFile(
        "appsettings.json",
        optional: true,
        reloadOnChange: true); ← |Configuration будет
                                |собран снова, если
                                |файл appsettings.json
                                |изменится

WebApplication app = builder.Build();

app.MapGet("/", () => app.Configuration.AsEnumerable());

app.Run();
```

В этом разделе было показано, как настраивать поставщиков ConfigurationManager, очищая источники по умолчанию и добавляя свои собственные, но в большинстве случаев в этом нет необходимости. Как описано в разделе 10.2.1, поставщики по умолчанию, добавляемые WebApplicationBuilder, обычно достаточно хороши, если только вы не хотите добавить нового поставщика, например Azure Key Vault. В качестве бонуса WebApplicationBuilder по умолчанию настраивает файл appsettings.json с помощью reloadOnChange:true. Изначально стоит придерживаться настроек по умолчанию, очищать источники и начинать заново только в том случае, если вам действительно это необходимо.

ВНИМАНИЕ! Добавление источника конфигурации файла с помощью `reloadOnChange:true` не является совершенно бесплатным, поскольку ASP.NET Core настраивает инструмент наблюдения за файлами в фоновом режиме. Обычно эта ситуация не вызывает проблем, но если вы настраиваете конфигурацию, отслеживающую тысячи файлов, то можете столкнуться с трудностями!

В листинге 10.5 любые изменения, вносимые в файл, будут отражены в `IConfiguration`. Но, как я сказал в начале этой главы, `IConfiguration` не является предпочтительным способом передачи настроек в приложении. Как вы увидите в следующем разделе, следует отдавать предпочтение строго типизированным объектам.

10.3 Использование строго типизированных настроек с паттерном «Параметры»

В этом разделе вы узнаете о строго типизированной конфигурации и паттерне «Параметры». Это предпочтительный способ доступа к конфигурации в ASP.NET Core. Используя строго типизированную конфигурацию, можно избежать проблем с опечатками при доступе к ней, что также упрощает тестирование классов, поскольку вы можете использовать простые объекты РОСО, вместо того чтобы полагаться на абстракцию `IConfiguration`.

Большинство примеров, которые я показывал до сих пор, касались *передачи* значений в `IConfiguration`, а не того, как их *использовать*. Вы уже видели, что можно получить доступ к ключу, используя синтаксис словаря `builder.Configuration["key"]`, но применение таких строковых ключей кажется беспорядочным и может привести к появлению опечаток. Вместо этого ASP.NET Core предлагает использовать строго типизированные настройки. Это объекты РОСО, которые вы определяете и создаете. Они представляют собой небольшой набор настроек, ограниченных одной функцией в приложении.

В следующем листинге показаны настройки для компонента локатора магазина и настройки отображения, чтобы настроить домашнюю страницу приложения. Они разделены на два разных объекта с ключами `"MapSettings"` и `"AppDisplaySettings"`, соответствующими различным областям приложения, на которые они влияют.

Листинг 10.6. Разделение настроек на разные объекты в файле `appsettings.json`

```
{
  "MapSettings": {
    "DefaultZoomLevel": 6,
    "DefaultLocation": {
      "latitude": 50.500,
      "longitude": -4.000
    }
  },
  "AppDisplaySettings": {
    "Title": "ASP.NET Core Application"
  }
}
```

Настройки, связанные с разделом поиска магазинов в приложении

```

    "AppDisplaySettings": {
        "Title": "Acme Store Locator",
        "ShowCopyright": true
    }
}

```

Общие настройки, связанные с отображением приложения

Самый простой способ сделать настройки домашней страницы доступными в обработчике конечной точки – внедрить `IConfiguration` в обработчик конечной точки и получить доступ к значениям, используя синтаксис словаря:

```

app.MapGet("/display-settings", ( IConfiguration config ) =>
{
    string title = config[ "AppDisplaySettings:Title" ];
    bool showCopyright = bool.Parse(
        config[ "AppDisplaySettings:ShowCopyright" ]);
    return new { title, showCopyright };
});

```

Но это не лучший способ, на мой взгляд, слишком много строк! А этот `bool.Parse?` Фу! Вместо этого можно использовать специальные строго типизированные объекты со всей присущей им безопасностью типов и достоинствами IntelliSense, как показано в следующем листинге:

Листинг 10.7. Внедрение строго типизированных параметров в обработчик с помощью `IOptions<T>`

Свойство `Value` предстает объект настроек POCO

```

app.MapGet("/display-settings",
    ( IOptions<AppDisplaySettings> options ) => {
    AppDisplaySettings settings = options.Value;
    string title = settings.Title;
    bool showCopyright = settings.ShowCopyright;

    return new { title, showCopyright };
});

```

Объект настроек содержит свойства, которые привязаны к значениям конфигурации во время выполнения

Вы можете внедрить строго типизированный класс параметров, используя интерфейс обертки `IOptions<>`

Средство связывания также может преобразовывать строковые значения непосредственно во встроенные типы

Система конфигурации ASP.NET Core включает *связыватель*, который может принимать коллекцию значений конфигурации и привязывать их к строго типизированному объекту, классу `options`. Эта привязка аналогична концепции десериализации JSON для создания типов из главы 6 и привязке модели, используемой Model-View-Controller (MVC) и Razor Pages, о которой вы узнаете в третьей части.

В следующем разделе показано, как настроить привязку значений конфигурации к классу `options`, а в разделе 10.3.2 показано, как убедиться, что он перезагружается, когда меняются базовые значения конфигурации. В разделе 10.3.3 мы также рассмотрим различные типы объектов, которые можно привязать.

10.3.1 Знакомство с интерфейсом `IOptions`

В ASP.NET Core были введены строго типизированные настройки, позволяющие коду конфигурации придерживаться принципа единой ответственности и разрешать внедрение классов конфигурации в виде явных зависимостей. Такие настройки также упрощают тестирование; вместо того чтобы создавать экземпляр `IConfiguration` для тестирования службы, можно создать экземпляр класса параметров POCO.

Например, `AppDisplaySettings`, показанный в предыдущем примере, может быть простым классом, предоставляя только значения, относящиеся к домашней странице:

```
public class AppDisplaySettings
{
    public string Title { get; set; }
    public bool ShowCopyright { get; set; }
}
```

Ваши классы параметров не должны быть абстрактными и должны иметь конструктор `public` без параметров, чтобы привязка работала корректно. Связыватель установит все открытые свойства, совпадающие со значениями конфигурации, как вы вскоре увидите.

СОВЕТ Вы не ограничены примитивными типами, такими как `string` и `bool`; также можно использовать вложенные сложные типы. Система параметров привязывает секции к сложным свойствам. См. исходный код для этой книги в качестве примера.

Чтобы упростить привязку значений конфигурации к специальным классам параметров POCO, ASP.NET Core представляет интерфейс `IOptions<T>`. Это простой интерфейс с одним свойством `Value`, которое содержит сконфигурированный класс POCO во время выполнения. Классы параметров настраиваются как службы в файле `Program.cs`, как показано в следующем листинге.

Листинг 10.8 Конфигурирование классов параметров с помощью `Configure<T>` в файле `Startup.cs`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<MapSettings>(
    builder.Configuration.GetSection("MapSettings")); ←
builder.Services.Configure<AppDisplaySettings>(
    builder.Configuration.GetSection("AppDisplaySettings"));

Привязывает секцию AppDisplaySettings
к классу параметров POCO
AppDisplaySettings
```

Привязывает секцию MapSettings к классу параметров POCO MapSettings

СОВЕТ Необязательно использовать одно и то же имя для секции и класса, как это делаю я в листинге 10.8; это просто условное соглашение, которому я люблю следовать. С этим соглашением

можно использовать оператор `nameof()`, чтобы снизить вероятность опечаток, например вызвав `GetSection(nameof(MapSettings))`.

Каждый вызов `Configure<T>` под капотом выполняет следующие действия:

- 1 создает экземпляр `ConfigureOptions<T>`, который указывает, что `IOptions<T>` должен быть настроен на основе конфигурации. Если `Configure<T>` вызывается несколько раз, будут использоваться несколько объектов `ConfigureOptions<T>`. Все они могут быть применены для создания конечного объекта, почти так же, как `IConfiguration` создается из нескольких слоев;
- 2 каждый экземпляр `ConfigureOptions<T>` привязывает секцию `IConfiguration` к экземпляру класса POCO, `T`. Он задает все открытые свойства класса параметров на основе ключей в предоставленном `ConfigurationSection`. Помните, что имя секции ("MapSettings" из листинга 10.8) может иметь любое значение; оно не обязательно должно совпадать с названием вашего класса параметров;
- 3 интерфейс `IOptions<T>` регистрируется в контейнере внедрения зависимостей как синглтон, с последним привязанным объектом POCO в свойстве `Value`.

Последний шаг позволяет внедрять классы параметров в контроллеры и сервисы путем внедрения `IOptions<T>`, как вы уже видели. Это дает вам инкапсулированный, строго типизированный доступ к значениям конфигурации. Больше никаких «магических» строк!

ВНИМАНИЕ! Если вы забыли вызвать `Configure<T>` и внедрить `IOptions<T>` в свои сервисы, это не вызовет ошибку, но класс параметров `T` не будет ни к чему привязан, и в его свойствах будут только значения по умолчанию.

Привязка класса `T` к `ConfigurationSection` происходит, когда вы впервые запрашиваете `IOptions<T>`. Объект регистрируется в контейнере внедрения зависимостей как синглтон, поэтому привязывается только один раз.

В этой настройке есть один недостаток: нельзя использовать параметр `reloadOnChange`, который я описывал в разделе 10.2.4, чтобы перезагрузить строго типизированные классы параметров при использовании `IOptions<T>`. `IConfiguration` все равно будет перезагружен, если вы отредактируете свои файлы `appsettings.json`, но это не распространяется на класс параметров.

Если это покажется вам шагом назад или даже каким-то неприятным фактором, не волнуйтесь. У `IOptions<T>` для таких случаев есть двоюродный брат `IOptionsSnapshot<T>`.

10.3.2 Перезагрузка строго типизированных параметров с помощью `IOptionsSnapshot`

В предыдущем разделе мы использовали `IOptions<T>` для обеспечения строго типизированного доступа к конфигурации. Это дало прекрасную инкапсуляцию настроек для конкретного сервиса, но с одним не-

достатком: класс параметров никогда не меняется, даже если вы измените базовый файл конфигурации, из которого он был загружен, например `appsettings.json`.

Часто это не проблема (в любом случае не следует изменять файлы на рабочих серверах в реальной ситуации), но если вам нужна такая функциональность, то можно использовать интерфейс `IOptionsSnapshot<T>`. Концептуально `IOptionsSnapshot<T>` идентичен `IOptions<T>` в том, что это строго типизированное представление секции конфигурации. Разница состоит в том, когда и как часто при использовании каждого из них создаются объекты параметров POCO:

- `IOptions<T>` – экземпляр создается один раз при первом использовании. Он всегда содержит конфигурацию с момента создания экземпляра объекта;
- `IOptionsSnapshot<T>` – при необходимости создается новый экземпляр, если с момента создания последнего экземпляра базовая конфигурация изменилась.

ВНИМАНИЕ! `IOptionsSnapshot<T>` зарегистрирован как сервис с жизненным циклом `scoped`, поэтому нельзя внедрить его в сервисы с жизненным циклом `singleton`; в этом случае у вас возникнет связанная зависимость, как обсуждалось в главе 9. Если вам нужна одноэлементная версия `IOptionsSnapshot<T>`, можно использовать аналогичный интерфейс `IOptionsMonitor<T>`. Подробности см. в этом посте в блоге: <http://mng.bz/9Da7>.

`IOptionsSnapshot<T>` автоматически настраивается для классов опций в то же самое время, что и `IOptions<T>`, поэтому его можно использовать в своих сервисах точно так же. В следующем листинге показано, как обновить API настроек отображения, чтобы всегда получать последние значения конфигурации в строго типизированном классе параметров `AppDisplaySettings`.

Листинг 10.9. Внедрение перезагружаемых параметров с помощью `IOptionsSnapshot<T>`

```
app.MapGet("/display-settings",
    (IOptionsSnapshot<AppDisplaySettings> options) => <-- IOptionsSnapshot<T>
    {
        AppDisplaySettings settings = options.Value; <-- обновляется автоматически, если соответствующие значения конфигурации изменяются
        return new
        {
            title = settings.Title,
            showCopyright = settings.ShowCopyright, <-- Свойство Value предоставляет объект настроек POCO, такой же, как для IOptions<T>
        };
    });

```

Настройки соответствуют значениям конфигурации на текущий момент времени, а не на момент запуска

Поскольку `IOptionsSnapshot<AppDisplaySettings>` регистрируется как сервис с жизненным циклом `scoped`, он воссоздается при каждом запросе. Если вы отредактируете файл настроек и вызовете перезагрузку `IConfiguration`, `IOptionsSnapshot<AppDisplaySettings>` покажет новые значения при следующем запросе. Новый объект `AppDisplaySettings` создается с новыми значениями конфигурации и используется для всех будущих внедрений зависимостей – до тех пор, пока вы снова не отредактируете файл!

Автоматическая перезагрузка настроек очень проста: обновите свой код, чтобы использовать `IOptionsSnapshot<T>` вместо `IOptions<T>` везде, где это нужно. Но имейте в виду, что это изменение не является бесплатным. Вы выполняете повторную привязку и конфигурирование своего объекта параметров при каждом запросе, а это может повлиять на производительность. На практике перезагрузка настроек не распространена в промышленном окружении, поэтому вы можете решить, что удобство для разработчика не стоит снижения производительности.

Важным моментом при использовании паттерна параметров является дизайн самих классов опций РОСО. Обычно это простые наборы свойств, но нужно помнить несколько вещей, чтобы не застрять во время отладки, выясняя, почему привязка не сработала.

10.3.3 Разработка классов параметров для автоматической привязки

Я уже коснулся некоторых требований к классам РОСО для работы со связывателем `IOptions<T>`, но следует помнить о нескольких правилах. Первый ключевой момент заключается в том, что связыватель будет создавать экземпляры ваших классов параметров с использованием отражения, поэтому эти классы должны:

- быть неабстрактными;
- иметь конструктор по умолчанию (без параметров).

Если ваши классы удовлетворяют этим двум пунктам, связыватель будет перебирать все свойства класса и выполнит привязку любого свойства, какого сможет. В самом широком смысле связыватель может привязывать любое свойство:

- которое является открытым;
- у которого есть метод получения (`get`-метод) – связыватель не будет записывать свойства, у которых есть только метод установки;
- у которого есть метод установки (`set`-метод) или, для сложных типов, ненулевое значение;
- которое не является индексатором.

В следующем листинге показан обширный класс параметров со множеством различных типов свойства. Все свойства `BindableOptions` допустимы для привязки, а все свойства `UnbindableOptions` – нет.

Листинг 10.10 Класс параметров, содержащий свойства, пригодные и непригодные для привязки

```

public class BindableOptions
{
    public string String { get; set; }
    public int Integer { get; set; }
    public SubClass Object { get; set; }
    public SubClass ReadOnly { get; } = new SubClass();
    public Dictionary<string, SubClass> Dictionary { get; set; }
    public List<SubClass> List { get; set; }
    public IDictionary<string, SubClass> IDictionary { get; set; }
    public IEnumerable<SubClass> IEnumerable { get; set; }
    public ICollection<SubClass> ReadOnlyCollection { get; }
        = new List<SubClass>();
    public class SubClass
    {
        public string Value { get; set; }
    }
}

public class UnbindableOptions
{
    internal string NotPublic { get; set; }
    public SubClass SetOnly { set => _setOnly = value; }
    public SubClass NullReadOnly { get; } = null;
    public SubClass NullPrivateSetter { get; private set; } = null;
    public SubClass this[int i] {
        get => _indexerList[i];
        set => _indexerList[i] = value;
    }
    public List<SubClass> NullList { get; }
    public Dictionary<int, SubClass> IntegerKeys { get; set; }
    public IEnumerable<SubClass> ReadOnlyEnumerable { get; }
        = new List<SubClass>();
    public SubClass _setOnly = null;
    private readonly List<SubClass> _indexerList
        = new List<SubClass>();
    public class SubClass
    {
        public string Value { get; set; }
    }
}

```

Связыватель может привязывать простые и сложные типы объектов, а также свойства, доступные только для чтения, со значением по умолчанию

Связыватель также привяжет коллекции, включая интерфейсы

Связыватель не может привязать закрытые свойства, у которых есть только метод установки или значение null, или свойства-индексаторы

Эти свойства коллекции нельзя привязать

Резервные поля для реализации свойств SetOnly и Indexer не привязаны напрямую

Как показано в листинге, связыватель обычно поддерживает коллекции – реализаций и интерфейсы. Если свойство коллекции уже инициализировано, оно будет использоваться, но связыватель также может создавать экземпляр коллекции автоматически. Если ваше свойство реализует любой из следующих интерфейсов, связыватель создает `List<>` соответствующего типа в качестве резервного объекта:

- `IReadOnlyList<>;`
- `IReadOnlyCollection<>;`
- `ICollection<>;`
- `IEnumerable<>.`

ВНИМАНИЕ! Нельзя выполнить привязку к свойству `IEnumerable<>`, которое уже было инициализировано, поскольку интерфейс не предоставляет функцию `Add` и связыватель не заменит резервное значение. Можно выполнить привязку к `IEnumerable<>`, если вы оставите его начальное значение равным `null`.

Точно так же связыватель создаст `Dictionary<,>` в качестве резервного поля для свойств со словарными интерфейсами, если они используют ключи `string`, `enum` или `integer` (`int`, `short`, `byte` и т. д.):

- `IDictionary<,>;`
- `IReadOnlyDictionary<,>.`

ВНИМАНИЕ! Нельзя привязывать словари к ключам, не являющимся строковыми или целочисленными, например собственные классы или тип `double`. Чтобы увидеть примеры привязки типов коллекций, см. исходный код для этой книги.

Понятно, что здесь есть немало нюансов, но если придерживаться простых случаев из предыдущего примера, то все будет в порядке. Обязательно проверьте свои файлы JSON на наличие опечаток! Так же можно рассмотреть возможность использования явной валидации параметров, как описано в этом посте: <http://mng.bz/jPjr>.

СОВЕТ Паттерн «Параметры» чаще всего используется для привязки классов РОСО к конфигурации, но вы также можете настроить свои строго типизированные классы настроек в коде путем предоставления лямбда-функции функции `Configure`. Например, `services.Configure<TestOptions>(opt => opt.Value=true)`.

Он широко используется в ASP.NET Core, но не все являются его поклонниками. В следующем разделе вы увидите, как использовать строго типизированные настройки и связыватель конфигурации без этого паттерна.

10.3.4 Привязка строго типизированных параметров без интерфейса `IOptions`

Интерфейс `IOptions` во многом каноничен в ASP.NET Core – он используется основными библиотеками ASP.NET Core и имеет различные удобные функции для привязки строго типизированных настроек, как вы уже видели. Однако во многих случаях он дает мало преимуществ *потребителям* строго типизированных объектов настроек. Сервисы должны зависеть от интерфейса `IOptions`, но затем немедленно извлекать «настоящий» объект, вызывая `IOptions<T>.Value`. Это может быть особенно неприятно, если вы создаете многократно используемую

библиотеку, которая по своей сути не связана с ASP.NET Core, так как вы должны предоставить интерфейс `IOptions<T>` во всех открытых API.

К счастью, связыватель конфигурации, отображающий объекты `IConfiguration` в строго типизированные объекты настроек, по своей сути не привязан к `IOptions`. В листинге 10.11 показано, как вручную привязать строго типизированный объект настроек к секции конфигурации, зарегистрировать его в контейнере внедрения зависимостей и внедрить объект `MapSettings` непосредственно в обработчик или сервис без дополнительных церемоний, необходимых для использования `IOptions<MapSettings>`.

Листинг 10.11. Настройка строго типизированных параметров без интерфейса `IOptions` в файле Program.cs

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
    var settings = new MapSettings(); <-- Создает новый экземпляр
    builder.Configuration.GetSection("MapSettings").Bind(settings); <-- объекта MapSettings
    builder.Services.AddSingleton(settings); <-- Привязывает раздел
    WebApplication app = builder.Build(); <-- MapSettings в IConfiguration
    app.MapGet("/", (MapSettings mapSettings) => mapSettings); <-- к объекту настроек
    app.Run(); <-- Внедряет объект
                    <-- MapSettings
```

Регистрирует
объект
настроек
как синглтон

В качестве альтернативы можно зарегистрировать тип `IOptions` в контейнере внедрения зависимостей, а затем использовать лямбду-функцию для дополнительной регистрации `MapSettings` в качестве синглтона, чтобы его можно было внедрить напрямую, как показано в листинге 10.12.

Листинг 10.12. Настройка строго типизированных параметров для прямого внедрения

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
    builder.Services.Configure<MapSettings>(<-- Настраивает IOptions как
        builder.Configuration.GetSection("MapSettings"));
    builder.Services.AddSingleton(provider =>
        provider.GetRequiredService<IOptions<MapSettings>>().Value); <-- Регистрирует объект MapSettings
    WebApplication app = builder.Build(); <-- в контейнере внедрения зависимостей, делегируя регистрацию IOptions
    app.MapGet("/", (MapSettings mapSettings) => mapSettings); <-- Внедряет объект
    app.Run(); <-- MapSettings
```

Если вы используете любой из этих подходов, вы не получите преимущества от возможности перезагрузки строго типизированных настроек без дальнейшей работы или от некоторых более продвинутых способов использования `IOptions`, но в большинстве случаев это не такая уж и большая проблема. В целом мне нравится данный подход, но, как и всегда, подумайте о том, что вы теряете, прежде чем использовать его.

СОВЕТ В главе 31 я показываю один такой сценарий, в котором мы настраиваем объект `IOptions`, используя сервисы в контейнере внедрения зависимостей. Чтобы увидеть другие продвинутые сценарии, см. <http://mng.bz/DR7y>, или просмотрите посты, посвященные `IOptions` в моем блоге, например этот: <http://mng.bz/11Aj>.

На этом мы подошли к концу раздела, посвященного строго типизированным настройкам. В следующем разделе мы рассмотрим, как динамически изменять настройки во время выполнения на основе окружения, в котором работает ваше приложение.

10.4 Настройка приложения для нескольких окружений

В этом разделе вы узнаете об окружениях размещения в ASP.NET Core. Вы увидите, как определить, в каком окружении работает приложение и как изменять используемые значения конфигурации в зависимости от него. Например, это позволяет легко переключаться между различными наборами значений конфигурации в промышленном окружении по сравнению с окружением разработки.

Любое приложение, попадающее в промышленное окружение, скорее всего, будет работать в нескольких окружениях. Например, если вы создаете приложение с доступом к базе данных, то, возможно, на вашем компьютере работает небольшая база данных, которую вы используете при разработке. В промышленном окружении на сервере будет работать совершенно другая база данных.

Еще одно распространенное требование – разный уровень журналирования в зависимости от того, где запущено приложение. В окружении разработки можно генерировать большое количество записей журнала, так как это помогает при отладке, но как только вы перейдете в промышленное окружение, такое количество может оказаться неподъемным. Вам нужно будет регистрировать предупреждения и ошибки, а также, возможно, записи на уровне информации, но определенно не записи журнала уровня отладки!

Чтобы справиться с этими требованиями, нужно убедиться, что ваше приложение загружает разные значения конфигурации в зависимости от окружения, в котором оно работает: загружает строку подключения к рабочей базе данных в промышленном окружении и т. д. Нужно учитывать три аспекта:

- как приложение определяет, в каком окружении оно работает;
- как вы загружаете различные значения конфигурации в зависимости от текущего окружения;
- как изменить окружение для конкретной машины.

В данном разделе по очереди рассматривается каждый из этих вопросов, чтобы вы легко могли отличить свою машину, используемую в окружении разработки, от рабочих серверов и действовать соответствующим образом.

10.4.1 Определение окружения размещения

Когда вы создаете экземпляр `WebApplicationBuilder` в файле `Program.cs`, он автоматически настраивает окружение размещения для вашего приложения. По умолчанию `WebApplicationBuilder` использует, что, наверное, неудивительно, переменную окружения для определения текущего окружения. `WebApplicationBuilder` ищет магическую переменную окружения `ASPNETCORE_ENVIRONMENT`, использует ее для создания объекта `IHostEnvironment` и предоставляет ее как `WebApplicationBuilder.Environment`.

ПРИМЕЧАНИЕ Можно использовать либо переменную окружения `DOTNET_ENVIRONMENT`, либо `ASPNETCORE_ENVIRONMENT`. Значение `ASPNETCORE_` переопределяет значение `DOTNET_`, если заданы оба. В этой книге я использую версию с `ASPNETCORE_`.

СОВЕТ Интерфейс `IHostEnvironment` предоставляет ряд полезных свойств, касающихся контекста запуска приложения. Некоторые из них вы уже видели, например `ContentRootPath`, которое сообщает приложению, в какой папке можно найти файлы конфигурации, например файл `appsettings.json`. Обычно в этой папке выполняется приложение.

В этом разделе нас интересует свойство `IHostEnvironment.EnvironmentName`. Ему задано значение переменной окружения `ASPNETCORE_ENVIRONMENT`, поэтому это может быть что угодно, но в большинстве случаев нужно придерживаться трех часто используемых значений:

- "Development";
- "Staging";
- "Production".

ASP.NET Core включает в себя несколько вспомогательных методов для работы с этими тремя значениями, поэтому будет легче, если вы будете их придерживаться. В частности, когда вы тестируете, работает ли ваше приложение в определенном окружении, нужно использовать один из следующих методов расширения:

- `IHostEnvironment.IsDevelopment();`
- `IHostEnvironment.IsStaging();`
- `IHostEnvironment.IsProduction();`
- `IHostEnvironment.IsEnvironment(string environmentName).`

Все эти методы проверяют переменную окружения без учета регистра, поэтому вы не получите никаких трудно воспроизводимых ошибок во время выполнения, если не используете прописные буквы для записи значения переменной окружения.

СОВЕТ По возможности используйте с `EnvironmentValue` методы расширения `IHostEnvironment` вместо прямого сравнения строк, поскольку они проверяют совпадение без учета регистра.

IHostEnvironment не делает ничего, кроме предоставления сведений о текущем окружении, но его можно использовать по-разному. В главе 4 вы видели, что WebApplication добавляет компонент DeveloperExceptionMiddleware в конвейер промежуточного ПО только в окружении разработки. Теперь вы знаете, откуда он получал информацию об окружении, – IHostEnvironment.

Можно использовать аналогичный подход для настройки загружаемых значений конфигурации во время выполнения, загружая разные файлы в окружении разработки или промышленном окружении. Это распространенный подход. Он включен в большинство шаблонов ASP.NET Core и по умолчанию при использовании ConfigurationManager, включенного в WebApplicationBuilder.

10.4.2 Загрузка файлов конфигурации для конкретного окружения

Значение EnvironmentName определяется на ранней стадии процесса начальной загрузки приложения, до того, как ConfigurationManager по умолчанию будет полностью заполнен WebApplicationBuilder. В результате вы можете динамически изменять то, какие поставщики конфигурации добавляются в построитель, а следовательно, какие значения конфигурации загружаются, при создании IConfiguration.

Распространенный шаблон – это наличие необязательного файла appsettings.ENVIRONMENT.json для конкретного окружения, который загружается после файла appsettings.json по умолчанию. В данном листинге показано, как этого добиться, если вы настраиваете метод ConfigurationManager в файле Program.cs, но это также то, что WebApplicationBuilder делает по умолчанию.

Листинг 10.13 Добавление файлов appsettings.json для конкретного окружения

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

IHostEnvironment env = builder.Environment; <-- Текущее окружение
builder.Configuration.Sources.Clear();
builder.Configuration
    .AddJsonFile(
        "appsettings.json",
        optional: false) <-- IHostEnvironment доступно
                                для WebApplicationBuilder
    .AddJsonFile(
        $"appsettings.{env.EnvironmentName}.json",
        Optional: true);
WebApplication app = builder.Build();

app.MapGet("/", () =>"Hello world!");

app.Run();
```

Базовый файл
appsettings.json обычно
делается обязательным

Добавляет необязатель-
ный файл в формате
JSON для конкретного
окружения, имя которого
зависит от окружения

Здесь глобальный файл appsettings.json содержит настройки, применимые к большинству окружений. Дополнительные необязательные файлы, appsettings.Development.json, appsettings.Staging.json и appsettings.Production.json, впоследствии добавляются в ConfigurationManager, в зависимости от текущего EnvironmentName.

Любые настройки в этих файлах будут перезаписывать значения из глобального файла appsettings.json, если в них есть такой же ключ. Это позволяет настроить журналирование таким образом, чтобы регистрировать больше информации, чем в обычном режиме, только в окружении разработки и переключаться на более избирательные логи в промышленном окружении.

Еще один распространенный шаблон – полное добавление или удаление поставщиков конфигурации в зависимости от окружения. Например, можно использовать поставщика User Secrets при локальной разработке, а Azure Key Vault – в промышленном окружении. В листинге 10.14 показано, как применять IHostEnvironment для условного включения поставщика User Secrets только в окружении разработки. Опять же, WebApplicationBuilder использует этот паттерн по умолчанию.

Листинг 10.14. Условное включение поставщика конфигурации User Secrets

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

IHostEnvironment env = builder.Environment;

builder.Configuration.Sources.Clear();
builder.Configuration
    .AddJsonFile(
        "appsettings.json",
        optional: false)
    .AddJsonFile(
        $"appsettings.{env}.json",
        Optional: true);
```

Методы расширения делают проверку окружения простой и явной

```
if(env.IsDevelopment()) {  
    builder.Configuration.AddUserSecrets<Program>();  
}  
  
WebApplication app = builder.Build();  
  
app.MapGet("/", () =>"Hello world!");  
  
app.Run();
```

В окружении для отладки и промышленном окружении поставщик User Secrets не будет использоваться

Как уже упоминалось, конвейер промежуточного программного обеспечения вашего приложения также часто настраивают в зависимости от среды. В главе 4 вы узнали, что WebApplication добавляет DeveloperExceptionPageMiddleware условно при локальной разработке.

В следующем листинге показано, как можно использовать `IHostEnvironment` для управления конвейером таким образом, чтобы при обкатке или в рабочем окружении ваше приложение вместо этого использовало `ExceptionHandlerMiddleware`.

Листинг 10.15. Использование окружения размещения для настройки конвейера промежуточного программного обеспечения

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.AddProblemDetails();           ← Добавляет службу Problem Details
WebApplication app = builder.Build();  в контейнер внедрения зависи-
                                         симостей для использования
                                         ExceptionHandlerMiddleware

if (!builder.Environment.IsDevelopment())
{
    app.UseExceptionHandler();          ← Когда конвейер не на-
                                         находится в окружении раз-
                                         работки, он использует
                                         ExceptionHandlerMiddleware

}

app.MapGet("/", () =>"Hello world!");

app.Run();
```

ПРИМЕЧАНИЕ В листинге 10.15 мы добавили сервисы `Problem Details` в контейнер внедрения зависимостей, чтобы `ExceptionHandlerMiddleware` мог автоматически генерировать ответ `Problem Details`. Поскольку мы добавляем дополнительное промежуточное ПО только в окружении для обкатки и промышленном окружении, вы также можете условно добавлять сервисы в контейнер, вместо того чтобы всегда добавлять их, как мы это делали здесь.

Можно внедрить `IHostEnvironment` в любое место своего приложения, но я бы не советовал использовать его в собственных сервисах за пределами файла `Program.cs`. Намного лучше использовать поставщика конфигурации для настройки строго типизированных параметров на основе текущего окружения размещения и внедрять эти настройки в свое приложение.

Как бы это ни было полезно, но, когда вы задаете `IHostEnvironment` с переменной окружения, это может выглядеть немного громоздко, если вы хотите переключаться между разными окружениями во время тестирования. Лично я всегда забываю, как задавать переменные окружения в различных операционных системах, которые использую. Последний навык, которому я хотел бы научить вас, – как задавать окружение размещения при локальной разработке.

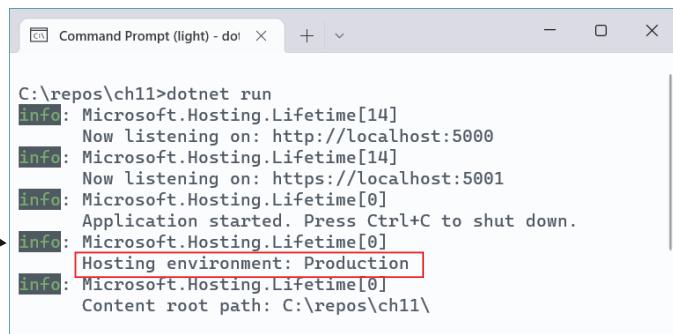
10.4.3 Задаем окружение размещения

В этом разделе я покажу вам несколько способов, с помощью которых можно задать окружение размещения во время разработки. Это упро-

щает тестирование поведения конкретного приложения в разных окружениях без необходимости менять окружение для всех приложений на компьютере.

Если ваше приложение ASP.NET Core не может найти переменную окружения `ASPNETCORE_ENVIRONMENT` при запуске, то по умолчанию используется промышленное окружение, как показано на рис. 10.6. Это означает, что при развертывании в промышленном окружении вы будете использовать правильный вариант.

Если `WebApplicationBuilder` не может найти переменную `ASPNETCORE_ENVIRONMENT` во время выполнения, по умолчанию используется значение `Production`



```
C:\repos\ch11>dotnet run
[Info]: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
[Info]: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5001
[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:\repos\ch11\
```

Рис. 10.6 По умолчанию приложения ASP.NET Core запускаются в промышленном окружении. Можно переопределить это, задав переменную `ASPNETCORE_ENVIRONMENT`

СОВЕТ По умолчанию текущее окружение размещения отображается в консоли при запуске. Это может быть полезно, чтобы быстро удостовериться в том, что переменная окружения подобрана правильно.

Еще один вариант – использовать файл `launchSettings.json` для управления окружением. Все приложения ASP.NET Core по умолчанию включают этот файл в папку `Properties`. Файл `launchSettings.json` определяет профили для запуска вашего приложения.

СОВЕТ *Профили* можно использовать для запуска приложения с другими переменными окружения, а также для имитации запуска в Windows за IIS с помощью профиля IIS Express. Лишь я редко пользуюсь этим профилем, даже для Windows, и всегда выбираю профиль `http` или `https`.

Типичный файл `launchSettings.json` показан в следующем листинге. В нем определены три профиля: `http`, `https` и `IIS Express`. Первые два профиля эквивалентны использованию команды `dotnet run` для запуска проекта. Профиль `http` прослушивает только запросы `http://`, тогда как `https` прослушивает и запросы `http://`, и запросы `https://`. Профиль `IIS Express` можно использовать только в Windows, и он использует IIS Express для запуска приложения.

Листинг 10.16. Типичный файл launchSettings.json, определяющий три профиля

```
{
    "iisSettings": {
        "windowsAuthentication": false,
        "anonymousAuthentication": true,
        "iisExpress": {
            "applicationUrl": "http://localhost:53846",
            "sslPort": 44399
        }
    },
    "profiles": {
        "http": {
            "commandName": "Project",
            "dotnetRunMessages": true,
            "launchBrowser": true,
            "applicationUrl": "http://localhost:5063",
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        },
        "https": {
            "commandName": "Project",
            "dotnetRunMessages": true,
            "launchBrowser": true,
            "applicationUrl": "https://localhost:7202;http://localhost:5063",
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        },
        "IIS Express": {
            "commandName": "IISExpress",
            "launchBrowser": true,
            "environmentVariables": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        }
    }
}
```

Профиль «http» используется по умолчанию в macOS

Если true, браузер запускается при запуске приложения

Запускает приложение через IIS Express (только для Windows)

Определяет настройки для работы за IIS или использования профиля IIS Express

Значение «истина» приводит к показу логов сборки и/или восстановления, когда dotnet run выполняет соответствующую задачу

Команда «проект» эквивалента вызову dotnet run для проекта

Определяет URL-адреса, которые приложение будет прослушивать в этом профиле

Определяет пользовательские переменные среды для профиля и устанавливает среду разработки

Профиль https используется по умолчанию в Visual Studio в Windows

Профиль https прослушивает URL-адреса http:// и https://

Каждый профиль может иметь разные переменные среды

Преимущество локального использования файла launchSettings.json заключается в том, что он позволяет задавать «локальные» переменные окружения проекта. Например, в листинге 10.16 окружение задано как окружение разработки. Это позволяет использовать разные переменные окружения для каждого проекта и даже для каждого профиля и сохранять их в системе управления версиями.

Вы можете выбрать профиль в Visual Studio, используя раскрывающийся список рядом с кнопкой **Debug** на панели инструментов, как показано на рис. 10.7. Можно выбрать профиль для запуска из

командной строки, используя команду `dotnet run --launch-profile <Profile Name>`. Если вы не укажете профиль, то будет использован первый профиль, перечисленный в файле `launchSettings.json`. Если вы не хотите использовать какой-либо профиль, то должны явно игнорировать файл `launchSettings.json` с помощью команды `dotnet run --no-launch-profile`.

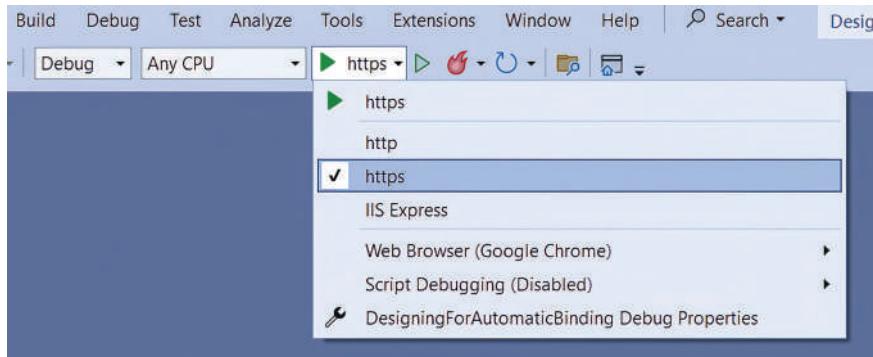


Рис. 10.7 Можно выбрать профиль для использования в Visual Studio из раскрывающегося списка, щелкнув по кнопке **Debug**. По умолчанию Visual Studio использует профиль `https`

Если вы используете Visual Studio, то вы можете редактировать файл `launchSettings.json` прямо из интерфейса: дважды щелкните на **Properties**, выберите вкладку **Debug** и выберите Open debug launch profiles UI. На рис. 10.8 можно увидеть, что переменная окружения `ASPNETCORE_ENVIRONMENT` установлена в `Development`. Любые изменения в этом окне синхронизируются с файлом `launchSettings.json`.

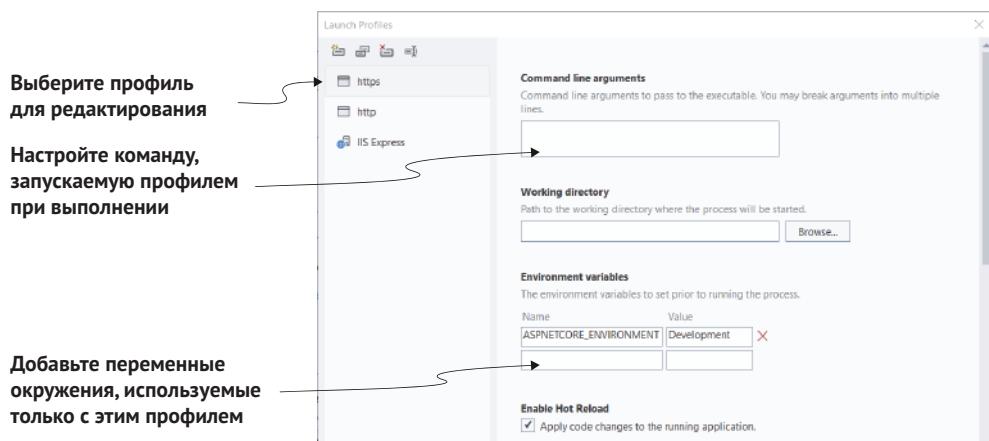


Рис. 10.8 Вы можете использовать Visual Studio для редактирования файла `launchSettings.json`. Изменения в окне **Properties** будут синхронизированы с файлом `launchSettings.json`

Файл `launchSettings.json` предназначен только для локальной разработки; по умолчанию он не развертывается на рабочих серверах. Хотя это можно сделать, но обычно оно того не стоит. Переменные окружения – более подходящий вариант.

Еще один последний прием, который я использовал, – это аргументы командной строки. Например, можно настроить окружение для обкатки таким образом:

```
dotnet run --no-launch-profile --environment Staging
```

Обратите внимание, что вам также необходимо передать `--no-launch-profile` при наличии файла `launchSettings.json`; в противном случае приоритет имеют значения в файле.

На этом мы подошли к концу главы, посвященной конфигурации. Это не особо увлекательная тема, но конфигурация является неотъемлемой частью всех приложений. Модель поставщика конфигурации ASP.NET Core обрабатывает широкий спектр сценариев, позволяя хранить параметры и секреты в различных местах.

Простые настройки можно сохранить в файле `appsettings.json`, где их легко изменить во время разработки, и их можно перезаписать с помощью файлов JSON для конкретного окружения. Между тем секреты и конфиденциальные настройки могут храниться за пределами файла проекта в диспетчере User Secrets или как переменные окружения, что дает вам и гибкость, и безопасность одновременно – пока вы не записываете свои секреты в файл `appsettings.json`!

В главе 11 мы рассмотрим спецификацию OpenAPI и как можно использовать ее для документирования своих API, тестирования конечных точек и создания строго типизированных клиентов.

Резюме

- Все, что можно считать настройкой или секретом, обычно хранится как значение конфигурации. Вынос этих значений за пределы приложения означает, что вы можете изменять их без повторной компиляции приложения;
- ASP.NET Core использует поставщиков конфигурации для загрузки пар «ключ–значение» из различных источников. Приложения могут использовать множество разных поставщиков;
- можно добавить поставщиков конфигурации в экземпляр `ConfigurationManager`, используя такие методы расширения, как `AddJsonFile()`;
- важен порядок, в котором вы добавляете поставщиков в `ConfigurationManager`; поставщики, добавленные позже, заменяют значения настроек, определенных в предыдущих поставщиках, сохранив уникальные настройки;
- ASP.NET Core включает в себя, среди прочего, встроенные поставщики для файлов JSON, XML, файлов окружения и аргументов командной строки. Для множества других поставщиков,

таких как файлы YAML и Azure Key Vault, существуют готовые Nuget-пакеты;

- ConfigurationManager реализует интерфейс IConfiguration, а также IConfigurationBuilder, поэтому вы можете напрямую получать значения конфигурации из него;
- ключи конфигурации не чувствительны к регистру, поэтому необходимо позаботиться о том, чтобы не потерять значения при загрузке настроек из источников с учетом регистра, таких как YAML;
- можно получить настройки из IConfiguration напрямую, используя синтаксис индексатора, например Configuration["MySettings:Value"]. Такой метод часто полезен для доступа к значениям конфигурации в файле Program.cs;
- WebApplicationBuilder настраивает ConfigurationManager, используя файлы JSON, переменные окружения, аргументы командной строки и поставщиков User Secrets. Такое сочетание обеспечивает хранение в репозитории в файлах JSON, хранилище секретов как в окружении разработки, так и в промышленном окружении, а также возможность легко переопределять настройки во время выполнения;
- в промышленном окружении храните секреты в переменных окружения, чтобы уменьшить вероятность неправильного предоставления этих данных в репозитории кода. Их можно загрузить после файловых настроек в построителе конфигурации;
- на машинах, используемых для разработки, удобнее использовать User Secrets Manager, нежели переменные окружения. Он хранит секреты в профиле пользователя вашей ОС, за пределами папки проекта, уменьшая вероятность неправильного предоставления этих данных в репозитории кода;
- имейте в виду, что ни переменные окружения, ни инструмент User Secrets Manager не шифруют секреты, они просто хранят их в местах, которые с наименьшей вероятностью станут открытыми, поскольку находятся за пределами папки вашего проекта;
- файловые поставщики, такие как JSON-поставщик, могут автоматически перезагружать значения конфигурации при изменении файла. Это позволяет обновлять значения конфигурации в реальном времени, без перезапуска приложения;
- используйте строго типизированные классы параметров POCO для доступа к конфигурации в приложении. Использование строго типизированных параметров уменьшает связность в вашем приложении и гарантирует, что классы будут зависеть только от используемых ими значений конфигурации;
- используйте метод расширения Configure<T>() в ConfigureServices для привязки своих объектов параметров POCO к ConfigurationSection. В качестве альтернативы можно настроить объекты IOptions<T> в коде вместо использования значений конфигурации, передав лямбда-функцию методу Configuration();
- можно внедрить интерфейс IOptions<T> в свои сервисы с помощью внедрения зависимостей. Вы можете получить доступ к строго

типовизированному объекту параметров свойства `Value`. Значения `IOptions<T>` регистрируются в контейнере внедрения зависимостей как синглтоны, поэтому остаются неизменными, даже если базовая конфигурация изменяется;

- если вы хотите перезагрузить объекты параметров РОСО при изменении конфигурации, используйте интерфейс `IOptionsSnapshot<T>`. Эти экземпляры регистрируются в контейнере внедрения зависимостей с жизненным циклом `scoped`, поэтому создаются заново для каждого запроса. Использование интерфейса `IOptionsSnapshot<T>` влияет на производительность из-за неоднократной привязки к объекту параметров, поэтому используйте его только в том случае, если такие затраты приемлемы;
- приложениям, работающим в разных окружениях, окружении разработки и промышленном окружении, например, часто требуются разные значения конфигурации. ASP.NET Core определяет текущее окружение размещения с помощью переменной окружения `ASPNETCORE_ENVIRONMENT`. Если она не задана, предполагается, что это промышленное окружение;
- можно настроить окружения размещения локально с помощью файла `launchSettings.json`. Это позволяет привязать переменные окружения к конкретному проекту;
- текущее окружение размещения предоставляется как интерфейс `IHostEnvironment`. Вы можете проверить наличие конкретного окружения с помощью методов `IsDevelopment()`, `IsStaging()` и `IsProduction()`. Затем можно использовать объект `IHostEnvironment` для загрузки файлов, относящихся к текущему окружению, например `appsettings.Production.json`.

11

Документирование API с помощью OpenAPI

В этой главе:

- что такое OpenAPI и почему она полезна;
- добавление описания OpenAPI в приложение;
- улучшение описаний OpenAPI путем добавления метаданных в конечные точки;
- создание клиента C# на основе описания OpenAPI.

В этой главе я познакомлю вас со спецификацией OpenAPI для описания RESTful API, продемонстрирую, как использовать OpenAPI для описания минимального API, и мы обсудим причины, по которым вам это может понадобиться.

В разделе 11.1 вы узнаете о самой спецификации OpenAPI и о том, как она вписывается в приложение ASP.NET Core. Вы узнаете о библиотеках, которые можно использовать для активации создания документации OpenAPI в вашем приложении, и о том, как предоставить документ с помощью промежуточного ПО.

Получив документ OpenAPI, вы увидите, как сделать с ним что-нибудь полезное, в разделе 11.2, где мы добавим в приложение Swagger UI. Swagger UI использует его для создания пользовательского интерфейса для тестирования и проверки конечных точек приложения, что может быть особенно полезно для локального тестирования.

После просмотра описания приложения в Swagger UI время вернуться к коду из раздела 11.3. OpenAPI и Swagger UI нуждаются в обширных метаданных о ваших конечных точках, чтобы обеспечить лучшую функциональность, поэтому мы рассмотрим базовые метаданные, которые вы можете добавить в свои конечные точки.

В разделе 11.4 вы узнаете об одной из лучших функциональных возможностей, которая появляется при создании описания OpenAPI: автоматически создаваемых клиентах. Используя стороннюю библиотеку NSwag, вы узнаете, как автоматически генерировать код C# и классы для взаимодействия с API на основе описания OpenAPI, добавленного вами в предыдущих разделах. Вы узнаете, как генерировать клиента, настраивать сгенерированный код и выполнить повторную сборку клиента при изменении описания OpenAPI вашего приложения.

Наконец, в разделе 11.5 вы узнаете дополнительные способы добавления метаданных в конечные точки, чтобы обеспечить наилучшее взаимодействие с созданными вами клиентами. Вы узнаете, как добавлять краткую информацию и описания в конечные точки, используя вызовы методов и атрибуты, а также извлекая комментарии XML-документации из кода C#.

Прежде чем мы рассмотрим эти расширенные сценарии, мы разберем спецификацию OpenAPI, узнаем, что это такое, и изучим, как добавить документ OpenAPI в приложение.

11.1 Добавление описания OpenAPI в приложение

OpenAPI (ранее известная как Swagger) – это независимая от языка спецификация для описания RESTful API. По своей сути OpenAPI описывает схему документа в формате JSON, который, в свою очередь, описывает URL-адреса, доступные в приложении, способы их вызова и типы данных, которые они возвращают. В этом разделе вы узнаете, как создать документ OpenAPI для приложения с минимальным API.

Предоставление документа OpenAPI приложению позволяет добавлять в него различные типы автоматизации. Например, вы можете:

- исследовать API своего приложения с помощью Swagger UI (раздел 11.2);
- генерировать строго типизированных клиентов для взаимодействия с приложением (раздел 11.4);
- автоматически интегрироваться со сторонними службами, такими как Azure API Management.

ПРИМЕЧАНИЕ Если вы знакомы с SOAP еще со времен ASP.NET, то можете рассматривать OpenAPI как HTTP/REST-эквивалент языка описания Web-служб (WSDL). Точно так же, как файл с расширением .wsdl описывает службы XML SOAP, документ OpenAPI описывает ваш REST API.

ASP.NET Core изначально включает поддержку документов OpenAPI, но, чтобы воспользоваться ими, придется использовать стороннюю библиотеку. Две наиболее известные библиотеки – это NSwag

и Swashbuckle. В этой главе я использую Swashbuckle для добавления документа OpenAPI в приложение ASP.NET Core. Вы можете прочитать, как вместо нее использовать NSwag, на странице <http://mng.bz/6Dmy>.

ПРИМЕЧАНИЕ NSwag и Swashbuckle предоставляют схожие функции для создания документов OpenAPI, хотя вы обнаружите небольшие различия в том, как их использовать и в функциях, которые они поддерживают. NSwag также поддерживает генерацию клиентов, как это происходит, вы увидите в разделе 11.4.

Добавьте пакет NuGet Swashbuckle.AspNetCore в свой проект с помощью диспетчера пакетов NuGet в Visual Studio или используйте интерфейс командной строки .NET, выполнив команду

```
dotnet add package Swashbuckle.AspNetCore
```

из папки проекта. Swashbuckle использует службы метаданных ASP.NET Core для получения информации обо всех конечных точках вашего приложения и создания документа OpenAPI. Затем этот документ обслуживается промежуточным ПО, предоставляемым Swashbuckle, как показано на рис. 11.1. Swashbuckle также включает компонент для визуализации документа OpenAPI, как вы увидите в разделе 11.2.

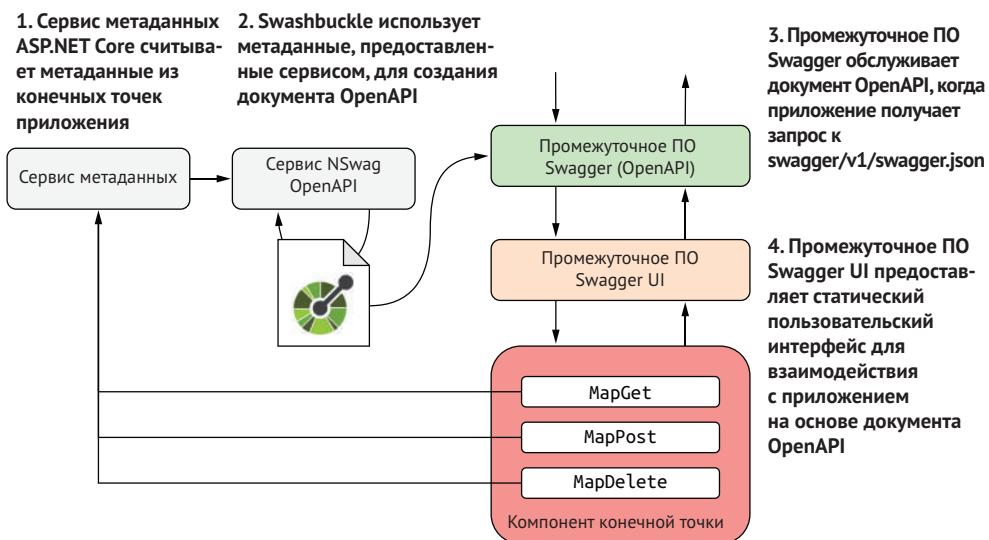


Рис. 11.1 Swashbuckle использует службы метаданных ASP.NET Core для получения информации о конечных точках приложения и создает документ OpenAPI. Промежуточное ПО OpenAPI предоставляет этот документ по запросу. Swashbuckle также включает дополнительный компонент для визуализации документа OpenAPI с использованием Swagger UI

После установки Swashbuckle настройте свое приложение для создания документа OpenAPI, как показано в листинге 11.1. В этом листинге показана сокращенная версия API fruit из главы 5, в которую для простоты включены только методы GET и POST. Дополнения, связанные с OpenAPI, выделены жирным шрифтом.

ПРИМЕЧАНИЕ Swashbuckle использует в именах методов старую номенклатуру Swagger, а не OpenAPI. Рассматривайте OpenAPI как название спецификации, а Swagger как название инструмента, связанного с OpenAPI, как описано в этом посте: <http://mng.bz/o18M>.

Листинг 11.1. Добавление поддержки OpenAPI в приложение с минимальным API с помощью Swashbuckle

```
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer(); <-- Добавляет функции обнаружения конечных точек ASP.NET Core, необходимые Swashbuckle
builder.Services.AddSwaggerGen(); <-- Добавляет службы Swashbuckle, необходимые для создания документов OpenAPI

WebApplication app = builder.Build(); <-- Добавляет службы Swashbuckle, необходимые для создания документов OpenAPI

var _fruit = new ConcurrentDictionary<string, Fruit>();

app.UseSwagger(); <-- Добавляет компонент для предоставления документа
app.UseSwaggerUI(); <-- OpenAPI для вашего приложения <-- Добавляет компонент для предоставления документа OpenAPI для вашего приложения

Добавляет компонент, обслуживающий Swagger UI
app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
    ? TypedResults.Ok(fruit)
    : Results.Problem(statusCode: 404));
app.MapPost("/fruit/{id}", (string id, Fruit fruit) =>
    _fruit.TryAdd(id, fruit)
    ? TypedResults.Created($"/fruit/{id}", fruit)
    : Results.ValidationProblem(new Dictionary<string, string[]>
    {
        { "id", new[] { "A fruit with this id already exists" } }
    }));
app.Run();
record Fruit(string Name, int Stock);
```

Благодаря изменениям в этом списке ваше приложение предоставляет описание своих конечных точек OpenAPI. Если вы запустите приложение и перейдете к `/swagger/v1/swagger.json`, то найдете большой файл JSON, похожий на тот, что показан на рис. 11.2. Этот файл представляет собой описание приложения в документе OpenAPI.

Документ OpenAPI включает общее описание приложения, например название и версию, а также конкретные сведения о каждой из конечных точек. Например, на рис. 11.2 конечная точка `/fruit/{id}` описывает тот факт, что ей нужен метод GET, и принимает параметр `id` в пути.

Вы можете изменить некоторые значения документа, например заголовок, добавив конфигурацию в метод `AddSwaggerGen()`. Можно задать для заголовка приложения значение "Fruitify" и добавить описание документа:

```
builder.Services.AddSwaggerGen(x =>
    x.SwaggerDoc("v1", new OpenApiInfo()
    {
        Title = "Fruitify",
        Description = "An API for interacting with fruit stock",
        Version = "1.0"
    }));
}
```

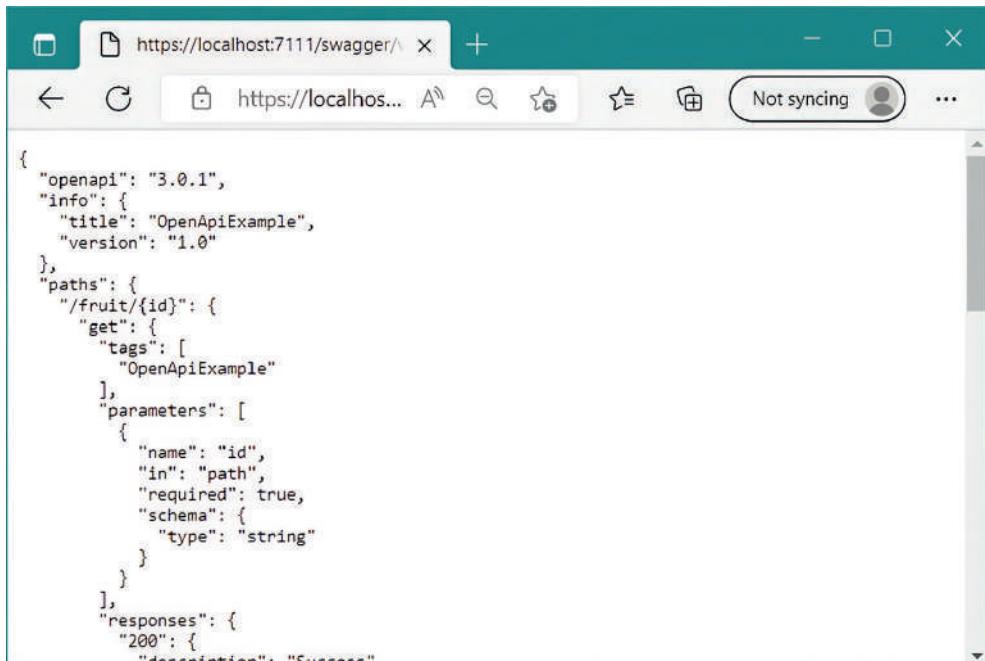


Рис. 11.2. Документ OpenAPI для приложения, описанный в листинге 11.1 и созданный с помощью NSwag

Вы также можете изменить такие параметры, как путь, используемый для предоставления документа, и различные детали того, как Swashbuckle генерирует окончательный файл в формате JSON. Подробности см. в документации: <http://mng.bz/OxQR>.

Это все, конечно, хорошо, но вы можете пожать плечами и спросить: «Ну и что?» OpenAPI по-настоящему проявляет себя, когда предоставляет другие инструменты. И мы уже добавили в свое приложение один такой инструмент: Swagger UI.

11.2 Тестирование API с помощью Swagger UI

В этом разделе вы познакомитесь с Swagger UI (<https://swagger.io/tools/swagger-ui>), пользовательским интерфейсом с открытым исходным кодом, который позволяет легко визуализировать и тестировать приложения с OpenAPI. В некотором смысле Swagger UI можно рассматривать как облегченную версию Postman, которую я использовал

в предыдущих главах для взаимодействия с приложениями с минимальными API. Swagger UI предоставляет простой способ просмотра всех конечных точек приложения и отправки запросов к ним. Postman предоставляет множество дополнительных функций, таких как создание коллекций и обмен ими с вашей командой, но если все, что вы пытаетесь сделать, – это протестировать свое приложение локально, тогда Swagger UI – отличный вариант.

Вы можете в любой момент добавить Swagger UI в свое приложение ASP.NET Core, вызвав

```
app.UseSwaggerUI()
```

как показано в листинге 11.1. Оно автоматически интегрируется с промежуточным ПО OpenAPI для документов и по умолчанию предоставляет веб-интерфейс Swagger UI в приложении по пути /swagger. Перейдите в /swagger в своем приложении, и вы увидите страницу, подобную той, что показана на рис. 11.3.

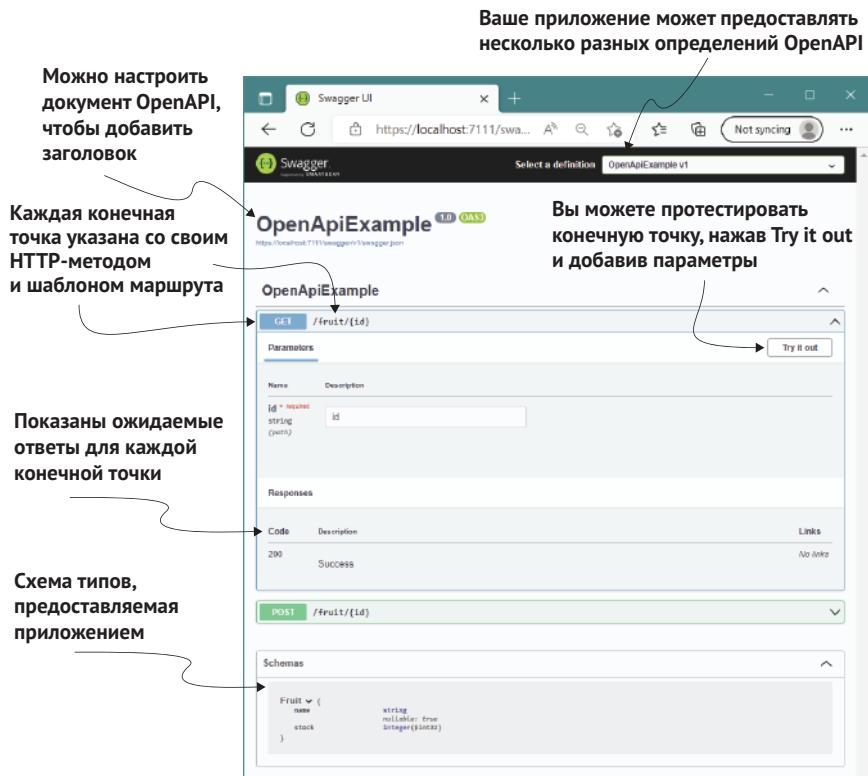


Рис. 11.3 Конечная точка Swagger UI для приложения. С помощью этого пользовательского интерфейса можно просматривать все конечные точки своего приложения, схему отправляемых и возвращаемых объектов и даже тестировать API, предоставляя параметры и отправляя запросы

В Swagger UI перечислены все конечные точки, описанные в документе OpenAPI, схема объектов, которые отправляются и получаются от каждого API, а также все возможные ответы, которые может возвращать каждая конечная точка. Вы даже можете протестировать API, выбрав **Try it out** (Попробовать), введя значение параметра и выбрав **Execute** (Выполнить). Swagger UI показывает выполненную команду, заголовки и тело ответа (рис. 11.4).

Swagger UI – это полезный инструмент для изучения API, который в некоторых случаях может заменить такой инструмент, как Postman. Но примеры, которые мы показывали до сих пор, демонстрируют проблему с нашим API: ответы, описанные для конечной точки GET на рис. 11.3, упоминают ответ с кодом состояния 200, но на рис. 11.4 показано, что он также может возвращать код состояния 404. Чтобы решить эту проблему с документацией, нужно добавить дополнительные метаданные в наши API.

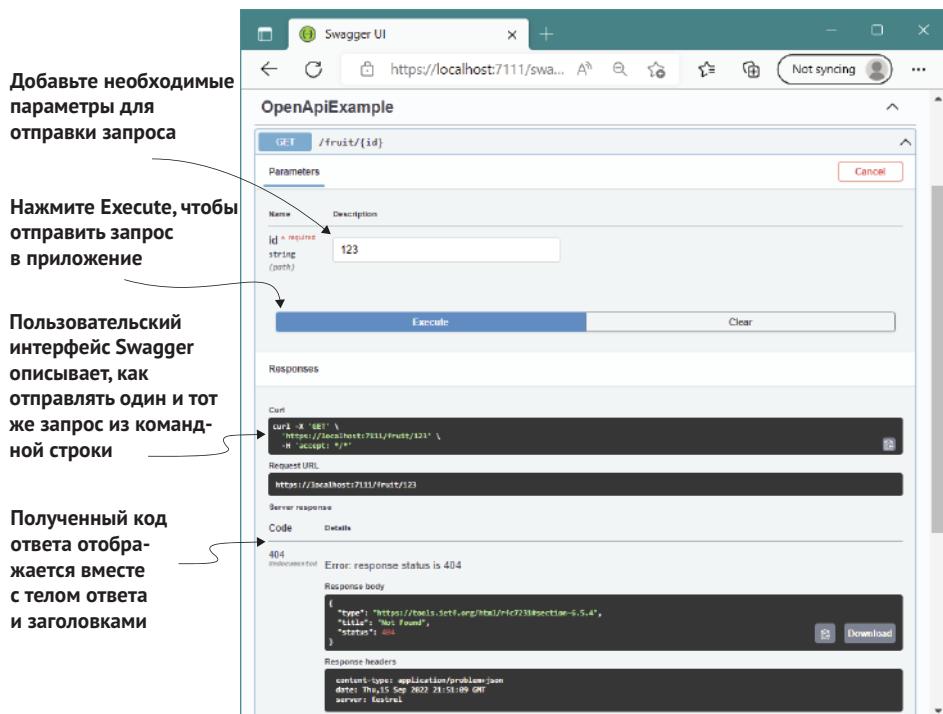


Рис. 11.4 Вы можете отправлять запросы с помощью Swagger UI, выбрав API, введя необходимые параметры и выбрав «Выполнить». Swagger UI показывает полученный ответ

11.3 Добавление метаданных в минимальные API

Метаданные – это информация об API, которая не меняет выполнение самого API. Мы использовали метаданные в главе 5, когда добавляли имена к своим конечным точкам с помощью метода `WithName()`, чтобы можно было ссылаться на них с помощью `LinkGenerator`. Имя ничего не

меняет в работе конечной точки, но предоставляет информацию для подключения других функций.

В настоящее время в конечные точки минимальных API можно добавить три основные категории метаданных:

- *метаданные маршрутизации* – как вы уже видели, методы `WithName()` добавляют глобально уникальное имя к конечной точке, которая используется для генерации URL-адресов;
- *метаданные для другого промежуточного ПО* – несколько частей промежуточного ПО можно настроить для каждого запроса путем добавления метаданных в конечную точку. При запуске оно проверяет метаданные выбранной конечной точки и действует соответствующим образом. Примеры включают авторизацию, фильтрацию имен хостов и кеширование вывода;
- *метаданные OpenAPI* – генерация документов OpenAPI управляетяется метаданными, предоставляемыми конечными точками, которые, в свою очередь, управляют пользовательским интерфейсом, предоставляемым Swagger UI.

В главе 25 мы рассмотрим, как добавить метаданные авторизации в конечные точки, а сейчас сосредоточимся на улучшении описания OpenAPI приложения с использованием метаданных. Можно предоставить множество подробностей для документирования API, некоторые из которых Swashbuckle использует во время генерации OpenAPI, а некоторые – нет. В следующем листинге показано, как добавить тег для каждого API и как явно описать возвращаемые ответы с помощью метода `Produces()`.

Листинг 11.2. Добавление метаданных OpenAPI для улучшения документирования конечных точек

```
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

WebApplication app = builder.Build();

var _fruit = new ConcurrentDictionary<string, Fruit>();

app.UseSwagger();
app.UseSwaggerUI();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .WithTags("fruit")           ◀
    .Produces<Fruit>()          ◀
    .ProducesProblem(404);       ◀
```

Если идентификатор не найден, конечная точка возвращает ответ 404

Добавление тега группирует конечные точки в Swagger UI. Каждая конечная точка может иметь несколько тегов

Конечная точка может возвращать объект `Fruit`. Если не указано, предполагается ответ 200

```

app.MapPost("/fruit/{id}", (string id, Fruit fruit) =>
    _fruit.TryAdd(id, fruit)
    ? TypedResults.Created($"/fruit/{id}", fruit)
    : Results.ValidationProblem(new Dictionary<string, string[]>
    {
        { "id", new[] { "A fruit with this id already exists" } }
    }))
    .WithTags("fruit")           ↪ Добавление тега группирует конечные точки в Swagger
    .Produces<Fruit>(201)      ↪ UI. Каждая конечная точка может иметь несколько тегов
    .ProducesValidationProblem(); ↪

app.Run();
record Fruit(string Name, int stock);
```

Если идентификатор уже существует, он возвращает ответ 400 «Сведения о проблеме» с ошибками проверки

Эта конечная точка также возвращает объект Fruit, но использует ответ 201 вместо 200

Благодаря этим изменениям Swagger UI показывает правильные ответы для каждой конечной точки, как представлено на рис. 11.5. Он также группирует конечные точки под тегом "fruits" вместо тега по умолчанию, выведенного из имени проекта, когда теги не указаны.

Если добавление всех этих дополнительных метаданных кажется вам утомительным занятием, не волнуйтесь. Добавление дополнительных метаданных OpenAPI не является обязательным, оно необходимо, только если вы планируете предоставить доступ к своему документу OpenAPI другим пользователям. Если все, что вам нужно, – это простой способ протестировать свои минимальные API, можно обойтись без многих из этих дополнительных вызовов методов.

Пользовательский интерфейс Swagger группирует конечные точки на основе метаданных их тегов

Ответ 200 теперь включает в себя схему ответа

Все ответы, добавленные с помощью методов Produces, теперь перечислены

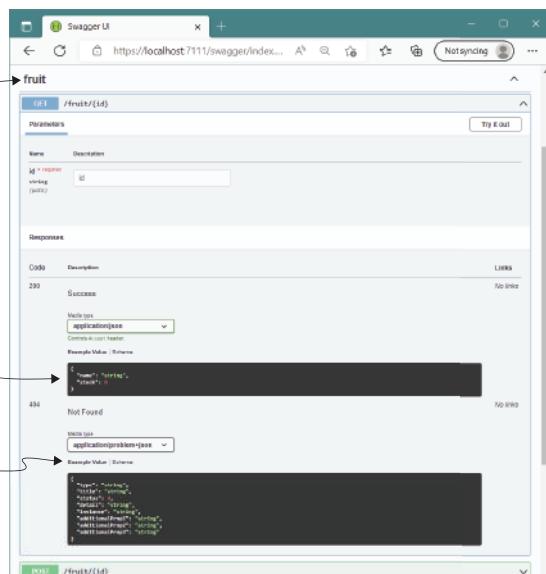


Рис. 11.5 Swagger UI группирует конечные точки приложения на основе прикрепленных к ним метаданных Tag. Пользовательский интерфейс использует метаданные, добавленные при вызове метода Produces(), для документирования ожидаемых типов возврата и кодов состояния для каждой конечной точки

СОВЕТ Помните, что вы также можете использовать группы маршрутов (описанные в главе 5) для одновременного применения метаданных к нескольким API.

Одним из самых сильных аргументов в пользу максимально подробного описания OpenAPI является тот факт, что это упрощает использование инструментов API. Swagger UI – один из примеров. Но, возможно, еще более полезный инструмент позволяет автоматически создавать клиентов C# для взаимодействия с вашими API.

11.4 Создание строго типизированных клиентов с помощью NSwag

В этом разделе вы узнаете, как использовать описание OpenAPI для автоматического создания клиентского класса, который можно использовать для вызова API из другого проекта C#. Вы создадите консольное приложение, используя инструмент .NET для создания клиента C# для взаимодействия с вашим API и, наконец, настроите сгенерированные типы. Сгенерированный код включает автоматическую сериализацию и десериализацию типов запросов и значительно упрощает взаимодействие с API из другого проекта C#, чем альтернативный метод создания HTTP-запросов вручную.

ПРИМЕЧАНИЕ Создание строго типизированного клиента не является обязательным. Это упрощает использование API из C#, но если вам не нужна эта функциональность, вы все равно можете протестировать свои API с помощью Postman или другого HTTP-клиента.

Для автоматического создания клиента C# на основе описания OpenAPI можно использовать любой из нескольких инструментов, например OpenAPI Generator (<http://mng.bz/Y1wB>), но в этой главе я использую NSwag. Возможно, вы помните из раздела 11.1, что NSwag можно использовать вместо Swashbuckle для создания описания OpenAPI для своего API. Но в отличие от Swashbuckle, NSwag также содержит генератор *клиентов*. Он является библиотекой по умолчанию, используемой как Visual Studio, так и глобальным инструментом Microsoft .NET OpenAPI для создания клиентского кода C#.

Генерация кода на основе описания OpenAPI осуществляется посредством процесса, показанного на рис. 11.6. Сначала Visual Studio или инструмент .NET загружают JSON-файл описания OpenAPI, чтобы он был доступен локально. Инструмент генерации кода считывает описание OpenAPI, идентифицирует все конечные точки и схемы, описанные в документе, и создает клиентский класс C#, который можно использовать для вызова API, описанного в документе. Инструмент генерации кода подключается к процессу сборки, поэтому при каждом изменении локального файла описания OpenAPI запускается генератор кода для повторной генерации клиента.

1. Visual Studio или инструмент командной строки .NET локально загружает файл описания OpenAPI и устанавливает необходимые пакеты NuGet
2. Пакет NuGet подключается к сборке проекта, считывает описание OpenAPI и генерирует клиента C#
3. Вы можете использовать генерированного клиента в своей программе для отправки запросов к API

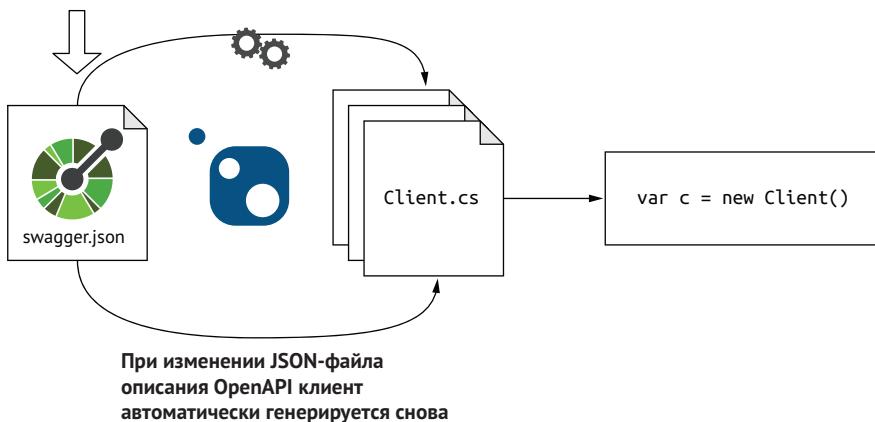


Рис. 11.6 Visual Studio или инструмент .NET локально загружает описание OpenAPI и устанавливает инструмент создания кода из NuGet. При сборке проекта инструмент генерации считывает описание OpenAPI и генерирует класс C# для взаимодействия с API

Вы можете создавать клиентов с помощью Visual Studio, как показано в разделе 11.4.1, или инструмента .NET, как показано в разделе 11.4.2. Оба подхода дают один и тот же результат, поэтому ваш выбор – вопрос личных предпочтений.

11.4.1 Генерация клиента с помощью Visual Studio

В этом разделе я покажу, как создать клиента, используя встроенную поддержку Visual Studio. Предполагается, что у вас есть простое консольное приложение .NET 7, которому необходимо взаимодействовать с приложением с минимальным API.

ПРИМЕЧАНИЕ В примере кода для этой главы оба приложения находятся в одном и том же решении для простоты, но это не обязательно. Вам даже не нужен исходный код API; если у вас есть описание OpenAPI для API, вы можете создать для него клиента.

Чтобы создать клиента, выполните следующие действия.

- 1 Убедитесь, что приложение API запущено и что JSON-файл описания OpenAPI доступен. Обратите внимание на URL-адрес, по которому доступен файл JSON. Если вы работаете в соответствии с исходным кодом из книги, запустите проект OpenApiExample.
- 2 В клиентском проекте щелкните правой кнопкой мыши файл проекта и выберите **Add > Service Reference** (Добавить > Ссылка на сервис) в контекстном меню, как показано на рис. 11.7. Эта команда открывает диалоговое окно **Add Service Reference** (Добавить ссылку на сервис).

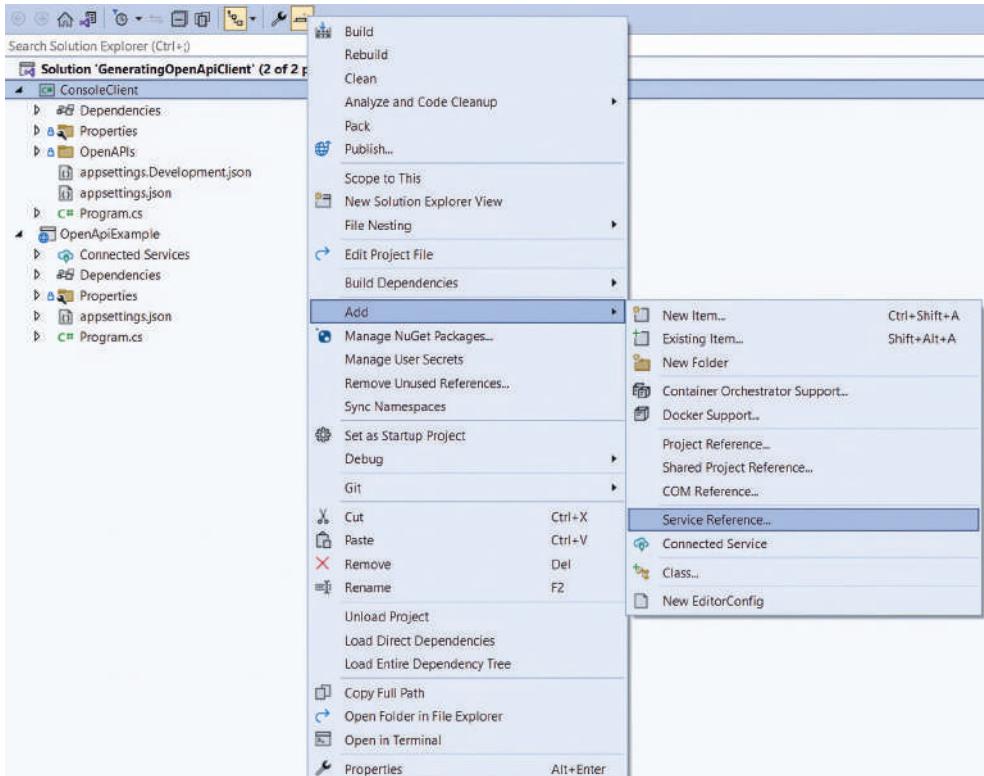


Рис. 11.7. Добавление ссылки на сервис с помощью Visual Studio. Щелкните правой кнопкой мыши на проект, который будет вызывать API, и выберите **Добавить > Ссылка на сервис**

- 3 В диалоговом окне «Добавить ссылку на сервис» выберите **OpenAPI** и нажмите **Next** (Далее). На странице **Add New OpenAPI Service Reference** (Добавить новую ссылку на сервис OpenAPI) введите URL-адрес, по которому находится документ OpenAPI. Введите пространство имен для сгенерированного кода и имя сгенерированного клиентского класса, как показано на рис. 11.8, а затем нажмите **Finish** (Готово). На экране **Service Reference Configuration Progress** (Ход настройки ссылки на сервис) показаны изменения, которые Visual Studio вносит в приложение, например установка различных пакетов NuGet и загрузка документа OpenAPI.

COBET Если вы запускаете пример кода в Visual Studio, то можете найти документ OpenAPI на странице <https://localhost:7186/swagger/v1/swagger.json>. Это расположение также отображается в Swagger UI.

После выполнения этих шагов просмотрите файл csproj вашего консольного приложения. Вы увидите, что было добавлено несколько ссылок на пакеты NuGet, а также новый элемент <OpenApiReference>, как показано в листинге 11.3.

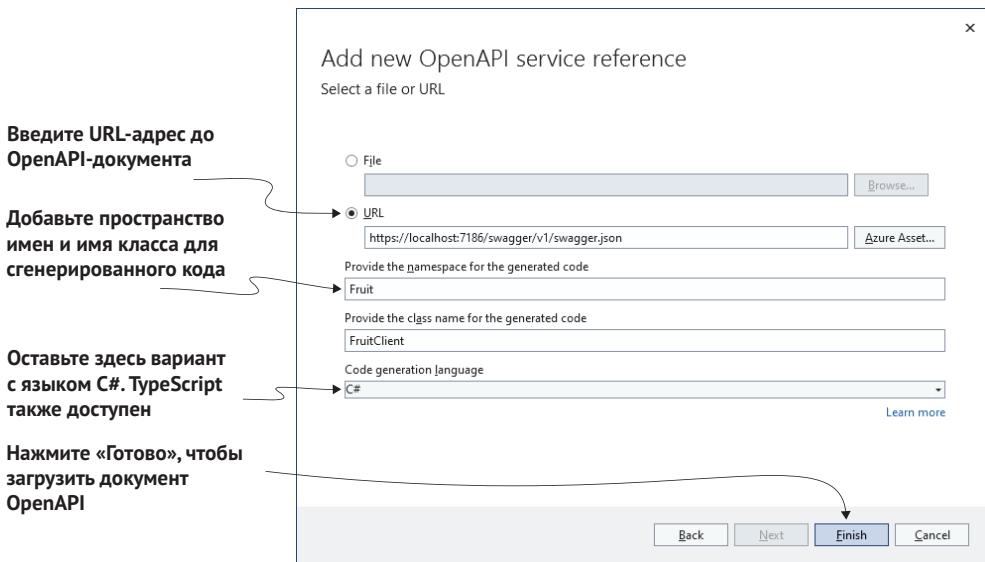


Рис. 11.8 Добавление ссылки на сервис OpenAPI с помощью Visual Studio. Добавьте ссылку на документ OpenAPI, параметры генерации кода и нажмите «Готово». Visual Studio загружает документ OpenAPI и сохраняет его в проекте, чтобы использовать его для генерации кода

Листинг 11.3. Добавление ссылки на сервис для генерации клиента OpenAPI с помощью Visual Studio

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

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net7.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
    </PropertyGroup>
    <ItemGroup>
        <OpenApiReference > Определяет, откуда было загружено описание
            <SourceUri>https://localhost:7186/swagger/v1/swagger.json</SourceUri>
        </OpenApiReference>
    </ItemGroup>

    <ItemGroup>
        <PackageReference > Генератору кода требуются дополнительные
            <Include>Microsoft.Extensions.ApiDescription.Client</Include>
            <Version>3.0.0</Version>
            <PrivateAssets>all</PrivateAssets>
            <IncludeAssets>runtime; build; native; contentfiles; analyzers;
                buildtransitive</IncludeAssets>
        </PackageReference>
    </ItemGroup>
```

```
<PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
<PackageReference Include="NSwag.ApiDescription.Client"
    Version="13.0.5">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers;
        buildtransitive</IncludeAssets>
</PackageReference>
</ItemGroup>

</Project>
```

Теоретически в этом коде должно быть все, что вам нужно для генерации клиента. К сожалению, Visual Studio добавляет некоторые устаревшие пакеты, которые необходимо обновить перед сборкой проекта, а именно:

- 1 обновите NSwag.ApiDescription.Client до последней версии (сейчас 13.18.2). Этот пакет выполняет генерацию кода на основе описания OpenAPI;
- 2 обновите Microsoft.Extensions.ApiDescription.Client до последней версии (7.0.0 на момент выпуска .NET 7). В любом случае на этот пакет NSwag.ApiDescription.Client ссылается транзитивно, поэтому не нужно ссылаться на него напрямую, но это гарантирует, что у вас будет последняя версия пакета.

ПРИМЕЧАНИЕ По умолчанию созданный клиент использует Newtonsoft.Json для сериализации запросов и ответов. В разделе 11.4.4 вы увидите, как заменить его встроенным System.Text.Json.

После внесения этих изменений ваш проект должен выглядеть примерно так, как показано в следующем листинге.

Листинг 11.4. Обновление версий пакета для генерации OpenAPI

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

<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
</PropertyGroup>
<ItemGroup>
    <OpenApiReference
        Include="OpenAPIs\swagger.json"
        CodeGenerator="NSwagCSharp"
        Namespace="Fruit"
        ClassName="FruitClient">
        <SourceUri>https://localhost:7186/swagger/v1/swagger.json</SourceUri>
    </OpenApiReference>
</ItemGroup>

<ItemGroup>
    <PackageReference
```

```

    Include="Microsoft.Extensions.ApiDescription.Client"
Обновление до последней версии
<PrivateAssets>all</PrivateAssets>
<IncludeAssets>runtime; build; native; contentfiles; analyzers;
  buildtransitive</IncludeAssets>
</PackageReference>
<PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
<PackageReference Include="NSwag.ApiDescription.Client"
  Version="13.18.2">
  <PrivateAssets>all</PrivateAssets>
  <IncludeAssets>runtime; build; native; contentfiles; analyzers;
  buildtransitive</IncludeAssets>
</PackageReference>
</ItemGroup>
</Project>

```

После обновления пакетов вы можете создать свой проект и сгенерировать `FruitClient`. В разделе 11.4.3 вы увидите, как использовать этого клиента для вызова вашего API, но сначала мы рассмотрим, как сгенерировать клиента с помощью глобального инструмента .NET, если вы не используете Visual Studio.

11.4.2 Создание клиента с помощью инструмента .NET Global

В этом разделе вы узнаете, как создать клиента на основе определения OpenAPI, используя глобальный инструмент .NET вместо Visual Studio. Результат, по сути, тот же, поэтому если вы выполнили шаги, описанные в разделе 11.4.1 в Visual Studio, можете пропустить этот раздел.

ПРИМЕЧАНИЕ Вам не обязательно использовать Visual Studio или инструмент .NET. В конце концов, вам в проекте нужен файл `csproj`, похожий на листинг 11.4, и JSON-файл определения OpenAPI, поэтому если вы готовы отредактировать файл проекта и загрузить определение вручную, то можете использовать этот подход. Visual Studio и инструмент .NET упрощают и автоматизируют некоторые из этих шагов.

Как и в разделе 11.4.1, инструкции в 11.4.2 предполагают, что у вас есть консольное приложение, которому необходимо вызывать ваш API, что API доступен и что у него есть описание OpenAPI.

Чтобы создать клиента с помощью NSwag, выполните следующие действия.

- 1 Убедитесь, что приложение API запущено и что JSON-файл описания OpenAPI доступен. Обратите внимание на URL-адрес, по которому доступен файл JSON. В исходном коде, связанном с книгой, запустите проект `OpenApiExample`.
- 2 Установите инструмент .NET OpenAPI (<http://mng.bz/GyOv>) глобально, выполнив команду

```
dotnet tool install -g Microsoft.dotnet-openapi
```

- 3 Из папки проекта консольного приложения добавьте ссылку OpenAPI с помощью следующей команды, заменив путь к документу OpenAPI и место для загрузки файла JSON:

```
dotnet openapi add url http://localhost:5062/swagger/v1/swagger.json  
→ --output-file OpenAPIs\fruit.json
```

СОВЕТ Если вы запускаете пример кода с помощью команды `dotnet run`, можете найти документ OpenAPI по предыдущему URL-адресу. Это расположение также отображается в Swagger UI.

- 4 Обновите пакеты, добавленные в ваш проект, выполнив следующие команды из папки проекта:

```
dotnet add package NSwag.ApiDescription.Client  
dotnet add package Microsoft.Extensions.ApiDescription.Client  
dotnet add package Newtonsoft.Json
```

После выполнения всех этих шагов файл описания OpenAPI должен быть загружен в `OpenAPIs\fruit.json`, а файл проекта должен выглядеть примерно так, как показано в следующем листинге (элементы, добавленные с помощью инструмента, выделены жирным шрифтом).

Листинг 11.5. Добавление ссылки OpenAPI с помощью инструмента .NET OpenAPI

```
<Project Sdk="Microsoft.NET.Sdk">  
  <PropertyGroup>  
    <OutputType>Exe</OutputType>  
    <TargetFramework>net7.0</TargetFramework>  
    <ImplicitUsings>enable</ImplicitUsings>  
    <Nullable>enable</Nullable>  
  </PropertyGroup>  
  
  <ItemGroup>  
    <PackageReference  
      Include="Microsoft.Extensions.ApiDescription.Client"  
      Version="7.0.0">  
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;  
      buildtransitive</IncludeAssets>  
      <PrivateAssets>all</PrivateAssets>  
    </PackageReference>  
    <PackageReference Include="Newtonsoft.Json" Version="13.0.1" />  
    <PackageReference Include="NSwag.ApiDescription.Client"  
      Version="13.18.2">  
      <IncludeAssets>runtime; build; native; contentfiles; analyzers;  
      buildtransitive</IncludeAssets>  
      <PrivateAssets>all</PrivateAssets>  
    </PackageReference>  
  </ItemGroup>  
  
  <ItemGroup>  
    <OpenApiReference Include="OpenAPIs\fruit.json"  
      SourceUrl="http://localhost:5062/swagger/v1/swagger.json" />  
  </ItemGroup>  
</Project>
```

Помимо незначительных различий в порядке, основное различие между подходом Visual Studio и подходом инструмента .NET заключается в том, что Visual Studio позволяет указать имя класса и пространство имен для нового клиента, тогда как инструмент .NET использует значения по умолчанию. Для обеспечения единства добавьте атрибуты `ClassName` и `Namespace` к элементу `<OpenApiReference>`, добавленному инструментом:

```
<OpenApiReference Include="OpenAPIs\fruit.json"
    SourceUrl="http://localhost:5062/swagger/v1/swagger.json"
    Namespace="Fruit"
    ClassName="FruitClient" />
```

В разделе 11.4.4 вы узнаете, как дальше настраивать сгенерированный код, но, прежде чем мы перейдем к этой теме, посмотрим на сгенерированный `FruitClient` и на то, как его использовать.

11.4.3 Использование сгенерированного клиента для вызова API

До сих пор вы верили мне на слово, что для вашего приложения волшебным образом создается клиент, но в этом разделе вы сможете его опробовать. Пакет `NSwag.ApiDescription.Client`, добавленный в ваш проект, работает с пакетом `Microsoft.Extensions.ApiDescription.Client` для чтения файла описания OpenAPI в проекте. Из этого описания можно определить, какие API у вас есть и какие типы нужно сериализовать. Наконец, он выводит класс C# с именем класса и пространством имен, указанными в элементе `OpenApiReference`.

ПРИМЕЧАНИЕ Сгенерированный файл обычно сохраняется в папке `obj` вашего проекта. После сборки своего проекта вы можете найти файл `FruitClient.cs` в этой папке. В качестве альтернативы используйте функцию Go To Definition (Перейти к определению) (**F12**) Visual Studio в экземпляре `FruitClient`, чтобы перейти к коду в интегрированной среде разработки.

Чтобы использовать `FruitClient` для вызова вашего API, вы должны создать его экземпляр, передав базовый адрес API и экземпляр `HttpClient`. Затем вы можете отправлять HTTP-запросы на обнаруженные конечные точки. Например, клиент, созданный на основе описания OpenAPI простого минимального API в листинге 11.2, будет иметь методы `FruitPOSTAsync()` и `FruitGETSync()`, соответствующие двум предоставленным методам, как показано в следующем листинге.

Листинг 11.6. Вызов API из листинга 11.2 с использованием сгенерированного клиента

```
using Fruit; ← Код генерируется в пространстве имен Fruit

var client = new FruitClient(← Использует сгенерированный FruitClient
    "https://localhost:7186", ← Указывает базовый адрес API
    new HttpClient()); ← Предоставленный HttpClient используется для вызова API
```

```

    Fruit.Fruit created = await client.FruitPOSTAsync("123",
        new Fruit.Fruit { Name = "Banana", Stock = 100 });
    Console.WriteLine($"Created {created.Name}");
    ↑
    ↑ Тип Fruit генерируется NSwag автоматически

    Fruit.Fruit fetched = await client.FruitGETAsync("123");
    Console.WriteLine($"Fetched {fetched.Name}");
    ↑
    ↑ Вызывает конечную точку API, MapGet

```

Этот код одновременно впечатляет и в некоторой степени ужасен:

- впечатляет тот факт, что вы можете генерировать весь шаблонный код для взаимодействия с API. Вам не нужно выполнять какую-либо интерполяцию строк для генерации пути, сериализовать тело запроса, или десериализовать ответ, или проверять коды состояния ошибок. Сгенерированный код берет все эти задачи на себя;
- у этих методов `FruitPOSTAsync` и `FruitGETAsync` действительно уродливые имена!

К счастью, вы можете исправить некрасивые имена методов: улучшите определение OpenAPI вашего API, добавив к каждому API метод `WithName()`. Имя, которое вы предоставляете для своей конечной точки, используется в качестве `OperationID` в описании OpenAPI; затем NSwag использует его для создания клиентских методов. Этот сценарий является ярким примером добавления большего количества макетов в OpenAPI, улучшая инструменты для ваших потребителей.

Помимо улучшения описания OpenAPI, в следующем разделе вы увидите, как настроить генерацию кода напрямую.

11.4.4 Настройка сгенерированного кода

В этом разделе вы узнаете о некоторых параметрах настройки, доступных в генераторе NSwag, и о том, почему можно их использовать. Здесь я рассматриваю три варианта настройки:

- использование `System.Text.Json` вместо `Newtonsoft.Json` для сериализации JSON;
- генерацию интерфейса для сгенерированной реализации клиента;
- когда не требуется явный параметр `BaseAddress` в конструкторе.

По умолчанию NSwag использует `Newtonsoft.Json` для сериализации запросов и десериализации ответов. `Newtonsoft.Json` – популярная, закаленная в боях библиотека JSON, но в .NET 7 естьстроенная библиотека JSON, `System.Text.Json`, которую ASP.NET Core использует по умолчанию для сериализации JSON.

Вместо использования двух библиотек JSON вы можете заменить сериализацию, используемую в вашем клиенте, на `System.Text.Json`.

Когда NSwag создает клиента, он помечает класс как `partial`. Это означает, что вы можете определить свой собственный частичный класс `FruitClient` (например) и добавить любые методы, которые, по вашему мнению, будут полезны для клиента. Сгенерированный клиент также предоставляет частичные методы, которые действуют как перехватчики (`hooks`) непосредственно перед отправкой или получением запроса.

ОПРЕДЕЛЕНИЕ Частичные методы в C# (<http://mng.bz/zXEB>) возвращают `void` и не имеют реализаций. Вы можете определить реализацию метода в отдельном файле частичного класса. Если вы не определяете реализацию, метод удаляется во время компиляции, таким образом можно использовать частичные методы в качестве высокопроизводительных обработчиков событий.

Расширение генерированных клиентов полезно, но во время тестирования часто возникает желание заменить генерированного клиента на интерфейс. Интерфейсы позволяют использовать имитации (mock) или фиктивные службы, чтобы ваши тесты не вызывали настоящий API, как вы узнали из главы 8. NSwag может помочь в этом процессе, автоматически генерируя интерфейс `IFruitClient`, который реализует `FruitClient`.

Наконец, предоставление базового адреса, на котором размещен API, на первый взгляд имеет смысл. Но, как мы обсуждали в главе 9, примитивные аргументы конструктора, такие как `string` и `int`, плохо сочетаются с внедрением зависимостей. Учитывая, что `HttpClient` содержит свойство `BaseAddress`, можно настроить NSwag так, чтобы он *не* требовал передачи базового адреса в качестве аргумента конструктора, а вместо этого напрямую задавал его для типа `HttpClient`. Такой подход помогает в сценариях внедрения зависимостей, как вы увидите, когда мы будем обсуждать `IHttpClientFactory` в главе 33.

Все эти три, казалось бы, несвязанных параметра настраиваются в NSwag одинаково: путем добавления элемента `Options` к элементу `<OpenApiReference>` в файле проекта. Параметры предоставляются в виде переключателей командной строки и должны быть указаны в одной строке без разрывов. Переключатели для трех описанных настроек:

- `/UseBaseUrl:false` – если значение равно `false`, NSwag удаляет параметр `baseUrl` из созданного конструктора клиента и вместо этого полагается на `HttpClient`, чтобы получить правильный базовый адрес. По умолчанию установлено значение `true`;
- `/GenerateClientInterfaces:true` – если задано значение `true`, NSwag создает интерфейс для клиента, содержащий все конечные точки. Генерированный клиент реализует этот интерфейс. По умолчанию имеет значение `false`;
- `/JsonLibrary:SystemTextJson` – этот параметр определяет используемую библиотеку сериализации JSON. По умолчанию используется `Newtonsoft.Json`.

СОВЕТ Для NSwag доступно огромное количество вариантов конфигурации. Я считаю, что лучшая документация доступна в инструменте NSwag .NET. Вы можете установить этот инструмент, используя команду `dotnet tool install -g NSwag.ConsoleCore`, а просмотреть доступные параметры можно, выполнив команду `nswag help openapi2csclient`.

Вы можете установить все три параметра, добавив элемент `<Options>` к элементу `<OpenApiReference>`, как показано в следующем

листеинге. Убедитесь, что вы правильно открываете и закрываете оба элемента, чтобы XML оставался действительным; легко допустить ошибку при редактировании вручную!

Листинг 11.7. Настройка параметров генератора NSwag

```
<OpenApiReference Include="OpenAPIs\fruit.json"
    SourceUrl="http://localhost:5062/swagger/v1/swagger.json"
    Namespace="Fruit"
    ClassName="FruitClient" >
    <Options>/UseBaseUrl:false /GenerateClientInterfaces:true
        ↳ /JsonLibrary:SystemTextJson</Options> ←
</OpenApiReference><!-- Обязательно закройте внешний элемент XML,
    чтобы XML оставался допустимым -->
```

Настраивает параметры, которые NSwag использует для генерации кода

Можно подумать, что после внесения этих изменений NSwag обновит сгенерированный код при следующей сборке. К сожалению, это не обязательно так просто.

NSwag отслеживает изменения в JSON-файле описания OpenAPI, сохраненном в вашем проекте, и будет повторно генерировать код при каждом изменении файла, но он не обязательно будет обновляться при изменении параметров в файле csproj. Хуже того, очистка или перестройка аналогичным образом не дадут никакого эффекта. Если вы оказались в такой ситуации, лучше всего удалить папку obj вашего проекта, чтобы убедиться, что все восстанавливается правильно.

СОВЕТ Еще один вариант – внести небольшое изменение в документ OpenAPI, чтобы NSwag обновлял сгенерированный код при сборке проекта. Затем вы можете отменить изменение документа OpenAPI.

После того как вы убедили NSwag повторно сгенерировать клиента, вам следует обновить свой код, чтобы использовать новые функции. Вы можете удалить ссылку Newtonsoft.Json из файла csproj и обновить Program.cs, как показано в следующем листинге.

Листинг 11.8. Использование обновленного клиента NSwag

```
using Fruit;          FruitClient теперь
                      реализует IFruitClient
IFruitClient client = new FruitClient( ←
    new HttpClient() { BaseAddress =
        new Uri("https://localhost:7186") });

Fruit.Fruit created = await client.FruitPOSTAsync("123",
    new Fruit.Fruit { Name = "Banana", Stock = 100 });
Console.WriteLine($"Created {created.Name}");

Fruit.Fruit fetched = await client.FruitGETAsync("123");
Console.WriteLine($"Fetched {fetched.Name}");
```

Устанавливает базовый адрес HttpClient вместо передачи в качестве аргумента конструктора

Если вы обновили идентификаторы операций для своих конечных точек API с помощью метода `WithName()`, возможно, вы несколько удивитесь, увидев, что эти уродливые методы `FruitPOSTAsync` и `FruitGETAsync` по-прежнему присутствуют, несмотря на то что вы повторно генерировали клиента. Это связано с тем, что описание OpenAPI, сохраненное в проекте, загружается только один раз, когда вы его изначально добавляете. Посмотрим, как обновить локальный документ OpenAPI, чтобы отразить изменения в вашем удаленном API.

11.4.5 Обновление описания OpenAPI

В этом разделе вы узнаете, как обновить документ описания OpenAPI, сохраненный в проекте и используемый для генерации. Этот документ не обновляется автоматически, поэтому клиент, созданный NSwag, может не отражать последнее описание OpenAPI для вашего API.

Независимо от того, использовали ли вы Visual Studio (как описано в разделе 11.4.1) или инструмент .NET OpenAPI (как описано в разделе 11.4.2), описание OpenAPI, сохраненное в виде файла JSON в проекте, представляет собой снимок API на определенный момент времени. Если вы добавите дополнительные метаданные в свой API, вам необходимо снова загрузить описание OpenAPI в свой проект.

СОВЕТ Я предпочитаю низкотехнологичный подход: я просто перехожу к описанию OpenAPI в браузере, копирую содержимое JSON и вставляю его в файл JSON в своем проекте.

Если вы не хотите обновлять описание OpenAPI вручную, то можно использовать Visual Studio или инструмент .NET OpenAPI, чтобы обновить сохраненный документ.

ВНИМАНИЕ! Если вы изначально использовали Visual Studio, то не сможете обновить документ с помощью инструмента OpenAPI, и наоборот. Причина в том, что Visual Studio использует атрибут `SourceUrl` элемента `OpenApiReference`, а инструмент .NET использует атрибут `SourceUrl`. И да, это произвольная ситуация, которая раздражает!

Чтобы обновить описание OpenAPI с помощью Visual Studio, выполните следующие действия.

- 1 Убедитесь, что ваш API работает и документ с описанием OpenAPI доступен.
- 2 Перейдите на страницу подключенных служб проекта, выбрав **Project > Connected Services > Manage Connected Services** (Проект > Подключенные службы > Управление подключенными службами).
- 3 Нажмите на кнопку, скрывающую дополнительные опции, рядом со ссылкой на OpenAPI и выберите **Refresh** (Обновить), как показано на рис. 11.9. Затем выберите **Yes** (Да) в диалоговом окне, чтобы обновить документ OpenAPI.

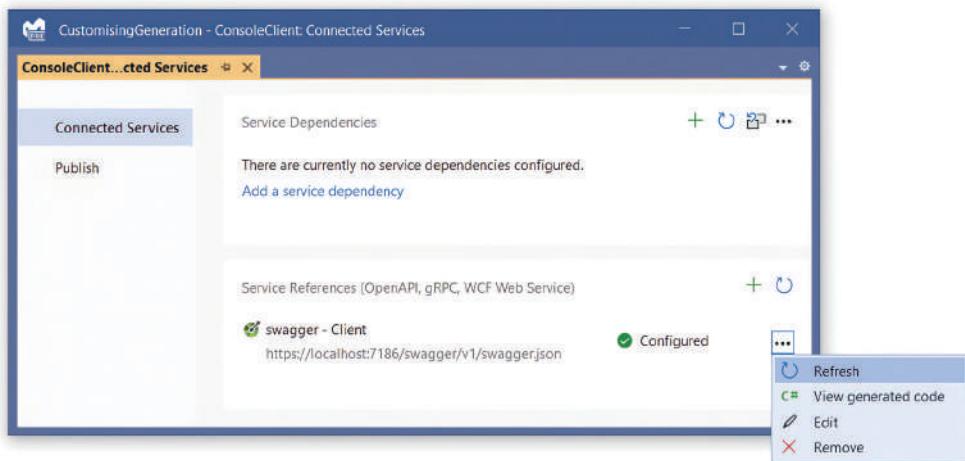


Рис. 11.9 Обновление описания OpenAPI для API. Выберите «Обновить», чтобы снова загрузить описание OpenAPI и сохранить его в своем проекте. Затем NSwag создаст обновленного клиента

Чтобы обновить описание OpenAPI с помощью инструмента .NET OpenAPI, выполните следующие действия.

- 1 Убедитесь, что ваш API работает и документ с описанием OpenAPI доступен.
- 2 В папке проекта выполните следующую команду, используя тот же URL-адрес, который вы использовали для первоначального добавления описания OpenAPI:

```
dotnet openapi refresh http://localhost:5062/swagger/v1/swagger.json
```

После обновления описания OpenAPI с помощью Visual Studio или инструмента .NET создайте приложение, чтобы NSwag активировал повторное создание клиента. Любые изменения, внесенные в описание OpenAPI (например, добавление идентификаторов операций), будут отражены в сгенерированном коде.

Я думаю, что генерация клиентов – это убийственное приложение для описаний OpenAPI, но лучше всего оно работает, когда вы используете метаданные для добавления обширной документации в свои API. В разделе 11.5 вы узнаете, как пойти еще дальше, добавив сводки и описания в конечные точки.

11.5 Добавление описаний и сводок в конечные точки

В этом разделе вы узнаете, как добавлять дополнительные описания и сводки в документ с описанием OpenAPI. Такие инструменты, как Swagger UI и NSwag, используют эти дополнительные описания и сводки, чтобы улучшить работу разработчиков с вашим API. Вы также уз-

наете об альтернативных способах добавления метаданных в конечные точки минимальных API.

11.5.1 Использование текущих методов для добавления описаний

Работая с конечными точками минимальных API и вызывая такие методы, как `WithName()` и `WithTags()`, вы, возможно, заметили методы `WithSummary()` и `WithDescription()`. Эти методы добавляют метаданные в конечную точку точно так же, как и другие методы `With*`, но, к сожалению, они не обновляют описание OpenAPI без дополнительных изменений.

Чтобы использовать сводки и описания метаданных, необходимо добавить дополнительный пакет NuGet, `Microsoft.AspNetCore.OpenApi`, и вызвать метод `WithOpenApi()` для своей конечной точки. Этот метод гарантирует, что сводки и описания метаданных будут правильно добавлены в описание OpenAPI, когда Swashbuckle генерирует документ.

Добавьте этот пакет через диспетчер пакетов NuGet или интерфейс командной строки .NET, вызвав команду

```
dotnet add package Microsoft.AspNetCore.OpenApi
```

из папки проекта. Затем обновите конечные точки, добавив сводки и/или описания, обязательно вызывая метод `WithOpenApi()`, как показано в следующем листинге.

Листинг 11.9. Добавление сводок и описаний к конечным точкам с помощью метода `WithOpenApi()`

```
using System.Collections.Concurrent;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

WebApplication app = builder.Build();

app.UseSwagger();
app.UseSwaggerUI();

var _fruit = new ConcurrentDictionary<string, Fruit>();

app.MapGet("/fruit/{id}", (string id) =>
    _fruit.TryGetValue(id, out var fruit)
        ? TypedResults.Ok(fruit)
        : Results.Problem(statusCode: 404))
    .WithName("GetFruit")
    .WithTags("fruit")
    .Produces<Fruit>()
    .ProducesProblem(404)
    .WithSummary("Fetches a fruit")
```

Добавляет описание
в конечную точку

```

Добавляет
сводку
в конечную
точку
    .WithDescription("Fetches a fruit by id, or returns 404" +
        " if no fruit with the ID exists")
    .WithOpenApi();
    ← Предоставляет метаданные, добавленные посред-
     ством сводки и описания, в описание OpenAPI
app.Run();
record Fruit(string Name, int Stock);

```

Благодаря этим изменениям в Swagger UI отображаются дополнительные метаданные, как показано на рис. 11.10.

NSwag также использует сводку в качестве комментария к документации при создании конечных точек для клиента. Однако на рис. 11.10 видно, что отсутствует часть документации: описание идентификатора параметра.

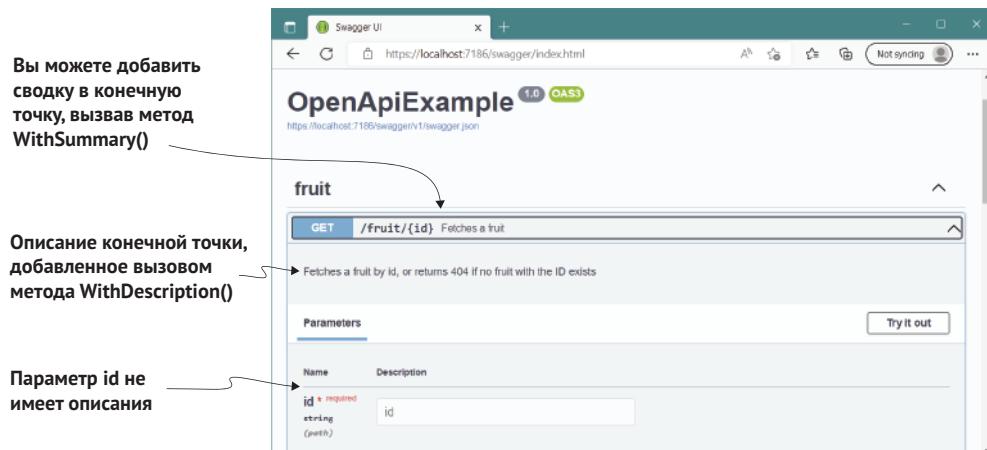


Рис. 11.10 Метаданные сводки и описания, отображаемые в Swagger UI. Обратите внимание, что для параметра id описание не отображается

К сожалению, у нас нет особенно элегантного способа добавления документации для параметров. Предлагаемый подход – использовать перегруженный вариант метода `WithOpenApi()`, принимающий лямбда-метод, в который можно добавить описание параметра:

```

.WithOpenApi(o =>
{
    o.Parameters[0].Description = "The id of the fruit to fetch";
    o.Summary = "Fetches a fruit";
    return o;
});

```

В этом примере показано, что вы можете использовать метод `WithOpenApi()`, чтобы задать любые метаданные OpenAPI для конечной точки, поэтому можно использовать этот единственный метод для установки (например) сводки и тегов вместо выделенного метода `WithSummary()` или `WithTags()`.

Добавление всех этих метаданных, несомненно, более подробно документирует API и упрощает понимание генерированного кода.

Но если вы хоть немного похожи на меня, то из-за огромного количества методов, которые вам нужно вызывать, будет сложно увидеть, где заканчивается конечная точка и начинаются метаданные! В следующем разделе мы рассмотрим альтернативный подход, предполагающий использование атрибутов.

11.5.2 Использование атрибутов для добавления метаданных

Во многих случаях я сторонник fluent-интерфейсов, поскольку считаю, что они облегчают понимание кода. Но расширения метаданных конечной точки, подобные показанным в листинге 11.9, доходят до крайностей. Трудно понять, что делает конечная точка, учитывая весь этот шум от методов метаданных! Начиная с версии 1.0 C# использует канонический подход для добавления метаданных в код – атрибутов, – и при желании можно заменить методы расширения конечной точки выделенными атрибутами.

Почти все методы расширения, которые вы добавляете в конечную точку, имеют эквивалентный атрибут, который можно использовать вместо него. Эти атрибуты следует применять непосредственно к методу-обработчику (лямбда-функции, если вы ее используете). В листинге 11.10 показан эквивалент листинга 11.9, в котором, где это возможно, вместо fluent-методов используются атрибуты. Метод `WithOpenApi()` – единственный вызов, который нельзя заменить; его нужно включить в состав, чтобы Swashbuckle правильно считывал метаданные OpenAPI.

Листинг 11.10. Использование атрибутов для описания API

```
using System.Collections.Concurrent;
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
WebApplication app = builder.Build();
app.UseSwagger();
app.UseSwaggerUI();
var _fruit = new ConcurrentDictionary<string, Fruit>();
app.MapGet("/fruit/{id}", [
    [EndpointName("GetFruit")]
    [EndpointSummary("Fetches a fruit")]
    [EndpointDescription("Fetches a fruit by id, or returns 404" +
        " if no fruit with the ID exists")]
    [ProducesResponseType(typeof(Fruit), 200)]
    [ProducesResponseType(typeof(HttpValidationProblemDetails), 404,
        "application/problem+json")]
    [Tags("fruit")]
    (string id) =>
        _fruit.TryGetValue(id, out var fruit)
            ? TypedResults.Ok(fruit)
            : Results.Problem(statusCode: 404))
    .WithOpenApi(o =>
```

Вы можете использовать атрибуты вместо вызовов текущих методов

```
{  
    o.Parameters[0].Description = "The id of the fruit to fetch";  
    return o;  
});  
  
app.Run();  
record Fruit(string Name, int Stock);
```

Какой листинг лучше – 11.10 или 11.9, – во многом это дело вкуса, но реальность такова, что ни один из них не отличается особой элегантностью. В обоих случаях метаданные существенно скрывают цель API, поэтому важно учитывать, какие метаданные стоит добавить, а какие – ненужный шум. Этот баланс может меняться в зависимости от вашей аудитории (внутренние или внешние клиенты), насколько зрелый ваш API и сколько вы можете извлечь из вспомогательных функций.

Одна из понятных жалоб как на атрибутный, так и на fluent-подход к присоединению метаданных OpenAPI заключается в том, что описания сводки и параметров отделены от обработчика конечной точки, к которому они применяются. В этом разделе вы увидите альтернативный подход, использующий комментарии к документации в формате Extensible Markup Language (XML).

Каждый пользователь-разработчик C# привыкнет к удобным описаниям методов и параметров, которые вы получаете в своей IDE от IntelliSense. Вы можете добавить эти описания в свои собственные методы, используя комментарии к документации в формате XML, например:

```
/// <summary>  
/// Adds one to the provided value and returns it  
/// </summary>  
/// <param name="value">The value to increment</param>  
public int Increment(int value) => value + 1;
```

В вашей IDE – будь то Visual Studio, JetBrains Rider или Visual Studio Code – это описание появляется при попытке вызвать метод. Разве не было бы чудесно использовать один и тот же синтаксис для определения сводки и описаний параметров для конечных точек OpenAPI? Что же, хорошая новость состоит в том, что мы можем это сделать!

ВНИМАНИЕ! Использование комментариев в формате XML поддерживается лишь частично в .NET 7. Эти комментарии работают только в том случае, если у вас есть статические обработчики или обработчики конечных точек метода экземпляра, а не лямбда-методы или локальные функции. Проблему, касающуюся комментариев в формате XML, можно найти на странице <https://github.com/dotnet/aspnetcore/issues/39927>.

Swashbuckle может использовать XML-комментарии, которые вы добавляете в обработчики конечных точек, в качестве описаний для OpenAPI. Если этот параметр активирован, .NET SDK создает

XML-файл, содержащий все ваши комментарии к документации. Swashbuckle может прочитать этот файл при запуске и использовать его для создания описаний OpenAPI, как показано на рис. 11.11.

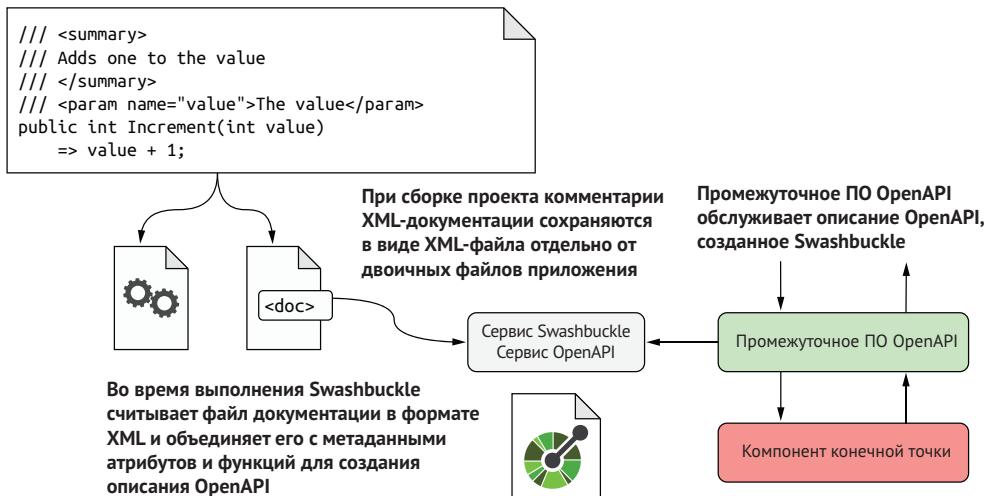


Рис. 11.11 Можно настроить приложение .NET для экспорта комментариев к документации в специальный XML-файл при сборке. Swashbuckle читает этот файл во время выполнения, объединяя его с метаданными атрибута и метода Fluent для конечной точки для создания окончательного описания OpenAPI

Чтобы активировать извлечение комментариев к документации в формате XML для документа описания OpenAPI, необходимо сделать три вещи:

- 1 Активируйте создание документации для своего проекта. Добавьте `<GenerateDocumentationFile>true</GenerateDocumentationFile>` внутри `<PropertyGroup>` в файле с расширением `.csproj` и задайте для него значение `true`:

```
<PropertyGroup>
    <GenerateDocumentationFile>true</GenerateDocumentationFile>
</PropertyGroup>
```

- 2 Настройте Swashbuckle для чтения сгенерированного XML-документа в методе `SwaggerGen()`:

```
builder.Services.AddSwaggerGen(opts =>
{
    var file = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
    opts.IncludeXmlComments(
        Path.Combine(ApplicationContext.BaseDirectory, file));
});
```

- 3 Используйте обработчик статического метода или метода экземпляра и добавьте комментарии в формате XML, как показано в следующем листинге.

Листинг 11.11. Добавление комментариев к документации в обработчик конечной точки

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Concurrent;
using System.Reflection; using Microsoft.AspNetCore.Mvc;
using System.Collections.Concurrent;
using System.Reflection;
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(opts => ← Активирует комментарии
{                                         в формате XML для описаний
    var file = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
    opts.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory, file));
});

WebApplication app = builder.Build();

app.UseSwagger();
app.UseSwaggerUI();

var _fruit = new ConcurrentDictionary<string, Fruit>();

var handler = new FruitHandler(fruit);
app.MapGet("/fruit/{id}", handler.GetFruit)
    .WithName("GetFruit"); ← Вы можете добавить дополнительные метаданные, используя методы
app.Run();
record Fruit(string Name, int Stock);

internal class FruitHandler
{
    private readonly ConcurrentDictionary<string, Fruit> _fruit;
    public FruitHandler(ConcurrentDictionary<string, Fruit> fruit)
    {
        _fruit = fruit;
    }
    /// <summary>
    /// Fetches a fruit by id, or returns 404 if it does not exist
    /// </summary>
    /// <param name="id" >The ID of the fruit to fetch</param>
    /// <response code="200">Returns the fruit if it exists</response>
    /// <response code="404">If the fruit doesn't exist</response>
    [ProducesResponseType(typeof(Fruit), 200)]
    [ProducesResponseType(typeof(HttpValidationProblemDetails),
        404, "application/problem+json")]
    [Tags("fruit")]
    public IActionResult GetFruit(string id)
        => _fruit.TryGetValue(id, out var fruit)
            ? TypedResults.Ok(fruit)
            : Results.Problem(statusCode: 404);
}

```

Вы должны использовать статические обработчики или обработчики экземпляров, а не лямбда-методы

Комментарии в формате XML используются в описании OpenAPI

Вы также можете добавить дополнительные метаданные, используя атрибуты метода обработчика

Мне нравится подход с комментариями в формате XML, поскольку он кажется гораздо более естественным для C#, а в IDE комментариям часто придают меньше внимания, что уменьшает визуальный беспорядок. Вам все равно придется использовать атрибуты и/или текущие методы для полного описания конечных точек для OpenAPI, но и такая мелочь будет полезна!

Как я уже неоднократно упоминал, насколько далеко вы зайдете с описанием OpenAPI, зависит от вас и какую пользу вы от него получите. Если вы хотите использовать OpenAPI только для локального тестирования с Swagger UI, нет смысла перегружать код большим количеством дополнительных метаданных. Фактически в таких случаях было бы лучше условно добавлять службы swagger и промежуточное ПО, только когда вы находитесь в окружении разработки, как в этом примере:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

if(builder.Environment.IsDevelopment())
{
    builder.Services.AddEndpointsApiExplorer();
    builder.Services.AddSwaggerGen();
}

WebApplication app = builder.Build();
if(app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
app.Run();
```

С другой стороны, если вы создаете C#-клиентов для вызова вашего API или предоставляете свой API для публичного использования, чем больше метаданных вы добавите, тем лучше! Также стоит отметить, что вы можете добавить описания OpenAPI для всех конечных точек приложения, а не только для конечных точек минимальных API. При создании контроллеров веб-API в главе 20 вы также можете включить их.

11.6 Ограничения OpenAPI

В этой главе я описал преимущества OpenAPI как для простого тестирования с использованием Swagger UI, так и для генерации кода. Но, как и в большинстве случаев, связанных с программным обеспечением, здесь не все так просто и легко. У OpenAPI и Swagger имеются ограничения, с которыми вы можете столкнуться, особенно по мере усложнения ваших API. В этом разделе описывается ряд проблем, на которые следует обратить внимание.

11.6.1 Не все API можно описать, используя OpenAPI

Спецификация OpenAPI предназначена для описания вашего API, чтобы любой клиент знал, как его вызывать. К сожалению, OpenAPI

не может описать все API, и это не случайность. В спецификации OpenAPI говорится: «Не все службы можно описать с помощью OpenAPI – эта спецификация не предназначена для охвата всех возможных стилей REST API». Таким образом, перед нами встает важный вопрос: какие API он *не может* описать?

Один из классических примеров – API, соответствующий архитектурным ограничениям для REST-приложений, известным как HATEOAS (Hypermedia as the Engine of Application State the Engine of Application State). В HATEOAS каждый запрос к конечной точке API включает список ссылок, описывающих действия, которые можно предпринять, и пути, используемые для каждого действия, что позволяет клиентам определять, какие действия доступны для данного ресурса. Сервер может добавлять или удалять ссылки динамически, в зависимости от состояния ресурса и того, какой пользователь делает запрос.

СОВЕТ У Мартина Фаулера есть отличное описание моделей зрелости REST, в которых HATEOAS является высшим уровнем зрелости: <http://mng.bz/OK1N>.

HATEOAS обычно усложняет работу небольших проектов, но это отличный способ отделить клиентские приложения от серверных API, чтобы они могли развиваться отдельно. Такой подход может окаться неоценимым, если у вас большие или независимые команды. Проблема OpenAPI состоит в том, что он не предназначен для таких динамических API. OpenAPI хочет заранее знать, каковы ответы для каждой конечной точки, а это не та информация, которую вы можете ему предоставить, если вы придерживаетесь HATEOAS.

В другом сценарии у вас может быть несколько серверных API, каждый из которых имеет собственную спецификацию OpenAPI. Вы предоставляете единое унифицированное приложение-шлюз API, с которым взаимодействуют все ваши клиенты. К сожалению, несмотря на то что каждый серверный API имеет спецификацию OpenAPI, не существует простого способа объединить API в единый унифицированный документ, которые вы можете предоставить в своем шлюзе API и которые клиенты могут использовать для тестирования и генерации кода.

Еще одна распространенная проблема связана с защитой API с помощью аутентификации и авторизации. Спецификация OpenAPI содержит раздел, описывающий требования к аутентификации, и Swagger UI поддерживает их. Но ситуация ухудшается, если вы используете какие-либо расширения общих протоколов аутентификации или расширенные функции. Хотя некоторые из этих рабочих процессов возможны, в некоторых случаях Swagger UI может просто не поддерживать ваш рабочий процесс, что делает Swagger UI непригодным для использования.

11.6.2 Сгенерированный код чрезмерно категоричен

В конце раздела 11.4 я сказал, что генерация кода – это главная особенность документов Open API, и во многих случаях так оно и есть.

Однако это утверждение предполагает, что вам нравится сгенерированный код. Если используемые вами инструменты – будь то NSwag или какой-либо другой генератор кода – не генерируют нужный вам код, вы можете потратить много усилий на настройку вывода. В какой-то момент и для некоторых API может оказаться проще написать собственного клиента!

ПРИМЕЧАНИЕ Классическая «боль» (которой я сочувствую) – использование исключений для потока процесса всякий раз, когда возвращается ошибка или неожиданный код состояния. Не все ошибки являются исключительными, создание исключений требует относительно больших вычислительных затрат, и это часто означает, что каждый вызов, выполняемый клиентом, требует специальной обработки исключений. Из-за этого генерация кода иногда кажется скорее бременем, чем преимуществом.

Другая, более тонкая проблема возникает, когда вы используете генерацию кода с двумя отдельными, но связанными документами OpenAPI, такими как API продуктов и API корзины. Если вы используете методы, описанные в этой главе, для создания клиентов, а затем попытаетесь следовать этой простой последовательности, вы столкнетесь с проблемой:

- 1 получить экземпляр `Product` из API продуктов с помощью `ProductsClient.Get()`;
- 2 отправить полученный экземпляр в API корзины с помощью `CartClient.Add(Product)`.

К сожалению, сгенерированный тип `Product`, полученный из API продуктов, отличается от сгенерированного типа `Product`, который требуется `CartClient`, поэтому этот код не скомпилируется. Даже если тип имеет одинаковые свойства и сериализуется в один и тот же файл в формате JSON при отправке клиенту, C# считает, что объекты относятся к разным типам, и не позволяет им поменяться местами. Необходимо вручную скопировать значения из первого экземпляра `Product` в новый экземпляр. Эти жалобы в основном представляют собой мелкие неприятности, но они могут накапливаться, если вы с ними часто сталкиваетесь.

11.6.3 Инструменты часто отстают от спецификации

Еще один фактор, который следует учитывать, – это множество групп, которые участвуют в создании документа OpenAPI и создания клиента:

- спецификация Open API – это проект сообщества, написанный группой OpenAPI Initiative;
- Microsoft предоставляет встроенные в ASP.NET Core инструменты для предоставления метаданных о конечных точках API;
- Swashbuckle – это проект с открытым исходным кодом, который использует метаданные ASP.NET Core для создания документа, совместимого с OpenAPI;

- NSwag – это проект с открытым исходным кодом, который использует документ, совместимый с OpenAPI, и генерирует клиентов (и имеет множество других функций!);
- Swagger UI – это проект с открытым исходным кодом для взаимодействия с API на основе документа OpenAPI.

Некоторые из этих проектов напрямую зависят от других (все зависит, например, от спецификации OpenAPI), но могут развиваться разными темпами. Если Swashbuckle не поддерживает какую-то новую функцию спецификации OpenAPI, она не появится в ваших документах, и NSwag не сможет ее использовать.

Большинство инструментов предоставляют способы переопределить поведение, чтобы обойти эти острые углы, но реальность такова, что если вы используете новые или менее популярные функции, вам может быть сложнее убедить все инструменты в вашей цепочке инструментов работать слаженно.

В целом важно помнить, что документы OpenAPI могут хорошо работать, если у вас простые требования или вы хотите использовать Swagger UI только для тестирования. В этих случаях для добавления поддержки OpenAPI требуется мало инвестиций, и это может улучшить рабочий процесс, так что, возможно, вам стоит попробовать.

Если у вас более сложные требования, вы создаете API, который OpenAPI не может с легкостью описать, или вы не являетесь поклонником генерации кода, возможно, не стоит тратить время на значительные инвестиции в OpenAPI для своих документов.

СОВЕТ Если вы поклонник генерации кода, но предпочитаете стиль программирования, основанный на удаленном вызове процедур (RPC), стоит обратить внимание на gRPC. Генерация кода для gRPC надежна, поддерживается на нескольких языках и имеет отличную поддержку в .NET. Подробную информацию можно найти в документации на странице <https://learn.microsoft.com/aspnet/core/grpc>.

В главе 12 мы кратко рассмотрим новый инструмент объектно-реляционного отображения, который хорошо сочетается с ASP.NET Core: Entity Framework Core. В этой книге вы только познакомитесь с ним, но вы узнаете, как загружать и сохранять данные, создавать базу данных из своего кода и переносить ее по мере развития кода.

Резюме

- OpenAPI – это спецификация для описания HTTP API в машиночитаемом формате, как документ JSON. Можно использовать этот документ для управления другими инструментами, например генераторами кода или тестерами API;
- можно добавить генерацию документов OpenAPI в приложение ASP.NET Core, используя пакет NuGet NSwag или Swashbuckle. Эти пакеты работают со службами ASP.NET Core для чтения метаданных обо всех конечных точках вашего приложения для создания документа OpenAPI;

- по умолчанию промежуточное ПО Swashbuckle Swagger предоставляет документ OpenAPI для вашего приложения по пути `/swagger/v1/swagger.json`. Предоставление документа таким образом облегчает другим инструментам понимание конечных точек в вашем приложении;
- вы можете изучать и тестировать свой API с помощью Swagger UI. Компонент Swashbuckle Swagger UI по умолчанию предоставляет пользовательский интерфейс по пути `/swagger`. Вы можете использовать Swagger UI, чтобы исследовать свой API, отправлять тестовые запросы в конечные точки и проверять, насколько хорошо документирован ваш API;
- можно настроить описание OpenAPI своих конечных точек, добавив метаданные. Вы можете предоставить теги, например, вызвав метод `WithTags()` для конечной точки и указав, что конечная точка возвращает тип `T` с кодом состояния `201`, используя `Produces<T>(201)`. Добавление метаданных улучшает описание OpenAPI вашего API, что, в свою очередь, улучшает такие инструменты, как Swagger UI;
- вы можете использовать NSwag для создания клиента C# на основе описания OpenAPI. Этот подход обеспечивает использование правильных путей для вызова API, замену параметров в пути, а также сериализацию и десериализацию запросов к API, удаляя большую часть шаблонного кода, связанного со взаимодействием с API;
- можно добавить генерацию кода в свой проект с помощью Visual Studio или инструмента .NET API либо внося изменения в проект вручную. Visual Studio и инструмент .NET автоматизируют загрузку описания OpenAPI в локальный проект и добавление необходимых пакетов NuGet. Следует обновить пакеты NuGet до последних версий, чтобы гарантировать наличие последних исправлений ошибок и безопасности;
- NSwag автоматически генерирует имя метода C# в основном клиентском классе для каждой конечной точки в описании OpenAPI. Если `OperationID` конечной точки отсутствует, NSwag генерирует имя, которое может быть неоптимальным. Вы можете указать `OperationID`, который будет использоваться для конечной точки, в описании OpenAPI, вызвав метод `WithName()` для конечной точки;
- вы можете настроить клиента, создаваемого NSwag, добавив элемент `<Options>` внутри `<OpenApiReference>` в файле с расширением `.csproj`. Эти параметры указываются как переключатели командной строки, например `/JsonLibrary:SystemTextJson`. С помощью этих переключателей можно изменять многие вещи в сгенерированном коде, например библиотеку сериализации, которую нужно использовать, и следует ли создавать интерфейс для клиента;
- если описание OpenAPI для удаленного API изменится, необходимо снова скачать документ в свой проект, чтобы созданный клиент отразил эти изменения. Если вы изначально добавили ссылку OpenAPI с помощью Visual Studio, следует использовать Visual

Studio для обновления документа, и то же самое относится и к инструменту .NET API. NSwag автоматически обновляет сгенерированный код при изменении загруженного документа OpenAPI;

- вы можете добавить сводку и описание OpenAPI в конечную точку, установив пакет Microsoft.AspNetCore.OpenApi, вызвав метод `WithOpenApi()` для конечной точки и добавив вызовы методов `WithSummary()` или `WithDescription()`. Эти метаданные отображаются в Swagger UI, и NSwag использует сводку для создания комментариев к документации в клиенте C#;
- если вы предпочитаете, то можете использовать атрибуты вместо текущих методов для добавления метаданных OpenAPI. Такой подход иногда помогает улучшить читабельность конечных точек. Вы все равно должны вызвать метод `WithOpenApi()` для конечной точки, чтобы прочитать атрибуты метаданных;
- можно использовать комментарии к документации в формате XML для документирования своих OpenAPI, чтобы уменьшить беспорядок из-за дополнительных вызовов методов и атрибутов. Для использования этого подхода необходимо активировать создание документации для проекта, настроить Swashbuckle для чтения XML-файла документации при запуске и использовать статические методы или методы обработчика экземпляра вместо лямбда-методов;
- не все API можно описать спецификацией OpenAPI. Некоторые стили, например HATEOAS, по своей природе динамичны и не подходят для статического дизайна OpenAPI. У вас также могут возникнуть трудности со сложными требованиями аутентификации, а также с объединением документов OpenAPI. В этих случаях вы можете обнаружить, что OpenAPI не приносит никакой пользы вашему приложению.

12

Сохранение данных с Entity Framework Core

В этой главе:

- что такое библиотека Entity Framework Core и почему нужно ее использовать;
- добавление Entity Framework Core в приложение ASP.NET Core;
- построение модели данных и ее использование для создания базы данных;
- запросы, создание и обновление данных с помощью Entity Framework Core.

Большинству приложений, которые вы будете создавать с помощью ASP.NET Core, нужно будет хранить и загружать данные. Даже для тех примеров, которые применялись до сих пор в этой книге, предполагалось, что вы используете некое хранилище данных, чтобы сохранять валютные курсы, содержимое корзины пользователя или места расположения магазинов. По большей части я об этом не говорил, но обычно все эти данные хранятся в базе данных.

Работа с базами данных зачастую может представлять собой довольно неуклюжий процесс. Вам нужно управлять подключениями к базе данных, транслировать данные из своего приложения в формат, понятный базе данных, и решать кучу других нюансов. Можно делать это раз-

ными способами, но я сосредоточусь на использовании библиотеки, созданной для современной платформы .NET: Entity Framework Core (EF Core). EF Core – это библиотека, позволяющая легко и быстро создавать код для доступа к базе данных для приложений ASP.NET Core. Она создана по образцу популярной библиотеки Entity Framework 6.x, но содержит значительные изменения. Это означает, что она стоит особняком сама по себе, и это больше, чем просто апгрейд.

Цель данной главы – предоставить краткий обзор библиотеки EF Core и рассказать, как использовать ее в своих приложениях, чтобы быстро выполнять запросы к базе данных и сохранять в ней данные. Вы научитесь подключать свое приложение к базе данных и управлять изменениями схемы базы данных, но я не буду рассматривать ни одну из этих тем подробно.

ПРИМЕЧАНИЕ Для более подробного знакомства с EF Core я рекомендую книгу «Entity Framework Core в действии», 2-е изд., Джона П. Смита (Manning, 2021). Кроме того, вы можете прочитать о библиотеке EF Core на странице <https://docs.microsoft.com/ef/core/>.

Раздел 12.1 знакомит вас с библиотекой EF Core и объясняет, почему вы, возможно, захотите использовать ее в своих приложениях. Вы узнаете, как дизайн EF Core помогает быстро обработать структуру базы данных и уменьшить неприятные моменты при взаимодействии с ней.

В разделе 12.2 вы узнаете, как добавить EF Core в приложение ASP.NET Core и сконфигурировать ее с помощью системы конфигурации ASP.NET Core. Вы увидите, как создать модель для своего приложения, представляющую данные, которые вы будете хранить в базе данных, и как подключить ее к контейнеру внедрения зависимостей ASP.NET Core.

ПРИМЕЧАНИЕ В этой главе я использую SQLite, небольшой, быстрый кросс-платформенный движок базы данных, но ни один из кодов, показанных в этой главе, не предназначен специально для SQLite. Пример кода для книги также включает версию, использующую функцию SQL Server Express – LocalDB. Эта версия устанавливается как часть Visual Studio 2022 (при выборе рабочей нагрузки ASP.NET and Web Development) и предоставляет легковесное ядро SQL Server. Подробнее о LocalDB можно прочитать на странице <http://mng.bz/5jEa>.

Независимо от того, насколько тщательно вы проектируете исходную модель данных, придет время, когда вам нужно будет ее изменить. В разделе 12.3 я покажу, как с легкостью обновить свою модель и применить эти изменения к самой базе данных. При этом всю тяжелую работу за вас будет делать EF Core.

После конфигурирования EF Core и создания базы данных в разделе 12.4 показано, как использовать библиотеку в коде приложения. Вы увидите, как создавать, читать, обновлять и удалять записи, а также узнаете о паттернах, которые следует использовать при проектировании доступа к данным.

В разделе 12.5 я выделил несколько вопросов, которые следует учитывать при использовании EF Core в рабочем приложении. Одна единственная глава, посвященная EF Core, может предложить только краткое знакомство со всеми связанными с этой темой концепциями, поэтому если вы решите использовать EF Core в собственных приложениях, особенно если вы впервые применяете такую библиотеку доступа к данным, то настоятельно рекомендую прочитать дополнительную литературу, когда вы освоите основы, приведенные в этой главе.

Прежде чем перейти к коду, посмотрим, что такое EF Core, какие проблемы она решает и когда вы, возможно, захотите использовать ее.

12.1 Знакомство с Entity Framework Core

Код доступа к базе данных присутствует во всех веб-приложениях. Создаете ли вы приложение для электронной торговли, блог или очередной технологический прорыв, скорее всего, вам потребуется взаимодействовать с базой данных.

К сожалению, взаимодействие с базами данных из кода приложения часто оказывается запутанным делом, и можно использовать множество различных подходов. Например, такая простая задача, как чтение данных из базы данных, требует работы с сетевыми подключениями, написания операторов SQL и обработки данных результатов. В экосистеме .NET есть целый ряд библиотек, которые можно использовать для этого, начиная от низкоуровневых библиотек ADO.NET и заканчивая высокую абстракцией, такой как EF Core.

В этом разделе описано, что такое EF Core и какие проблемы эта библиотека призвана решать. Я расскажу о мотивации, побуждающей к использованию такой абстракции, как EF Core, и о том, как она помогает преодолеть разрыв между кодом вашего приложения и базой данных.

В рамках этого рассказа я представляю некоторые компромиссы, на которые вы можете пойти, используя ее в своих приложениях. Это должно помочь вам решить, подходит ли она вам. Наконец, мы рассмотрим пример отображения из кода приложения в базу данных, чтобы понять основные концепции EF Core.

12.1.1 Что такое EF Core?

EF Core – это библиотека, предоставляющая объектно ориентированный способ доступа к базам данных. Она действует как *инструмент объектно-реляционного отображения (ORM)*, взаимодействуя с базой данных и отображая ответы базы данных в классы и объекты .NET, как показано на рис. 12.1.

ОПРЕДЕЛЕНИЕ С помощью инструмента объектно-реляционного отображения можно управлять базой данных, используя концепции объектно ориентированного программирования, такие как классы и объекты, путем отображения их в таблицы и столбцы базы данных.

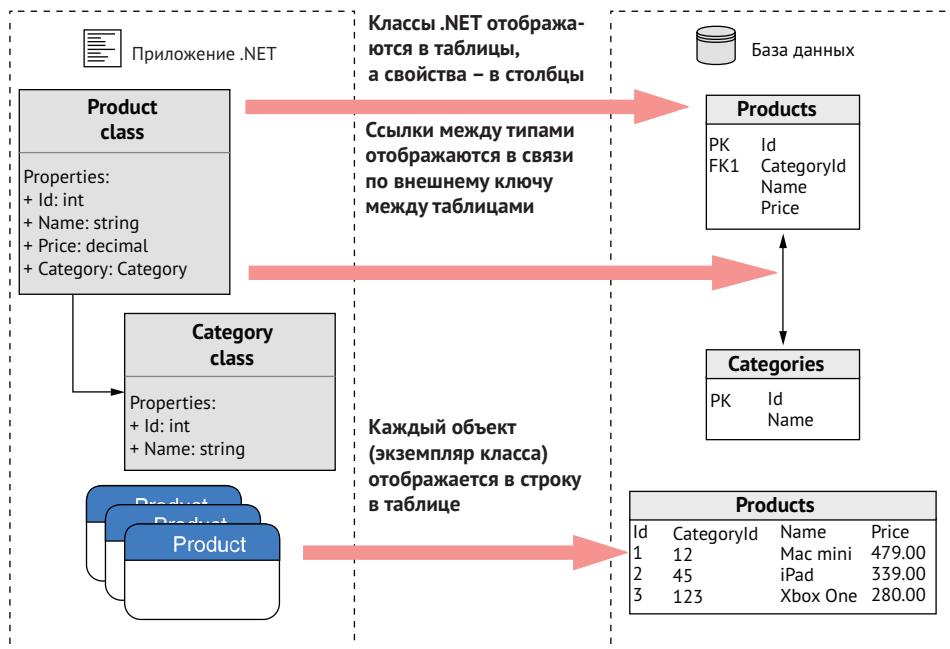


Рис. 12.1 EF Core отображает классы и объекты .NET в таблицы и строки базы данных

Библиотека EF Core основана на существующих библиотеках Entity Framework (в настоящее время до версии 6.x). Она была создана как часть .NET Core для кроссплатформенной работы, но с учетом дополнительных целей. В частности, команда EF Core хотела создать высокопроизводительную библиотеку, которую можно было бы использовать при работе с широким спектром баз данных.

Существует множество различных типов баз данных, но, вероятно, наиболее часто используемым семейством являются *реляционные* базы данных, доступ к которым осуществляется с помощью языка структурированных запросов (SQL). Это основа EF Core – библиотека может работать с Microsoft SQL Server, MySQL, Postgres и многими другими реляционными базами данных. У нее даже есть отличная функция, InMemory Provider, которую можно использовать при тестировании для создания временной базы данных. EF Core использует модель провайдера, поэтому поддержку других реляционных баз данных можно будет подключить позже, когда они станут доступны.

ПРИМЕЧАНИЕ Начиная с .NET Core 3.0 EF Core теперь также работает с нереляционными, NoSQL или документоориентированными базами данных, такими как Cosmos DB. Однако в этой книге я буду рассматривать отображение только в реляционные базы данных, поскольку это наиболее распространенный вариант, по моему опыту. Исторически сложилось так, что большая часть доступа к данным, особенно в экосистеме .NET, осуществлялась с использованием реляционных баз данных, поэтому в целом это по-прежнему наиболее популярный подход.

Это объясняет, что такое EF Core, но не отвечает на вопрос, для чего нужно применять эту библиотеку. Почему бы не получать доступ к базе данных напрямую с помощью традиционных библиотек ADO.NET? Большинство аргументов в пользу использования EF Core можно применить к ORM в целом. Так каковы же ее преимущества?

12.1.2 Зачем использовать инструмент объектно-реляционного отображения?

Одно из самых больших преимуществ ORM – это скорость, с которой вы можете разрабатывать приложение. Вы можете оставаться на знакомой территории объектно ориентированного .NET, часто даже без необходимости напрямую управлять базой данных или писать собственный SQL-код.

Представьте, что у вас есть сайт для онлайн-торговли и вы хотите загрузить подробную информацию о продукте из базы данных. Используя низкоуровневый код доступа к базе данных, вам нужно будет открыть соединение с базой данных, написать необходимый SQL-код, используя правильные имена таблиц и столбцов, прочитать данные по соединению, создать РОСО-объект для хранения данных и вручную задать свойства объекта, конвертируя данные в правильный формат по мере необходимости. Звучит неудобно, не правда ли?

Такой инструмент объектно-реляционного отображения, как EF Core, позаботится об этом за вас. Он обрабатывает соединение с базой данных, генерирует SQL-код и отображает данные на объекты РОСО. Все, что вам нужно предоставить, – это *LINQ-запрос*, описывающий данные, которые вы хотите получить.

Инструменты объектно-реляционного отображения служат высокуюровневыми абстракциями по отношению к базам данных, поэтому они могут значительно сократить объем связующего кода, который необходимо написать для взаимодействия с базой данных. На самом базовом уровне они заботятся об отображении SQL-операторов в объекты и наоборот, но большинство из них идут еще дальше и предоставляют дополнительные функции.

Такие ORM-инструменты, как EF Core, отслеживают, какие свойства были изменены у объектов, которые они получают из базы данных. Это позволяет загружать объект из базы данных, отображая его из таблицы базы данных, изменить его в коде .NET, а затем попросить ORM-инструмент обновить ассоциированную запись в базе данных. Он определит, какие свойства изменились, и предоставит операторы обновления для соответствующих столбцов, сэкономив вам кучу работы.

Как это часто бывает при разработке программного обеспечения, использование такого рода инструментов имеет свои недостатки. Одно из самых больших преимуществ ORM-инструментов (а также их ахиллесова пятна) – они скрывают от вас базу данных. Иногда такой высокий уровень абстракции может привести к возникновению проблемных шаблонов запросов к базе данных в приложениях. Классический пример – проблема *N + 1*, когда один запрос к базе данных превращается в отдельные запросы для каждой отдельной строки в таблице базы данных.

Еще один часто упоминаемый недостаток – производительность. ORM-инструменты – это абстракции нескольких концепций, поэтому по своей сути они выполняют больше работы, чем если бы вы вручную создавали каждую часть доступа к данным в своем приложении. Большинство ORM-инструментов, включая EF Core, жертвуют производительностью ради простоты разработки.

Тем не менее если вы знаете об этих подводных камнях, то нередко можете значительно упростить код, необходимый для взаимодействия с базой данных. Как и во всех остальных ситуациях, если абстракция вам подходит, используйте ее, в противном случае не делайте этого. Если у вас минимальные требования к доступу к базе данных или вам нужна максимальная производительность, то возможно, что такой вариант, как EF Core, вам не подойдет.

Альтернативный способ – получить лучшее из обоих миров: использовать ORM-инструмент для быстрой разработки основной части своего приложения, а затем вернуться к низкоуровневым API, таким как ADO.NET, для тех немногих областей, которые оказываются проблемными местами в приложении. Таким образом, можно получить достаточно хорошую производительность, используя EF Core и жертвуя производительностью ради времени разработки, оптимизируя только те области, которые в этом нуждаются.

ПРИМЕЧАНИЕ В наши дни аспект производительности является одним из самых слабых аргументов против ORM-инструментов. EF Core использует множество приемов работы с базами данных и создает чистые SQL-запросы, поэтому если вы не являетесь экспертом по базам данных, то можете обнаружить, что она превосходит даже ваши запросы ADO.NET, созданные вручную!

Даже если вы решите применять ORM-инструмент в своем приложении, для .NET доступно множество различных вариантов, одним из которых является EF Core. Подходит ли он вам, будет зависеть от необходимых вам функций и компромиссов, на которые вы готовы пойти, чтобы их получить. В следующем разделе EF Core сравнивается с другим предложением от компании Microsoft, Entity Framework, но есть много иных альтернатив, которые можно рассмотреть, например Dapper и NHibernate. У каждой из них есть свой набор компромиссов.

12.1.3 Когда следует выбирать EF Core?

Компания Microsoft разработала EF Core как переосмысление зрелой технологии Entity Framework 6.x (EF 6.x), появившейся в 2008 году. После многих лет разработки EF 6.x представляет собой стабильное и многофункциональное средство объектно-реляционного отображения, но больше не находится в активной разработке.

EF Core, выпущенная в 2016 г., – сравнительно новый проект. API EF Core разработаны так, чтобы быть близкими к API EF 6.x (хотя они и не идентичны), но основные компоненты были полностью пе-

реписаны. Следует считать, что EF Core отличается от EF 6.x; обновление непосредственно с EF 6.x до EF Core нетривиально.

Хотя Microsoft поддерживает как EF Core, так и EF 6.x, EF 6.x не рекомендуется для новых приложений .NET. В настоящее время нет особых причин запускать новое приложение с EF 6.x, но точные компромиссы будут во многом зависеть от конкретного приложения. Если вы решите выбрать EF 6.x вместо EF Core, убедитесь, что вы понимаете, чем жертвуете. Также обязательно следите за рекомендациями и сравнением функций от команды EF на странице <http://mng.bz/GxgA>.

Если вы решите использовать ORM-инструмент для своего приложения, то EF Core – отличный кандидат. Она также поддерживается «из коробки» различными другими подсистемами ASP.NET Core. В главе 23 вы узнаете, как использовать EF Core с системой аутентификации ASP.NET Core Identity для управления пользователями в ваших приложениях.

Прежде чем я перейду к подробностям применения EF Core в приложении, я опишу приложение, которое мы будем использовать, в качестве примера для этой главы. Я расскажу о приложении и базе данных, и мы обсудим, как использовать EF Core для обмена данных.

12.1.4 Отображение базы данных в код приложения

EF Core фокусируется на обмене данными между приложением и базой данных, поэтому, чтобы продемонстрировать это, нам нужно приложение. В этой главе в качестве примера используется простое кулинарное приложение, в котором перечислены рецепты и которое позволяет просматривать ингредиенты рецепта, как показано на рис. 12.2. Пользователи могут просматривать рецепты, добавлять новые, редактировать их и удалять старые.

Очевидно, что это простое приложение, но оно содержит все необходимые нам взаимодействия между базой данных и двумя его *сущностями*: Recipe и Ingredient.

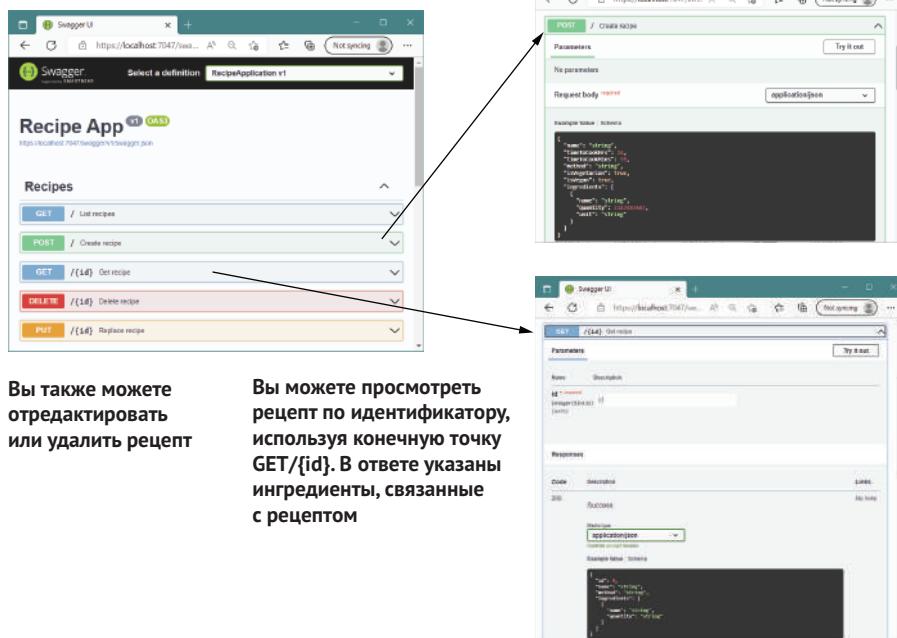
ОПРЕДЕЛЕНИЕ *Сущность* – это класс .NET, отображаемый EF Core в базу данных. Это классы, которые вы определяете обычно как классы РОСО, которые можно сохранять и загружать путем отображения в таблицы базы данных с помощью EF Core.

При взаимодействии с EF Core вы в основном будете использовать сущности РОСО и контекст базы данных, производный от класса EF Core, DbContext. Классы сущностей – это объектно ориентированные представления таблиц в базе данных; они представляют данные, которые вы хотите сохранить в базе данных. Класс DbContext используется в приложении как для конфигурирования EF Core, так и для доступа к базе данных во время выполнения.

ПРИМЕЧАНИЕ Потенциально можно иметь несколько классов DbContext в своем приложении и даже можно сконфигурировать их для интеграции с различными базами данных.

Приложение рецептов предоставляет API для взаимодействия с рецептами

Используйте конечную точку POST / для создания рецепта



Вы также можете отредактировать или удалить рецепт

Вы можете просмотреть рецепт по идентификатору, используя конечную точку GET/{id}. В ответе указаны ингредиенты, связанные с рецептом

Рис. 12.2 В кулинарном приложении перечислены рецепты. Вы можете просматривать, обновлять и удалять их или создавать новые

Когда приложение впервые использует EF Core, EF Core создает внутреннее представление базы данных на основе свойств DbSet<T> класса DbContext приложения и самих классов сущностей, как показано на рис. 12.3.

Для приложения с рецептами EF Core создаст модель класса Recipe, поскольку она представлена в классе AppDbContext как DbSet<Recipe>. Кроме того, EF Core будет перебирать все свойства Recipe, искать типы, о которых не знает, и добавлять их в свою внутреннюю модель. В приложении коллекция Ingredients в Recipe предоставляет сущность Ingredient как ICollection<Ingredient>, поэтому EF Core моделирует сущность соответствующим образом.

Каждая сущность отображается в таблице в базе данных, но EF Core также отображает связи между сущностями. В каждом рецепте может быть много ингредиентов, но каждый ингредиент (у которого есть имя, количество и единица измерения) принадлежит одному рецепту, поэтому такой тип связи называется «многие к одному». EF Core использует эти сведения для правильного моделирования эквивалентной структуры базы данных.

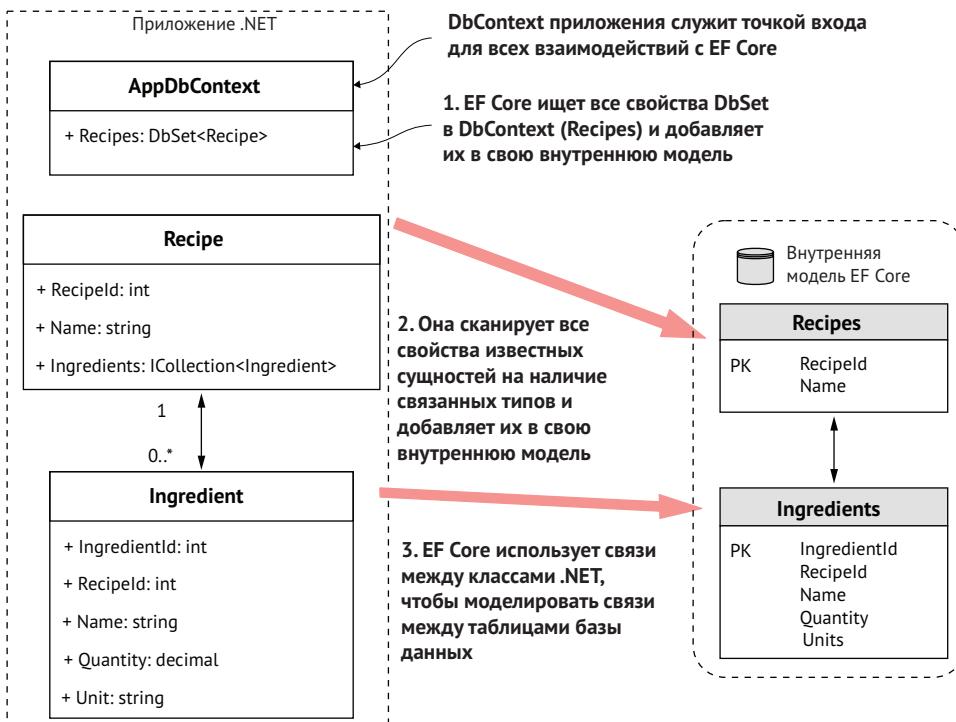


Рис. 12.3 EF Core создает внутреннюю модель модели данных вашего приложения, исходя из типов в коде. Она добавляет все типы, указанные в свойствах `DbSet<T>` класса `DbContext`, и все связанные типы

ПРИМЕЧАНИЕ Два разных рецепта, скажем рыбный пирог и курица с лимоном, могут использовать ингредиент с одинаковым названием и количеством, например сок одного лимона, но, по сути, это два разных экземпляра. Если вы обновите рецепт курицы с лимоном, чтобы использовать два лимона, вы не захотите, чтобы это изменение автоматически привело к обновлению рецепта рыбного пирога, чтобы в нем тоже использовалось два лимона!

EF Core использует внутреннюю модель, которую создает при взаимодействии с базой данных. Это гарантирует построение правильно го SQL-кода для создания, чтения, обновления и удаления сущностей.

Хорошо, пора писать код! В следующем разделе мы приступим к созданию приложения с рецептами. Вы увидите, как добавить EF Core в приложение ASP.NET Core, сконфигурировать провайдер базы данных и спроектировать модель данных приложения.

12.2 Добавляем EF Core в приложение

В этом разделе мы сосредоточимся на установке и настройке EF Core в приложении рецептов. Вы узнаете, как установить необходимые пакеты NuGet и как создать модель данных для своего приложения. По-

скольку в этой главе мы говорим об EF Core, я не буду вдаваться в подробности создания приложения в целом – в качестве основы я создал простое приложение с минимальным API – ничего особенного.

СОВЕТ Образец кода для этой главы показывает состояние приложения на трех разных этапах: в конце раздела 12.2, в конце раздела 12.3 и в конце главы. Он также включает в себя примеры с использованием провайдеров LocalDB и SQLite.

Взаимодействие с EF Core в приложении, использующееся в качестве примера, происходит на уровне сервисов, который инкапсулирует весь доступ к данным за пределами обработчиков конечных точек минимальных API, как показано на рис. 12.4. Это позволяет разделить задачи и сделать сервисы доступными для тестирования. Процесс добавления EF Core в приложение состоит из нескольких этапов:

- 1 Выбрать провайдера базы данных, например Postgres, SQLite или MS SQL Server.
- 2 Установить пакеты NuGet для EF Core.
- 3 Спроектировать класс DbContext своего приложения и сущности, составляющие модель данных.
- 4 Зарегистрировать этот класс в контейнере внедрения зависимостей ASP.NET Core.
- 5 Использовать EF Core для создания миграции, описывающей модель данных.
- 6 Применить миграцию к базе данных, чтобы обновить схему базы данных.

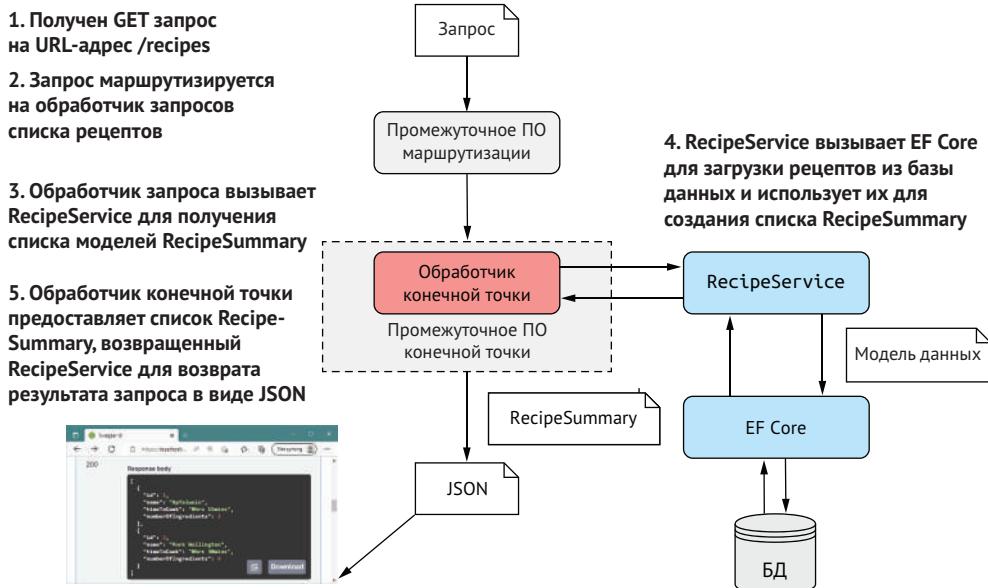


Рис. 12.4 Обработка запроса путем загрузки данных из базы данных с помощью EF Core. Взаимодействие с EF Core ограничено только RecipeService – страница Razor не обращается к EF Core напрямую

Это уже может показаться вам немного сложным, но мы рассмотрим шаги 1–4 в этом разделе и шаги 5–6 в разделе 12.3, поэтому это не займет много времени. Учитывая ограниченное пространство данной главы, я буду придерживаться соглашений EF Core по умолчанию в коде, который показываю. EF Core – гораздо более настраиваемая библиотека, чем может показаться на первый взгляд, но я рекомендую вам по возможности придерживаться значений по умолчанию. В конечном итоге это облегчит вам жизнь.

Первый шаг в настройке EF Core – решить, с какой базой данных вы хотите взаимодействовать. Вполне вероятно, что это будет проектировано клиентом или политикой вашей компании, но все же стоит подумать о выборе.

12.2.1 Выбор провайдера базы данных и установка EF Core

EF Core поддерживает ряд баз данных с помощью модели провайдера. Модульная природа EF Core означает, что вы можете использовать один и тот же высокоуровневый API для программирования с различными базами данных, а EF Core знает, как сгенерировать необходимый код, зависящий от реализации, и операторы SQL.

Вероятно, у вас уже есть на примете база данных, когда вы запускаете свое приложение, и вам будет приятно узнать, что EF Core работает с большинством популярных баз данных. Добавление поддержки базы данных включает в себя добавление правильного пакета NuGet в файл с расширением .csproj. Например:

- PostgreSQL – Npgsql.EntityFrameworkCore.PostgreSQL;
- Microsoft SQL Server – Microsoft.EntityFrameworkCore.SqlServer;
- MySQL – MySql.Data.EntityFrameworkCore;
- SQLite – Microsoft.EntityFrameworkCore.SQLite.

Компания Microsoft занимается сопровождением некоторых пакетов провайдеров баз данных, часть из них сопровождается сообществом разработчиков ПО с открытым исходным кодом, а для каких-то пакетов может потребоваться платная лицензия (например, для провайдера Oracle), поэтому обязательно проверьте свои требования. Список провайдеров можно найти на странице <https://docs.microsoft.com/ef/core/providers>.

Провайдер базы данных устанавливается в приложение так же, как и любая другая библиотека: путем добавления пакета NuGet в файл с расширением .csproj из вашего проекта и выполнения команды `dotnet restore` из командной строки (или можно позволить Visual Studio выполнить восстановление автоматически).

EF Core по своей сути является модульной библиотекой, поэтому вам потребуется установить несколько пакетов. Я использую провайдера базы данных SQL Server с LocalDB для приложения с рецептами, поэтому буду применять пакеты SQL Server:

- *Microsoft.EntityFrameworkCore.SqlServer* – это основной пакет провайдера базы данных для использования EF Core во время выполнения. Он также содержит ссылку на основной пакет EF Core NuGet;

- *Microsoft.EntityFrameworkCore.Design* – содержит совместно используемые компоненты времени проектирования для EF Core.

СОВЕТ Вам также понадобится установить инструменты командной строки, которые помогут вам создавать и обновлять базу данных. Я покажу, как их установить, в разделе 12.3.1.

В листинге 12.1 показан файл приложения с рецептами с расширением .csproj после добавления пакетов EF Core. Помните, что вы добавляете пакеты NuGet как элементы PackageReference.

Листинг 12.1. Установка EF Core в приложении ASP.NET Core

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

    <PropertyGroup>
        <TargetFramework>net7.0</TargetFramework> ← Приложение предна-
        <Nullable>enable</Nullable> значено для .NET 7.0
        <ImplicitUsings>enable</ImplicitUsings>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference
            Include="Microsoft.EntityFrameworkCore.SQLite"
            Version="7.0.0" /> Устанавливаем соот-
        <PackageReference
            Include="Microsoft.EntityFrameworkCore.Design"
            Version="7.0.0" > ветствующий пакет
            <IncludeAssets>runtime; build; native; contentfiles;
            Analyzers; buildtransitive</IncludeAssets> NuGet для выбранной
            <PrivateAssets>all</PrivateAssets> вами базы данных
        </PackageReference> Содержит совместно
    </ItemGroup> используемые компо-
        <!-- Добавляется
        ненты времени проек-
        тирования для EF Core
        тируется для EF Core
        автоматически
        NuGet -->
```

После установки и восстановления этих пакетов у нас есть все необходимое, чтобы приступить к созданию модели данных для приложения. В следующем разделе мы создадим классы сущностей и DbContext для приложения с рецептами.

12.2.2 Создание модели данных

В разделе 12.1.4 я привел обзор того, как EF Core создает внутреннюю модель базы данных из класса DbContext и моделей сущностей. Помимо этого механизма обнаружения, EF Core – довольно гибкая библиотека, позволяющая определять сущности так, как вы хотите, как классы POCO.

Некоторые инструменты объектно-реляционного отображения требуют, чтобы сущности наследовали от определенного базового класса, или вы должны декорировать свои модели атрибутами, чтобы описать, как их отображать. EF Core в значительной степени отдает предпочтение подходу с использованием соглашений, а не конфигу-

рации, как видно в листинге 12.2. В нем показаны классы сущностей `Recipe` и `Ingredient` нашего приложения.

СОВЕТ Ключевое слово `required`, использованное в нескольких свойствах в листинге 12.2, было введено в C# 11. Здесь оно используется для предотвращения предупреждений о неинициализированных значениях, не допускающих значения `NULL`. Подробнее о том, как EF Core взаимодействует с типами, не допускающими значения `NULL`, можно прочитать в документации на странице <http://mng.bz/Keoj>.

Листинг 12.2. Определение классов сущностей EF Core

```
public class Recipe
{
    public int RecipeId { get; set; }
    public required string Name { get; set; }
    public TimeSpan TimeToCook { get; set; }
    public bool IsDeleted { get; set; }
    public required string Method { get; set; }
    public required ICollection<Ingredient> Ingredients { get; set; } ◀
}
public class Ingredient
{
    public int IngredientId { get; set; }
    public int RecipeId { get; set; }
    public required string Name { get; set; }
    public decimal Quantity { get; set; }
    public required string Unit { get; set; }
}
```

В классе `Recipe` может быть множество ингредиентов, представленных `ICollection`

Эти классы соответствуют определенным соглашениям по умолчанию, которые EF Core использует для создания картины отображаемой базы данных. Например, у класса `Recipe` есть свойство `RecipeId`, а у класса `Ingredient` – свойство `IngredientId`. EF Core определяет этот шаблон суффикса `Id` как указание на *первичный ключ* таблицы.

ОПРЕДЕЛЕНИЕ *Первичный ключ* таблицы – это значение, которое однозначно идентифицирует строку среди всех остальных в таблице. Часто это `int` или `Guid`.

Еще одно соглашение, которое мы видим здесь, – это свойство `RecipeId` класса `Ingredient`. EF Core интерпретирует его как *внешний ключ*, указывающий на класс `Recipe`. В сочетании с `ICollection<Ingredient>` в классе `Recipe` он представляет связь типа «многие к одному», где у каждого рецепта имеется множество ингредиентов, но каждый ингредиент принадлежит только одному рецепту, как показано на рис. 12.5.

ОПРЕДЕЛЕНИЕ *Внешний ключ* в таблице указывает на первичный ключ другой таблицы, образуя связь между двумя строками.

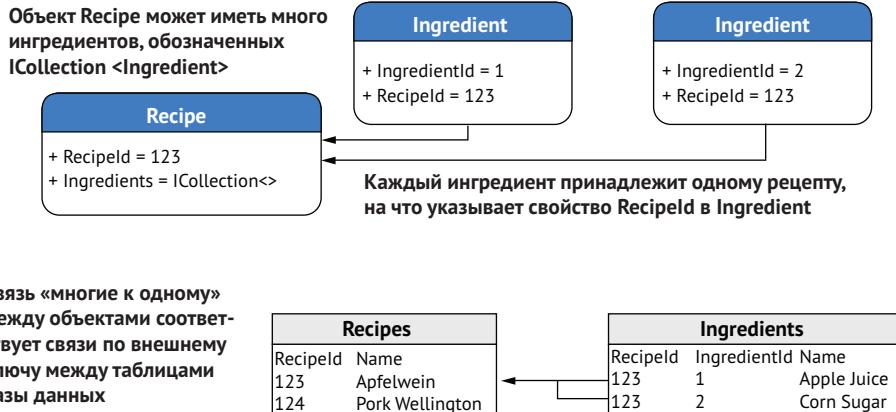


Рис. 12.5 Связь типа «многие к одному» в коде преобразуется в связь по внешнему ключу между таблицами

Здесь задействованы и многие другие соглашения, такие как имена, которые EF Core будет использовать для таблиц и столбцов базы данных, или типы столбцов базы данных, которые будут использоваться для каждого свойства, но я не буду обсуждать их здесь. В документации по EF Core содержится подробная информация обо всех соглашениях, а также о том, как настроить их для своего приложения: <https://docs.microsoft.com/ef/core/modeling/>.

СОВЕТ Вы также можете использовать атрибуты `DataAnnotations` для декорирования классов сущностей, управляя такими вещами, как именование столбцов или длина строки. EF Core будет использовать эти атрибуты, чтобы переопределить соглашения по умолчанию.

Помимо сущностей, вы также определяете класс `DbContext`. Это сердце вашего приложения, используемое для всех вызовов к базе данных. Создайте собственный класс `DbContext`, в данном случае это `AppDbContext`, и наследуйте его от базового класса `DbContext`, как показано в листинге 12.3. Он предоставляет `DbSet<Recipe>`, позволяющий EF Core обнаружить и отобразить сущность `Recipe`. Таким образом вы можете предоставить несколько экземпляров `DbSet<>` для каждой из сущностей верхнего уровня своего приложения.

Листинг 12.3. Определение класса `DbContext`

```

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options) { }

    public DbSet<Recipe> Recipes { get; set; }
}

```

Объект параметров конструктора, содержащий такие сведения, как строка подключения

Вы будете использовать свойство `Recipes` для выполнения запроса к базе данных

`AppDbContext` – это простой класс, содержащий список корневых сущностей, но вы можете делать с ним гораздо больше вещей в более сложном приложении. При желании можно полностью настроить способ отображения сущностей в базу данных, но для этого приложения мы будем использовать значения по умолчанию.

ПРИМЕЧАНИЕ Мы не указали класс `Ingredient` в классе `AppDbContext`, но он будет смоделирован EF Core, как он предоставляется в `Recipe`. Вы по-прежнему можете получить доступ к объектам `Ingredient` в базе данных, но для этого нужно использовать свойство `Ingredients` сущности `Recipe`, как будет показано в разделе 12.4.

В этом простом примере ваша модель данных состоит из трех классов: `AppDbContext`, `Recipe` и `Ingredient`. Эти две сущности будут отображены в таблицы, а их столбцы – в свойства, и вы будете использовать класс `AppDbContext` для доступа к ним.

ПРИМЕЧАНИЕ Это типичный подход `code-first` (сначала код), но если у вас есть действующая база данных, можно автоматически сгенерировать сущности EF и класса `DbContext`. (Более подробную информацию можно найти в статье Microsoft «Реверс-инжиниринг» на странице <http://mng.bz/mgd4>.)

Модель данных завершена, но мы еще не совсем готовы ее использовать. Приложение не знает, как создать `AppDbContext`, а `AppDbContext` нужна строка подключения, чтобы можно было взаимодействовать с базой данных. В следующем разделе мы рассмотрим обе эти проблемы и закончим настройку EF Core в приложении.

12.2.3 Регистрация контекста данных

Как и в случае с любым другим сервисом в ASP.NET Core, нужно зарегистрировать `AppDbContext` в контейнере внедрения зависимостей. При регистрации контекста вы также конфигурируете провайдера базы данных и задаете строку подключения, чтобы EF Core знала, как взаимодействовать с базой данных.

`AppDbContext` регистрируется в `WebApplicationBuilder` файла `Program.cs`. EF Core предоставляет для этой цели обобщенный метод расширения `AddDbContext<T>`, который принимает функцию конфигурации для экземпляра `DbContextOptionsBuilder`. Этот конструктор можно использовать, чтобы задать множество внутренних свойств EF Core. При желании он позволяет полностью заменить внутренние сервисы EF Core.

Конфигурация вашего приложения, опять же, простая и удобная, как видно из следующего листинга. Вы задаете провайдера базы данных с помощью метода расширения `UseSqlServer`, доступного из пакета `Microsoft.EntityFrameworkCore.SqlServer`, и передаете ему строку подключения.

ПРИМЕЧАНИЕ Если вы используете другого провайдера базы данных, например провайдера для SQLite, то нужно будет вызвать соответствующий метод `Use*` объекта `options` при регистрации `AppDbContext`.

Листинг 12.4 Регистрация DbContext в контейнере внедрения зависимостей

```
using Microsoft.EntityFrameworkCore;
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
var connString = builder.Configuration
    .GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<AppDbContext>(←
    options => options.UseSqlite(connString));
```

Указываем провайдера базы данных в параметрах настройки для DbContext

```
WebApplication app = builder.Build();
app.Run();
```

Строка подключения берется из конфигурации, из раздела ConnectionStrings

Регистрируем DbContext вашего приложения, используя его в качестве обобщенного параметра

Строка подключения обычно является секретом, как мы уже обсуждали в предыдущей главе, поэтому имеет смысл загрузить ее из конфигурации. Во время выполнения будет использоваться правильная строка конфигурации для вашего текущего окружения, поэтому вы можете использовать разные базы данных при локальной разработке или в промышленном окружении.

СОВЕТ Можно настроить AppDbContext другими способами и предоставить строку подключения, например с помощью метода `OnConfiguring`, но я рекомендую способ, показанный здесь для веб-сайтов ASP.NET Core.

Теперь у вас есть `DbContext`, `AppDbContext`, зарегистрированный в контейнере внедрения зависимостей (типично для служб, связанных с базами данных), и модель данных, соответствующая вашей базе данных. Вы готовы приступить к использованию EF Core, но единственное, чего у вас *нет*, – это базы данных! В следующем разделе вы увидите, как с легкостью можно использовать интерфейс командной строки .NET, чтобы гарантировать актуальность своей базы данных с помощью модели данных EF Core.

12.3 Управление изменениями с помощью миграций

В этом разделе вы узнаете, как с помощью миграций генерировать SQL-код, чтобы синхронизировать схему базы данных с моделью данных приложения. Вы узнаете, как создать начальную миграцию и использовать ее для разработки базы данных. Затем вы обновите свою модель данных, создадите вторую миграцию и воспользуетесь ею для обновления схемы базы данных.

Известно, что управление изменениями схемы баз данных, например когда вам нужно добавить новую таблицу или новый столбец, является сложной задачей. Код вашего приложения явно привязан к конкретной версии базы данных, и нужно убедиться, что они всегда синхронизированы.

ОПРЕДЕЛЕНИЕ Схема – это способ организации данных в базе данных, в том числе таблиц, столбцов и связей между ними.

При развертывании приложения вы обычно можете удалить старый код или исполняемый файл и заменить его новым кодом – работа сделана. Если вам нужно отменить изменение, удалите новый код и разверните старую версию приложения.

Сложность с базами данных состоит в том, что они содержат данные. Это означает, что невозможно пренебречь этим, создавая новую базу данных при каждом развертывании.

Существует общепринятая передовая практика, которая состоит в том, чтобы явно версионировать схему базы данных наряду с кодом приложения. Это можно сделать несколькими способами, но обычно нужно сохранить разницу между предыдущей схемой и новой, часто в виде SQL-скрипта. Затем можно использовать такие библиотеки, как DbUp (<https://github.com/DbUp/DbUp>) и FluentMigrator (<https://github.com/fluentmigrator/fluentmigrator>), чтобы отслеживать, какие скрипты были применены, и обеспечивать актуальность схемы своей базы данных. Кроме того, можно использовать внешние инструменты, чтобы они делали это за вас.

EF Core предоставляет собственную версию управления схемой, которую называют *миграциями*. Миграции позволяют управлять изменениями схемы базы данных при изменении модели данных EF Core.

ОПРЕДЕЛЕНИЕ Миграция – это файл с кодом C# в приложении, который определяет, как изменилась модель данных – какие столбцы были добавлены, новые сущности и т. д. Миграции обеспечивают запись того, как развивалась схема вашей базы данных, будучи частью вашего приложения, поэтому схема всегда синхронизируется с моделью данных приложения.

Можно использовать инструменты командной строки, чтобы создать новую базу данных из миграций или обновить существующую базу данных, применив к ней новые миграции. Вы даже можете откатить миграцию, в результате чего база данных обновится до предыдущей схемы.

ВНИМАНИЕ! Применение миграций изменяет базу данных, поэтому всегда нужно помнить о потере данных. Если вы удалите таблицу из базы данных с помощью миграции, а затем выполните откат, то таблица будет воссоздана заново, но данные, которые ранее содержались в ней, исчезнут навсегда!

В этом разделе вы увидите, как создать первую миграцию и использовать ее для создания базы данных. Затем вы обновите свою модель данных, создадите вторую миграцию и воспользуетесь ею для обновления схемы базы данных.

12.3.1 Создаем первую миграцию

Прежде чем вы сможете создавать миграции, нужно установить необходимые инструменты. Это можно сделать двумя основными способами:

- консоль диспетчера пакетов – командлеты PowerShell можно использовать в консоли диспетчера пакетов Visual Studio. Вы можете установить их напрямую из консоли или добавить в свой проект пакет Microsoft.EntityFrameworkCore.Tools;
 - инструменты командной строки .NET – кросс-платформенный набор инструментов, которые можно запускать из командной строки и который расширяет набор средств разработки .NET. Вы можете установить эти инструменты глобально на свой компьютер, выполнив команду `dotnet tool install --global dotnet-e`.

В этой книге я буду использовать кроссплатформенные инструменты командной строки .NET, но если вы знакомы с EF 6.x или предпочитаете использовать консоль диспетчера пакетов Visual Studio, существуют эквивалентные команды для всех этих шагов (<http://mng.bz/9DK7>). Проверить правильность установки инструмента .NET можно, выполнив команду `dotnet ef`. Должен появиться экран справки, похожий на тот, что показан на рис. 12.6.

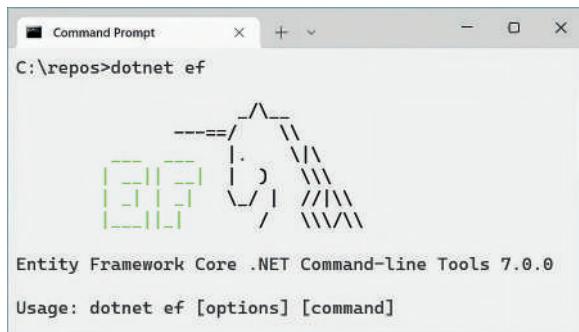


Рис. 12.6 Выполняем команду dotnet ef, чтобы проверить правильность установки инструментов .NET EF Core

СОВЕТ Если при выполнении предыдущей команды вы получили сообщение `No executable found matching command 'dotnet-ef'` (Не найден исполняемый файл, соответствующий команде 'dotnetef'), убедитесь, что вы установили глобальный инструмент EF Core с помощью команды `dotnet tool install --global dotnet-ef`. Как правило, необходимо выполнять команду `dotnet ef` из папки проекта, в которой вы зарегистрировали `AppDbContext` (не на уровне папки решения).

Установив инструменты EF Core и сконфигурировав контекст базы данных, можно создать свою первую миграцию, выполнив следующую команду из папки веб-проекта и указав имя миграции – в данном случае `InitialSchema`:

```
dotnet ef migrations add InitialSchema
```

Эта команда создает три файла в папке Migrations:

- *файл миграции* – файл в формате `Timestamp_MigrationName.cs`. Здесь описаны действия, которые нужно предпринять в базе дан-

ных, такие как создание таблицы или добавление столбца. Обратите внимание, что сгенерированные здесь команды зависят от провайдера базы данных, основываясь на провайдере, сконфигурированном в вашем проекте;

- файл *Migration designer.cs* – этот файл описывает внутреннюю модель вашей модели данных EF Core на момент создания миграции;
- *AppDbContextModelSnapshot.cs* – описывает текущую внутреннюю модель EF Core. Этот файл обновляется при добавлении другой миграции, поэтому всегда должен быть таким же, как текущая (последняя) миграция. EF Core может использовать его для определения предыдущего состояния базы данных при создании новой миграции без прямого взаимодействия с базой данных.

Эти три файла инкапсулируют процесс миграции, но при добавлении миграции в самой базе данных ничего не обновляется. Для этого нужно выполнить другую команду, чтобы применить миграцию к базе данных.

СОВЕТ Вы можете и должны заглянуть внутрь файла миграции, созданного EF Core, чтобы проверить, что он будет делать с вашей базой данных, прежде чем запускать следующие команды. Береженого Бог бережет!

Можно применить миграции одним из четырех способов:

- используя инструмент командной строки .NET;
- используя командлеты PowerShell;
- в коде, получив экземпляр класса *AppDbContext* из контейнера внедрения зависимостей и вызвав метод *context.Database.Migrate()*;
- создав приложение пакета миграции (см. <http://mng.bz/jPyr>).

Какой вариант лучше всего вам подходит, зависит от того, как вы спроектировали свое приложение, как будете обновлять рабочую базу данных, и от ваших личных предпочтений. Пока я буду использовать инструмент командной строки .NET, но некоторые из этих соображений рассмотрю в разделе 12.5. Можно применить миграции к базе данных, выполнив команду

```
dotnet ef database update
```

из папки проекта своего приложения. Я не буду вдаваться в подробности того, как она работает, но эта команда выполняет четыре шага.

- 1 Выполняет сборку приложения.
- 2 Загружает сервисы, настроенные в файле приложения, *Program.cs*, включая *AppDbContext*.
- 3 Проверяет, существует ли база данных в строке подключения *AppDbContext*. Если ее там нет, она ее создает.
- 4 Обновляет базу данных, применяя все миграции, которые не были применены.

Если все настроено правильно, как я показал в разделе 12.2, то после выполнения этой команды у вас будет установлена новая база данных, как та, что показана на рис. 12.7.

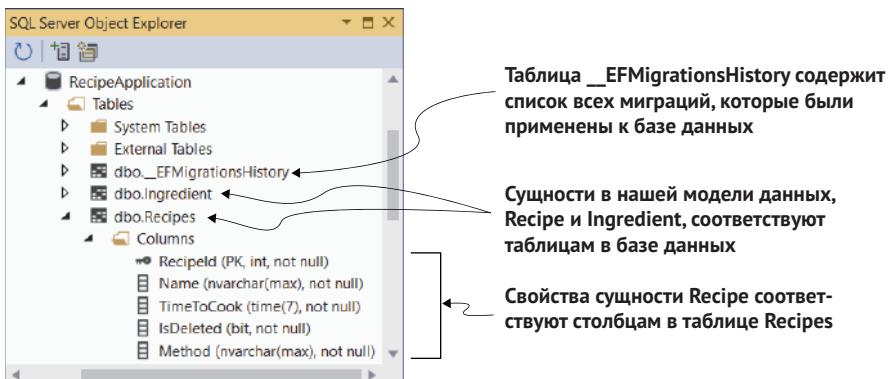


Рис. 12.7 Применение миграции к базе данных приведет к созданию базы данных, если она не существует, и обновлению базы данных в соответствии с внутренней моделью данных EF Core. Список примененных миграций хранится в таблице **_EFMigrationsHistory**

ПРИМЕЧАНИЕ Если при выполнении этих команд вы получаете сообщение об ошибке «Проект не найден», убедитесь, что вы запускаете их в папке проекта приложения, а не в папке решения верхнего уровня.

Когда вы применяете миграции к базе данных, EF Core создает необходимые таблицы в базе данных и добавляет соответствующие столбцы и ключи. Возможно, вы также заметили таблицу **_EFMigrationsHistory**. EF Core использует ее для хранения имен миграций, примененных к базе данных. В следующий раз, когда вы выполните команду `dotnet ef database update`, EF Core сможет сравнить эту таблицу со списком миграций в приложении и применит к базе данных только новые миграции. В следующем разделе мы рассмотрим, как это упрощает изменение модели данных и обновление схемы базы данных без необходимости воссоздавать базу данных с нуля.

12.3.2 Добавляем вторую миграцию

Большинство приложений неизбежно развиваются, будь то из-за увеличения объема или сопровождения. Добавление свойств к сущностям, добавление новых сущностей целиком и удаление устаревших классов – все это вполне вероятные действия.

Миграции EF Core упрощают эти процессы. Представьте, что вы решили выделить вегетарианские и веганские блюда в своем приложении с рецептами, предоставив свойства `IsVegetarian` и `IsVegan` в сущности `Recipe` (листинг 12.5). Переведите сущности в желаемое состояние, сгенерируйте миграцию и примените ее к базе данных, как показано на рис. 12.8.

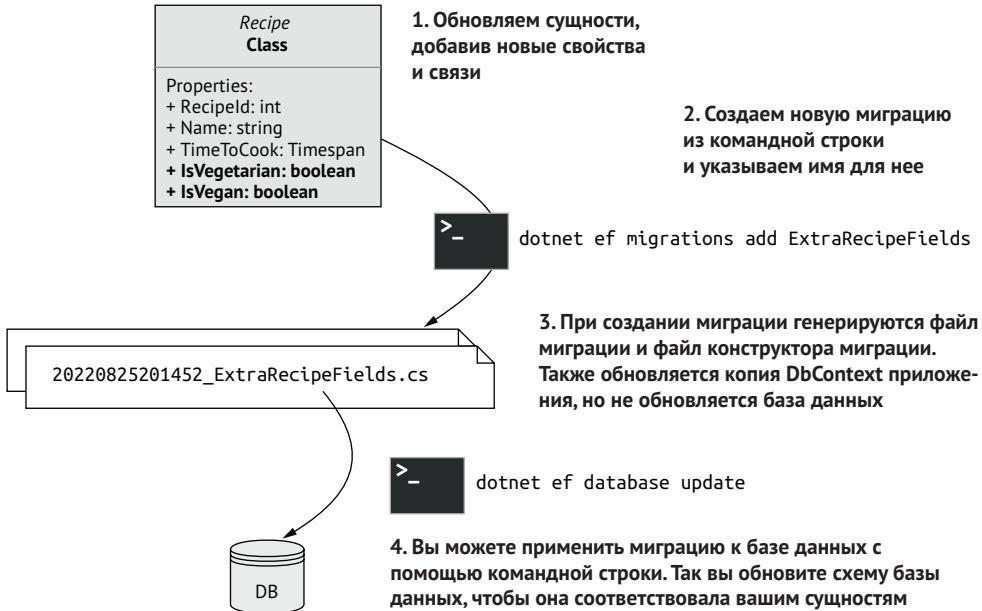


Рис. 12.8 Создание второй миграции и применение ее к базе данных с помощью инструментов командной строки

Листинг 12.5 Добавляем свойства в сущность Recipe

```

public class Recipe
{
    public int RecipeId { get; set; }
    public required string Name { get; set; }
    public TimeSpan TimeToCook { get; set; }
    public bool IsDeleted { get; set; }
    public required string Method { get; set; }
    public bool IsVegetarian { get; set; }
    public bool IsVegan { get; set; }
    public required ICollection<Ingredient> Ingredients { get; set; }
}
  
```

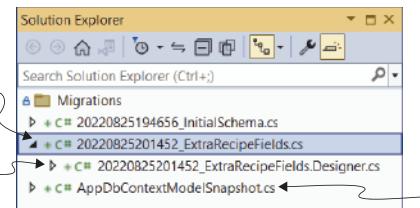
Как показано на рис. 12.8, после изменения сущностей необходимо обновить внутреннее представление модели данных. Делается это точно так же, как и при первой миграции, путем вызова команды `dotnet ef migrations add` и предоставления имени миграции:

```
dotnet ef migrations add ExtraRecipeFields
```

Так вы создаете вторую миграцию в своем проекте путем добавления файла миграции и файла копии файла `.designer.cs` и обновления файла `AppDbContextModelSnapshot.cs`, как показано на рис. 12.9.

При создании миграции в ваше решение добавляется файл с расширением .cs с временной меткой и именем, которое вы дали миграции

А также добавляется файл Designer.cs, содержащий копию внутренней модели данных EF Core на тот момент времени



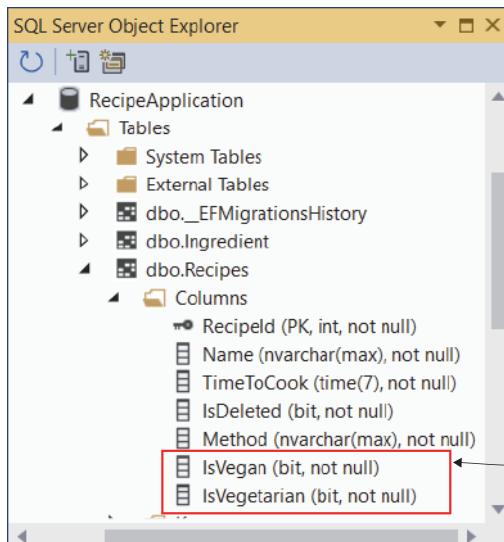
AppDbContextModelSnapshot обновляется, чтобы соответствовать копии для новой миграции

Рис. 12.9 При добавлении второй миграции мы добавляем новый файл миграции и файл Designer.cs, а также обновляем файл AppDbContextModelSnapshot, чтобы он соответствовал файлу новой миграции, Designer.cs

Как и раньше, при этом создаются файлы миграции, но база данных не изменяется. Вы можете применить миграцию и обновить базу данных, выполнив команду

```
dotnet ef database update
```

Она сравнивает миграции в приложении с таблицей __EFMigrationsHistory в базе данных, чтобы увидеть, какие миграции еще остались, а затем выполняет их. EF Core выполнит миграцию 20220825201452_ExtraRecipeFields, добавив поля IsVegetarian и IsVegan в базу данных, как показано на рис. 12.10. Использование миграций – отличный способ обеспечить версионность базы данных наряду с кодом приложения в системе управления версиями. Вы можете легко проверить исходный код своего приложения и воссоздать схему базы данных, которую приложение использовало в этот момент.



Применяя вторую миграцию к базе данных, вы добавляете новые поля в таблицу Recipes

Рис. 12.10 Применяя миграцию ExtraRecipeFields к базе данных, вы добавляете поля IsVegetarian и IsVegan в таблицу рецептов

Миграции легко использовать, когда вы работаете в одиночку или когда развертываете приложение на одном веб-сервере, но даже в этих случаях есть важные моменты, которые следует учитывать при принятии решения о том, как управлять своими базами данных. В случае с приложениями с несколькими веб-серверами, использующими общую базу данных или для контейнерных приложений, следует быть особенно внимательными.

Эта книга посвящена ASP.NET Core, а не EF Core, поэтому я не хочу подробно останавливаться на управлении базами данных, но в разделе 12.5 указаны некоторые моменты, которые следует учитывать при использовании миграций в промышленном окружении.

В следующем разделе мы вернемся к основной теме – определению бизнес-логики и выполнению CRUD-операций с базой данных.

12.4 Выполнение запроса к базе данных и сохранение в ней данных

Посмотрим, на каком этапе создания приложения с рецептами мы находимся:

- мы создали простую модель данных для приложения, состоящую из рецептов и ингредиентов;
- мы создали миграции для модели данных, чтобы обновить внутреннюю модель сущностей;
- мы применили миграции к базе данных, так чтобы ее схема соответствовала модели EF Core.

В этом разделе мы создадим бизнес-логику для нашего приложения, создав сервис `RecipeService`. Он будет обрабатывать запросы к базе данных для получения рецептов, создания новых рецептов и изменения существующих. Поскольку у этого приложения простая предметная область, я буду использовать `RecipeService` для обработки всех требований, но в ваших приложениях может быть несколько сервисов, которые взаимодействуют друг с другом для обеспечения бизнес-логики.

ПРИМЕЧАНИЕ Когда речь идет о простых приложениях, у вас может возникнуть соблазн перенести эту логику в обработчики конечных точек или Razor Pages. Данный подход может подойти для небольших приложений, но я призываю вас сопротивляться этому побуждению; извлекая бизнес-логику в другие сервисы, вы отделяете HTTP-ориентированную природу Razor Pages и веб-API от базовой бизнес-логики. Благодаря этой связи бизнес-логику легче тестировать, и ее можно использовать повторно.

В нашей базе пока нет данных, поэтому лучше начать с создания рецепта.

12.4.1 Создание записи

В этом разделе мы дадим пользователям возможность создавать рецепты, используя API. Клиенты отправляют все сведения о рецепте в теле запроса POST в конечную точку приложения. Конечная точка использует атрибуты привязки и валидацию модели для подтверждения валидности запроса, как вы узнали в главе 7.

Если запрос валиден, обработчик конечной точки вызывает сервис RecipeService для создания нового объекта Recipe в базе данных. Поскольку тема этой главы – EF Core, я сосредоточусь только на этом сервисе, но вы всегда можете обратиться к исходному коду для данной книги, если хотите увидеть, как это все сочетается друг с другом в приложении с минимальным API.

Бизнес-логика создания рецепта в этом приложении проста – ее нет! Скопируйте свойства из модели привязки команд, предоставленной в обработчике конечной точки, в сущность `Recipe` и ее `Ingredients`, добавьте объект `Recipe` в `AppDbContext` и сохраните его в базе данных, как показано на рис. 12.11.

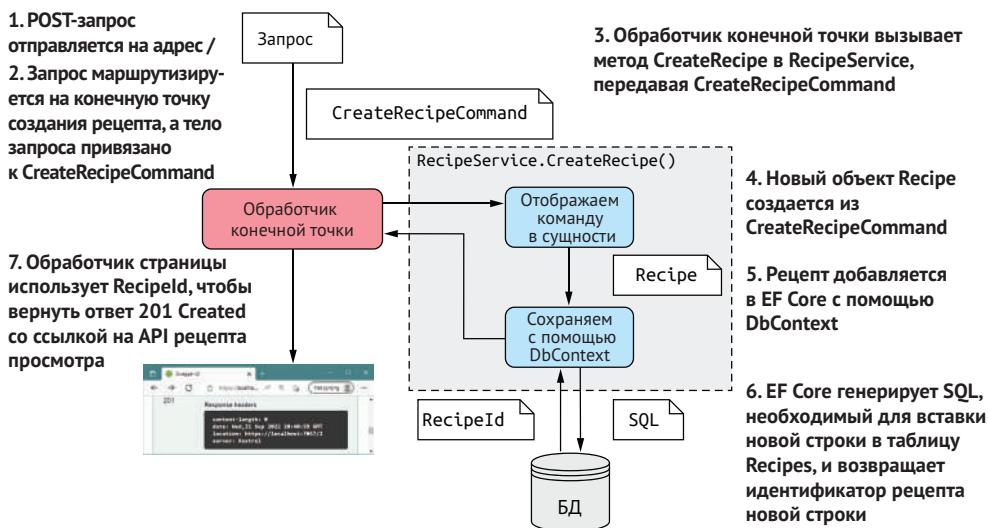


Рис. 12.11 Вызов конечной точки POST и создание новой сущности. Объект Recipe создается на основе модели CreateRecipeCommand и добавляется в DbContext. EF Core генерирует SQL-запрос для добавления новой строки в таблицу рецептов в базе данных.

ВНИМАНИЕ! Множество простых эквивалентных примеров приложений с использованием EF или EF Core позволяют напрямую выполнять привязку к сущности `Recipe` в качестве модели в конечной точке. К сожалению, из-за этого приложение становится уязвимым для *чрезмерной передачи данных* (*overposting*), а это плохая практика. Если вы хотите избежать использования шаблонного кода отображения в своих приложениях, рассмотрите возможность применения такой библиотеки, как AutoMapper (<http://>

automapper.org/). Дополнительную информацию о чрезмерной передаче данных см. в моей статье: <http://mng.bz/d480>.

Создание сущности в EF Core включает в себя добавление новой строки в отображаемую таблицу. Всякий раз, когда вы создаете новый рецепт, вы также добавляете связанные сущности `Ingredient`. EF Core позаботится о том, чтобы правильно связать все это, создав правильный идентификатор `RecipeId` для каждого ингредиента в базе данных.

Все взаимодействия с EF Core и базой данных начинаются с экземпляра `AppDbContext`, который обычно вводится с использованием внедрения зависимостей через конструктор. Для создания новой сущности требуется выполнить три шага:

- 1 создать сущности `Recipe` и `Ingredient`;
- 2 добавить их в список отслеживаемых сущностей EF Core, применив `_context.Add(entity)`;
- 3 использовать оператор `INSERT`, чтобы добавить необходимые строки в таблицы `Recipe` и `Ingredient`, вызвав `_context.SaveChangesAsync()`.

ПОДСКАЗКА Существуют *синхронные* и *асинхронные* версии большинства команд EF Core, которые включают в себя взаимодействие с базой данных, например `SaveChanges()` и `SaveChangesAsync()`. В целом асинхронные версии позволяют приложению обрабатывать больше одновременных подключений, поэтому при возможности я всегда выбираю их.

В листинге 12.6 показаны эти три шага на практике. Основная часть кода в этом примере включает копирование свойств из `CreateRecipeCommand` в сущность `Recipe`. Взаимодействие с `AppDbContext` состоит всего из двух методов: `Add()` и `SaveChangesAsync()`.

Листинг 12.6. Создание сущности Recipe в базе данных в RecipeService

Создаем
рецепт
путем ото-
бражения
из объекта
команды
в сущность
`Recipe`

```
readonly AppDbContext _context; ← Экземпляр AppDbContext передает-  
public async Task<int> CreateRecipe(CreateRecipeCommand cmd) ← ся в конструктор класса с помощью  
{ ← внедрения зависимостей  
    var recipe = new Recipe ← 2. CreateRecipeCommand передает-  
    { ← ся из обработчика конечной точки  
        Name = cmd.Name,  
        TimeToCook = new TimeSpan(  
            cmd.TimeToCookHrs, cmd.TimeToCookMins, 0),  
        Method = cmd.Method,  
        IsVegetarian = cmd.IsVegetarian,  
        IsVegan = cmd.IsVegan,  
        Ingredients = cmd.Ingredients.Select(i =>
```

```

    new Ingredient
    {
        Name = i.Name,
        Quantity = i.Quantity,
        Unit = i.Unit,
    }).ToList()
};

_context.Add(recipe);
await _context.SaveChangesAsync();
return recipe.RecipeId;
}

```

Сообщаем EF Core, что нужно отслеживать новые сущности

Отображаем каждую команду CreateIngredientCommand в сущность Ingredient

Даем EF Core указание вести запись сущностей в базу данных; используется асинхронная версия команды

EF Core заполняет поле RecipeId в новом рецепте при его сохранении

Если возникает проблема, когда EF Core пытается взаимодействовать с базой данных, – например, вы не запустили миграции, чтобы обновить схему базы данных, – то будет выброшено исключение. Здесь я этого не показывал, но важно учитывать это в своем приложении, чтобы не демонстрировать пользователям неприятную страницу с ошибкой, если что-то пошло не так.

Если все идет хорошо, то EF Core обновляет все автоматически сгенерированные идентификаторы сущностей (RecipeId для Recipe, а также RecipeId и IngredientId для Ingredient). Возвращайте идентификатор рецепта в обработчик конечной точки, чтобы он мог его использовать, например чтобы вернуть идентификатор в ответе API.

Вот и все – вы создали свою первую сущность с помощью EF Core.

В следующем разделе мы рассмотрим загрузку этих сущностей из базы данных, чтобы их можно было увидеть в списке.

ПОДСКАЗКА Тип `DbContext` является реализацией паттернов «Единица работы» и «Репозиторий», поэтому, как правило, не нужно вручную реализовывать эти паттерны в своих приложениях. Подробнее о них можно прочитать на странице <https://martinfowler.com/eaaCatalog>.

Вот и все – мы создали свою первую сущность с помощью EF Core. В следующем разделе мы рассмотрим загрузку этих сущностей из базы данных, чтобы их можно было увидеть в списке.

12.4.2 Загрузка списка записей

Теперь, когда вы можете создавать рецепты, нужно написать код для их просмотра. К счастью, загружать данные в EF Core просто, широко используя методы LINQ, чтобы контролировать, какие поля нам нужны. Для своего приложения мы создадим метод в `RecipeService`, который возвращает сокращенное представление рецепта, состоящий из `RecipeId`, `Name` и `TimeToCook`. Все это представлено в виде модели `RecipeSummaryViewModel`, как показано на рис. 12.12.

ПРИМЕЧАНИЕ С технической точки зрения создание модели представления является задачей пользовательского интерфейса, а не бизнес-логики. Здесь я возвращаю их напрямую из `RecipeService`

главным образом для того, чтобы дать понять вам, что не следует использовать сущности EF Core напрямую в общедоступном API конечной точки. В качестве альтернативы можно вернуть сущность Recipe непосредственно из RecipeService, а затем создать и вернуть RecipeSummaryViewModel в коде обработчика конечной точки.

Метод GetRecipes из RecipeService концептуально прост и следует общепринятым шаблонам выполнения запроса к базе данных, как показано на рис. 12.13. EF Core использует цепочку команд LINQ для определения запроса, исполняемого в базе данных. Свойство DbSet<Recipe> в AppDataContext является IQueryable, поэтому вы можете использовать все обычные предложения Select и Where, которые вы бы применяли с другими провайдерами IQueryable. EF Core преобразует их в инструкцию SQL для осуществления запроса к базе данных, когда вы вызываете функцию выполнения, например ToListAsync(), ToArrayAsync(), SingleAsync() или их неасинхронных собратьев.

Вы также можете использовать метод расширения Select() для отображения в объекты, отличные от ваших сущностей, как часть SQL-запроса. Вы можете использовать это для выполнения эффективного запроса к базе данных, извлекая только нужные вам столбцы.

В листинге 12.7 показан код для получения списка моделей RecipeSummaryViewModel, следуя тому же базовому шаблону, что и на рис. 12.12.

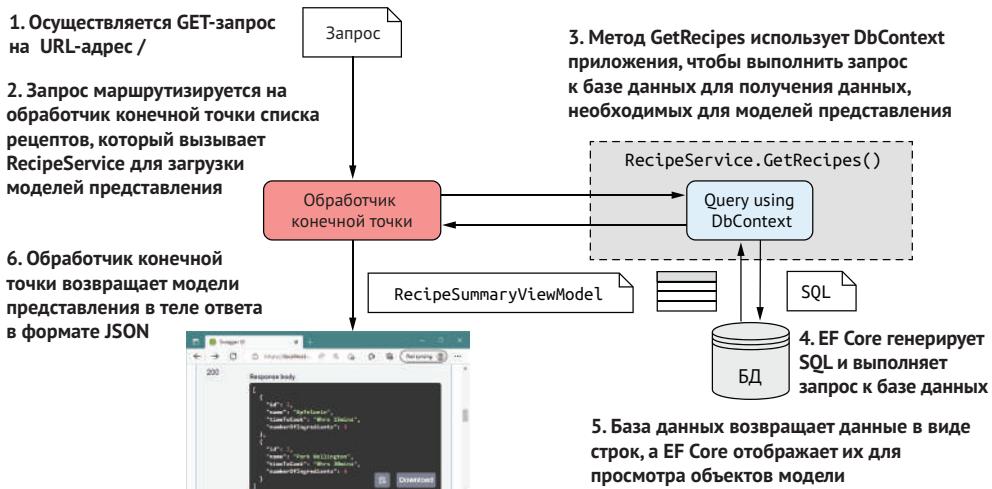
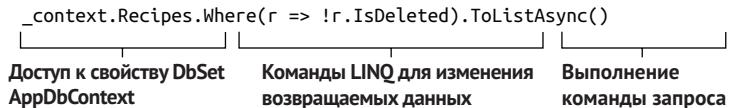


Рис. 12.12 Вызов конечной точки списка GET и запрос к базе данных для получения списка RecipeSummaryViewModels. EF Core генерирует SQL-код для получения необходимых полей из базы данных и отображает их для просмотра объектов модели

Здесь используется LINQ-выражение Where для фильтрации рецептов, помеченных как удаленные, и предложение Select для отображения в модели представления. Команда ToListAsync() дает указание EF Core сгенерировать SQL-запрос, выполнить его в базе данных и создать RecipeSummaryViewModel из возвращаемых данных.

Рис. 12.13 Три части запроса к базе данных с использованием EF Core



Листинг 12.7 Загрузка списка элементов с помощью EF Core в RecipeService

```
public async Task<ICollection<RecipeSummaryViewModel>> GetRecipes()
{
    return await _context.Recipes <-- | Запрос начинается
        .Where(r => !r.IsDeleted)
        .Select(r => new RecipeSummaryViewModel
    {
        Id = r.RecipeId,
        Name = r.Name,
        TimeToCook = $"{r.TimeToCook.TotalMinutes}mins"
    })
    .ToListAsync(); <-- | Выполняем SQL-запрос
} <-- | и создаем окончательные
                                         | модели представления
```

EF Core будет запрашивать только те столбцы Recipe, которые необходимы для правильного отображения модели представления.

Обратите внимание, что в методе Select свойство TimeToCook из TimeSpan преобразуется в строку с использованием строковой интерполяции:

```
TimeToCook = $"{r.TimeToCook.TotalMinutes}mins"
```

Я уже говорил, что EF Core преобразует серию выражений LINQ в SQL, но это не вся правда; EF Core не может или не знает, как преобразовать некоторые выражения. В таких случаях, как в этом примере, EF Core находит поля из базы данных, которые ей нужны для выполнения выражения на стороне клиента, выбирает их из базы данных, а затем выполняет выражение на языке C#. Это позволяет сочетать мощность и производительность вычисления на стороне базы данных без ущерба для функциональности C#.

ВНИМАНИЕ! Вычисление на стороне клиента – мощная и полезная вещь, но оно может вызвать проблемы. Как правило, последние версии EF Core будут выдавать исключение, если запрос требует опасного вычисления на стороне клиента, гарантируя (например), что вы не сможете случайно вернуть все записи клиенту перед фильтрацией. Чтобы увидеть дополнительные примеры, а также узнать, как избежать этих проблем, см. документацию на странице <http://mng.bz/zxP6>.

На данном этапе у вас есть список записей, отображающий сводку данных рецепта, поэтому следующий очевидный шаг – загрузка деталей для отдельной записи.

12.4.3 Загрузка отдельной записи

В большинстве случаев загрузка отдельной записи аналогична загрузке списка записей. Они имеют ту же общую структуру, которую вы ви-

дели на рис. 12.13, но при загрузке отдельной записи обычно используется предложение `Where` и выполняется команда, ограничивающая данные одной сущностью.

В листинге 12.8 показан код для извлечения рецепта по идентификатору, следуя тому же базовому шаблону, что и раньше (рис. 12.12). Здесь используется LINQ-выражение `Where()` для ограничения запроса одним рецептом, где `RecipeId == id`, и есть предложение `Select` для отображения в `RecipeDetailViewModel`. Предложение `SingleOrDefaultAsync()` заставит EF Core сгенерировать SQL-запрос, выполнить его в базе данных и создать модель представления.

ПРИМЕЧАНИЕ Предложение `SingleOrDefaultAsync()` возбудит исключение, если предыдущее предложение `Where` вернет несколько записей.

Листинг 12.8 Загрузка отдельного элемента с помощью EF Core в `RecipeService`

```
public async Task<RecipeDetailViewModel> GetRecipeDetail(int id)
{
    return await _context.Recipes
        .Where(x => x.RecipeId == id)
        .Select(x => new RecipeDetailViewModel
    {
        Id = x.RecipeId,
        Name = x.Name,
        Method = x.Method,
        Ingredients = x.Ingredients
            .Select(item => new RecipeDetailViewModel.Item
            {
                Name = item.Name,
                Quantity = $"{item.Quantity} {item.Unit}"
            })
    })
    .SingleOrDefaultAsync();
}
```

The diagram illustrates the execution flow of the `GetRecipeDetail` method:

- Идентификатор загружаемого рецепта передается в качестве параметра**: An annotation pointing to the `id` parameter.
- Ограничивает запрос рецептами с указанным идентификатором**: An annotation pointing to the `Where` clause.
- Выполняет запрос и сопоставляет данные с моделью представления**: An annotation pointing to the `Select` clause.
- Как и раньше, запрос начинается со свойства `DbSet`**: An annotation pointing to the `Recipes` property.
- Сопоставляет рецепт с `RecipeDetailViewModel`**: An annotation pointing to the inner `Select` statement.
- Загружает и сопоставляет связанные ингредиенты как часть одного запроса**: An annotation pointing to the nested `Select` statement for ingredients.

Обратите внимание, что помимо отображения `Recipe` в `RecipeDetailViewModel` вы также отображаете соответствующие ингредиенты для рецепта, как если бы вы работали с объектами прямо в памяти. Это одно из преимуществ использования объектно-реляционного отображения – вы можете легко отображать дочерние объекты, позволяя EF Core решить, как лучше создать SQL-запросы для получения данных.

ПРИМЕЧАНИЕ EF Core регистрирует все выполняемые инструкции SQL как события `LogLevel.Information` по умолчанию, поэтому можно легко увидеть, какие запросы выполняются к базе данных.

Наше приложение определено приобретает форму: мы можем создавать новые рецепты, просматривать их все в виде списка и прокручивать, чтобы увидеть отдельные рецепты с ингредиентами. Однако вскоре кто-то сделает опечатку и захочет изменить свои данные. Для этого нужно будет заняться *обновлением*.

12.4.4 Обновление модели с изменениями

Обновление сущностей, после того как они изменились, – обычно самая сложная часть CRUD-операций, поскольку переменных очень много. На рис. 12.14 представлен обзор этого процесса применительно к нашему приложению рецептов.

Я не буду обсуждать в этой книге аспект связей, потому что обычно это комплексная проблема, и то, как вы ее решите, зависит от специфики вашей модели данных. Вместо этого я сосредоточусь на обновлении свойств самой сущности `Recipe`.

ПРИМЕЧАНИЕ Подробнее об обновлении связей в EF Core см. в книге «Entity Framework Core в действии», 2-е изд., Джона П. Смита (Manning, 2021): <http://mng.bz/w9D2>.



Рис. 12.14 Обновление сущности состоит из трех этапов: чтение сущности с помощью EF Core, обновление ее свойств и вызов метода `SaveChangesAsync()` в `DbContext`, чтобы сгенерировать SQL-код для обновления правильных строк в базе данных

В случае с веб-приложениями при обновлении сущности обычно выполняются действия, приведенные на рис. 12.14:

- 1 чтение сущности из базы данных;
- 2 изменение свойств сущности;
- 3 сохранение изменений в базе данных.

Эти три шага инкапсулируются в методе `RecipeService` с именем `UpdateRecipe`. Этот метод принимает параметр `UpdateRecipeCommand` и содержит код для изменения сущности `Recipe`.

ПРИМЕЧАНИЕ Как и в случае с командой `Create`, нельзя напрямую изменять сущности в обработчике конечной точки минимального API, гарантируя, что пользовательский интерфейс или API будет отделен от бизнес-логики.

В следующем листинге показан метод `RecipeService.UpdateRecipe`, который обновляет сущность `Recipe`. Он выполняет три шага, которые мы определили ранее: чтение, изменение и сохранение сущности. Я извлек код, чтобы обновить рецепт новыми значениями для вспомогательного метода.

Листинг 12.9 Обновление существующей сущности с помощью EF Core в `RecipeService`

```
Поиск нужного рецепта осуществляется по идентификатору непосредственно в множестве Recipes
public async Task UpdateRecipe(UpdateRecipeCommand cmd)
{
    var recipe = await _context.Recipes.FindAsync(cmd.Id); ←
    if(recipe is null) {
        throw new Exception("Unable to find the recipe");
    }
    → UpdateRecipe(recipe, cmd); ← Выполняет SQL для сохранения изменений в базе данных
    await _context.SaveChangesAsync();
}

Устанавливает новые значения для сущности Recipe
static void UpdateRecipe(Recipe recipe, UpdateRecipeCommand cmd)
{
    recipe.Name = cmd.Name;
    recipe.TimeToCook =
        new TimeSpan(cmd.TimeToCookHrs, cmd.TimeToCookMins, 0);
    recipe.Method = cmd.Method;
    recipe.IsVegetarian = cmd.IsVegetarian;
    recipe.IsVegan = cmd.IsVegan;
}
```

Если указан неверный идентификатор, Recipe будет нулевым

Вспомогательный метод для установки новых свойств сущности Recipe

В этом примере я прочитал сущность `Recipe`, используя метод `FindAsync(id)`, предоставленный `DbSet`. Это простой вспомогательный метод для загрузки объекта по идентификатору, в данном случае `RecipeId`. Я мог бы написать аналогичный запрос, используя LINQ:

```
_context.Recipes.Where(r=>r.RecipeId == cmd.Id).FirstOrDefault();
```

Однако применение методов `FindAsync()` или `Find()` немного более декларативно и короче.

ПРИМЕЧАНИЕ Метод `Find()` немного сложнее. Сначала он проверяет, не отслеживается ли сущность в `DbContext`. Если это так (по-

тому что сущность была уже загружена ранее в этом запросе), то сущность возвращается немедленно без обращения к базе данных. Очевидно, что это быстрее, если сущность отслеживается, а может и нет, если вы знаете, что это не так.

Возможно, вам интересно, откуда EF Core знает, какие столбцы нужно обновить при вызове метода `SaveChangesAsync()`. Самый простой подход – обновлять каждый столбец; если поле не изменилось, то не имеет значения, напишете ли вы снова то же значение. Но EF Core намного умнее.

Она внутренне отслеживает *состояние* всех сущностей, загружаемых из базы данных, и создает копию всех значений свойств сущности, чтобы можно было отслеживать, какие из них изменились. Когда вы вызываете метод `SaveChanges()`, EF Core сравнивает состояние всех отслеживаемых сущностей (в данном случае сущности `Recipe`) с копией отслеживания. Все свойства, которые были изменены, включаются в инструкцию `UPDATE`, отправляемую в базу данных, а неизмененные свойства игнорируются.

ПРИМЕЧАНИЕ EF Core предоставляет другие механизмы для отслеживания изменений, а также параметры, чтобы полностью отключить отслеживание изменений. Обратитесь к документации или третьей главе книги «Entity Framework Core в действии» Джона П. Смита, 2-е изд. (Manning, 2021) для получения подробной информации: <http://mng.bz/q9PJ>. Вы можете просмотреть, какие сведения отслеживает `DbContext`, выполнив доступ к `DbContext.ChangeTracer.DebugView`, как описано в документации: <http://mng.bz/8rlz>.

Теперь, когда вы можете обновлять рецепты, приложение с рецептами почти готово. «Но подождите, – кричите вы, – мы еще не работали с функцией `Delete!`» И это правда, однако на самом деле я обнаружил всего несколько случаев, когда вам понадобится удалить данные. Рассмотрим требования для удаления рецепта из приложения:

- необходимо предоставить API, который удаляет рецепт;
- после удаления рецепта он не должен появляться в списке рецептов и его нельзя восстановить.

Этого можно добиться, удалив рецепт из базы данных, но проблема с данными состоит в том, что, когда они исчезают, их уже нет! Что делать, если пользователь случайно удалил запись? Кроме того, удаление строки из реляционной базы данных обычно влияет на другие сущности. Например, нельзя удалить строку из таблицы `Recipe` в приложении, не удалив также все строки `Ingredient`, которые ссылаются на нее, благодаря ограничению внешнего ключа для `Ingredient.RecipeId`.

EF Core может легко справиться с такими случаями истинного удаления с помощью команды `DbContext.Remove(entity)`, но обычно при необходимости удалить данные имеется в виду их архивирование или скрытие от пользовательского интерфейса. Распространенный подход к работе с этим сценарием заключается в том, чтобы использовать

что-то вроде параметра «Is this entity deleted», например `IsDeleted`, который я включил в сущность `Recipe`:

```
public bool IsDeleted { get; set; }
```

Если вы воспользуетесь данным подходом, удалять данные внезапно станет проще, поскольку это не что иное, как обновление сущности. Больше никаких проблем с потерянными данными и ссылочной целостностью.

ПРИМЕЧАНИЕ Основное исключение, которое я обнаружил в этом шаблоне, – когда вы сохраняете информацию, позволяющую установить личность ваших пользователей. В этих случаях вы, вероятно, будете обязаны (и, возможно, юридически) удалять эти сведения из своей базы данных по запросу.

При таком подходе можно создать метод удаления в `RecipeService`, который обновляет параметр `IsDeleted`, как показано в следующем листинге. Кроме того, вы должны убедиться, что у вас есть предложения `Where()` во всех других методах `RecipeService`, чтобы гарантировать, что вы не можете отобразить удаленный рецепт, как вы видели в листинге 12.9 для метода `GetRecipes()`.

Листинг 12.10 Помечаем сущности как удаленные в EF Core

```
public async Task DeleteRecipe(int recipeId) {
    var recipe = await _context.Recipes.FindAsync(recipeId); <-- Извлекает сущность Recipe
    if(recipe is null) { <-- по идентификатору
        throw new Exception("Unable to find the recipe");
    }
    recipe.IsDeleted = true; <-- Выполняет SQL
    await _context.SaveChangesAsync(); <-- для сохранения
} <-- изменений
    <-- в базе данных
```

Если указан неверный идентификатор, то рецепт будет нулевым

Помечает рецепт как удаленный

Такой подход удовлетворяет требованиям – вы удаляете рецепт из пользовательского интерфейса приложения, – но упрощает ряд вещей. Это мягкое удаление подойдет не для всех ситуаций, но я обнаружил, что это распространенный шаблон в проектах, над которыми я работал.

СОВЕТ В EF Core есть удобная функция – *глобальные фильтры запросов*. Они позволяют указать предложение `Where` на уровне модели, чтобы вы могли, например, гарантировать, что EF Core никогда не загрузит рецепты, у которых `IsDeleted` имеет значение `true`. Это также полезно для разделения данных в многоклиентском окружении. Подробнее об этом – на странице <http://mng.bz/EQxd>.

Мы почти подошли к концу этой главы, посвященной EF Core. Мы рассмотрели основы добавления EF Core в проект и выяснили, как использовать эту библиотеку для упрощения доступа к данным, но вам, скорее всего, понадобится узнать больше о ней, по мере того

как ваши приложения будут становиться более сложными. В заключительном разделе данной главы я хотел бы выделить ряд вещей, которые нужно принять во внимание, перед тем как использовать EF Core в своих приложениях, чтобы вам были знакомы некоторые проблемы, с которыми вы столкнетесь по мере роста своих приложений.

12.5 Использование EF Core в промышленных приложениях

Эта книга посвящена ASP.NET Core, а не EF Core, поэтому я не хотел тратить слишком много времени на ее изучение. Данная глава должна была дать вам достаточно информации, чтобы приступить к работе, но вам непременно нужно узнать еще кое-что, прежде чем вы даже подумаете о том, чтобы использовать EF Core в промышленном окружении. Как я уже неоднократно говорил, я рекомендую книгу «Entity Framework Core в действии» для получения подробной информации об этой библиотеке, или вы можете изучить сайт с документацией по EF Core: <https://docs.microsoft.com/ef/core/>.

Приведенные ниже темы не важны для начала работы с EF Core, но вы быстро столкнетесь с ними, когда создадите готовое к промышленной эксплуатации приложение. Данный раздел не является предписывающим руководством по решению каждой из этих проблем; это скорее набор вещей, которые нужно принять во внимание, перед тем как вы перейдете к промышленной эксплуатации.

- *Скаффолдинг столбцов* – EF Core использует консервативные значения для таких вещей, как строковые столбцы, допуская строки большой или неограниченной длины. На практике у вас может возникнуть желание ограничить эти и другие типы данных разумными значениями;
- *валидация* – вы можете декорировать сущности атрибутами валидации `DataAnnotations`, но EF Core не будет автоматически проверять значения, перед тем как сохранить их в базе данных. Это отличается от поведения EF 6.x, в котором валидация была автоматической;
- *параллелизм* – EF Core предоставляет несколько способов для работы с параллелизмом, когда несколько пользователей пытаются обновить сущность одновременно. Одно из частичных решений – использовать для своих сущностей столбцы `Timestamp`;
- *обработка ошибок* – базы данных и сети по своей природе нестабильны, поэтому вам всегда придется учитывать временные ошибки. EF Core включает в себя различные функции для поддержания устойчивости подключения путем повторных попыток при сбоях сети;
- *синхронные и асинхронные команды* – EF Core предоставляет синхронные и асинхронные команды для взаимодействия с базой данных. Часто асинхронный режим лучше подходит для веб-приложений, но у этого аргумента есть нюансы, которые не позволяют рекомендовать использование одного подхода вместо другого во всех ситуациях.

EF Core – отличный инструмент для продуктивной работы при написании кода доступа к данным, но есть некоторые аспекты работы с базой данных, которые неизбежно неудобны. Проблема управления базами данных – одна из самых сложных задач, которую приходится решать. Большинство веб-приложений используют какую-либо базу данных, поэтому приведенные ниже ситуации могут повлиять на разработчиков ASP .NET Core в какой-то момент:

- *автоматические миграции* – если вы автоматически развертываете свое приложение в промышленном окружении как часть некоего конвейера DevOps, вам неизбежно понадобится способ автоматического применения миграций к базе данных. Эту проблему можно решить несколькими способами, например создать скрипт для инструмента командной строки .NET EF Core, применить миграции в коде запуска вашего приложения или использовать специальный инструмент. У каждого подхода есть свои плюсы и минусы;
- *несколько веб-хостов* – особое внимание следует уделять тому, есть ли у вас веб-серверы, на которых размещено ваше приложение и которые указывают на одну и ту же базу данных. Если это так, то применять миграции в коде запуска вашего приложения становится сложнее, так как вы должны убедиться, что только одно приложение может обновлять схему базы данных за раз;
- *делать изменения схемы обратно совместимыми* – следствием подхода с использованием нескольких веб-хостингов является то, что вы часто будете оказываться в ситуации, когда ваше приложение обращается к базе данных, у которой более *новая* схема, чем думает приложение. Это означает, что обычно вы должны будете стремиться к тому, чтобы делать изменения схемы обратно совместимыми везде, где это возможно;
- *сохранение миграций в другой сборке* – в этой главе я включил всю свою логику в один проект, но в больших приложениях доступ к данным часто осуществляется в другом проекте. Для приложений с такой структурой необходимо использовать несколько иные команды при работе с интерфейсом командной строки .NET или командлетами PowerShell;
- *наполнение базы данных* – когда вы впервые создаете базу данных, то часто хотите, чтобы в ней были некие исходные данные, такие как пользователь по умолчанию. У EF 6.x имелся встроенный механизм для наполнения базы данных, тогда как EF Core требует, чтобы вы сами явно заполняли базу данных.

То, как вы решите каждую из этих проблем, будет зависеть от инфраструктуры и подхода к развертыванию, который вы применяете в своем приложении. Ни один из них не доставляет особого удовольствия, но это – досадная необходимость. Мужайтесь – все их так или иначе можно решить!

На этом мы подошли к концу данной главы, посвященной EF Core, и к концу второй части. В следующей части мы оставим минимальные API и рассмотрим создание страничных приложений с отрисовкой на сервере с помощью Razor Pages.

Резюме

- EF Core – инструмент объектно-реляционного отображения, позволяющий взаимодействовать с базой данных путем манипулирования стандартными классами РОСО, называемыми сущностями, в приложении. Это может уменьшить объем SQL-кода и необходимых знаний о базах данных, которые нужны для продуктивной работы;
- EF Core отображает классы сущностей в таблицы, свойства сущности – в столбцы в таблицах, а экземпляры объектов сущностей – в строки в этих таблицах. Даже если вы используете EF Core, чтобы не работать напрямую с базой данных, необходимо учитывать это;
- EF Core использует модель провайдера базы данных, позволяющую изменять основную базу данных без изменения кода манипулирования объектами. У EF Core имеются провайдеры баз данных для Microsoft SQL Server, SQLite, PostgreSQL, MySQL и многих других;
- EF Core – кросс-платформенная библиотека. Она обладает хорошей производительностью для объектно-реляционного отображения, но у нее иной набор функций, которые отличаются от тех, что были у EF 6.x. Тем не менее EF Core рекомендуется использовать для всех новых приложений вместо с EF 6.x;
- EF Core хранит внутреннее представление сущностей в приложении и способ их отображения в базу данных на основе свойств `DbSet<T>` класса `DbContext`. EF Core создает модель на основе самих классов сущностей и любых других сущностей, на которые они ссылаются;
- EF Core добавляется в приложение при добавлении пакета провайдера базы данных NuGet. Также следует установить пакеты проектирования для EF Core. Это работает вкупе с инструментами командной строки .NET для создания и применения миграции к базе данных;
- EF Core включает множество соглашений о том, как определяются сущности, например основные и внешние ключи. Можно настроить определение сущностей декларативно, используя `DataAnnotations` или с помощью механизма Fluent API;
- приложение использует `DbContext` для взаимодействия с EF Core и базой данных. Вы регистрируете его в контейнере внедрения зависимостей, используя `AddDbContext<T>`, определяя провайдера базы данных и предоставляя строку подключения. Благодаря этому `DbContext` становится доступен в контейнере во всем приложении;
- EF Core использует миграции для отслеживания изменений в определениях сущностей. Они используются для того, чтобы гарантировать, что определения сущностей, внутренняя модель EF Core и схема базы данных совпадают;
- после изменения сущности можно создать миграцию с помощью инструмента командной строки .NET или командлетов PowerShell. Чтобы создать новую миграцию, используя интерфейс команд-

ной строки .NET, выполните команду `dotnet ef migrations add NAME` в папке проекта, где NAME – это имя, которое вы хотите присвоить миграции. Будет проведено сравнение вашей текущей копии `DbContext` с предыдущей версией и сгенерированы необходимые инструкции SQL для обновления базы данных;

- можно применить миграцию к базе данных с помощью команды `dotnet ef database update`. Будет создана база данных, если она еще не существует, и будут применены все оставшиеся миграции;
- EF Core не взаимодействует с базой данных при создании миграций, а только когда вы явно обновляете базу данных, поэтому вы все равно можете создавать их, когда не находитесь в сети;
- вы можете добавлять сущности в базу данных EF Core, создавая новую сущность, вызвав `_context.Add(e)` в экземпляре контекста данных приложения, `_context`, и `_context.SaveChangesAsync()`. Так вы сгенерируете необходимые инструкции `INSERT` для добавления новых строк в базу данных;
- вы можете загружать записи из базы данных, используя свойства `DbSet<T>` класса `DbContext`. Они предоставляют интерфейс `IQueryable`, поэтому вы можете использовать инструкции LINQ для фильтрации и преобразования данных в базе данных, прежде чем они будут возвращены;
- обновление сущности состоит из трех шагов: чтение сущности из базы данных, изменение объекта и сохранение изменений в базе данных. EF Core будет отслеживать, какие свойства были изменены, чтобы оптимизировать SQL-код, который она генерирует;
- вы можете удалять сущности в EF Core с помощью метода `Remove`, но следует тщательно подумать, нужна ли вам такая функциональность. Часто техника мягкого удаления с использованием параметра `IsDeleted` безопаснее и проще в реализации;
- в этой главе рассматривается только часть вопросов, которые необходимо принять во внимание при применении EF Core в своем приложении. Прежде чем использовать ее в промышленном окружении, нужно учитывать, среди прочего, типы данных, сгенерированные для полей, валидацию, параллелизм, наполнение исходных данных, миграции в работающем приложении и в ситуации с веб-фермой.

Часть III

Генерация HTML-кода с помощью Razor Pages и MVC

В частях I и II мы подробно рассмотрели, как создавать приложения с JSON API с использованием минимальных API. Вы узнали, как настроить приложение из нескольких источников, как использовать внедрение зависимостей для уменьшения связанности в приложении и как документировать API с помощью OpenAPI.

Приложения API сегодня повсюду. Их используют мобильные приложения, односторонние веб-приложения, создаваемые с помощью фреймворков Angular, React и Blazor; даже другие приложения используют их для связи между серверами. Но во многих случаях нам не нужны раздельные серверное и клиентское приложения. Вместо этого можно создать приложение с отрисовкой на стороне сервера.

При отрисовке на стороне сервера приложение генерирует HTML-код на сервере, а браузер отображает этот код непосредственно; никакого дополнительного фреймворка для разработки клиентских приложений не требуется. Вы по-прежнему можете добавить динамическое поведение на стороне клиента с помощью JavaScript, но, по сути, каждая страница вашего приложения представляет собой отдельный запрос и ответ, что упрощает работу разработчика.

В части III вы познакомитесь с фреймворками Razor Pages и Model-View-Controller (MVC), которые ASP.NET Core использует для создания приложений с отрисовкой на стороне сервера. В главах с 13 по 16 мы рассмотрим поведение самого фреймворка Razor Pages, а также маршрутизацию и привязку модели. В главах 17 и 18 вы увидите, как создать пользовательский интерфейс для своего приложения, используя синтаксис Razor и тег-хелперы, чтобы пользователи могли перемещаться по приложению и взаимодействовать с ним.

В главе 19 вы узнаете, как использовать фреймворк MVC напрямую вместо Razor Pages. Вы узнаете, как использовать контроллеры MVC для создания приложений с отрисовкой на стороне сервера и когда следует выбирать контроллеры MVC вместо Razor Pages. В главе 20 вы узнаете, как использовать контроллеры MVC для создания приложений API в качестве альтернативы минимальным API. Наконец, в главах 21 и 22 вы узнаете, как провести рефакторинг приложений для извлечения общего кода из страниц Razor Pages и контроллеров API с помощью фильтров.

13

Создание сайта с помощью Razor Pages

В этой главе:

- начало работы с Razor Pages;
- знакомство с Razor Pages и паттерном проектирования «модель–представление–контроллер» (MVC);
- использование Razor Pages в ASP.NET Core.

На данный момент мы создали один тип приложения ASP.NET Core: приложения с минимальным API, которые возвращают JSON. В этой главе вы узнаете, как создавать многостраничные приложения с отрисовкой на стороне сервера с помощью Razor Pages. Большинство приложений ASP.NET Core попадают в одну из трех категорий:

- *API, предназначенный для использования на другой машине или в коде –* веб-приложения часто служат API для серверных процессов, мобильного приложения или фреймворка для создания одностраничных приложений. В этом случае приложение предоставляет данные в машиночитаемых форматах, таких как JSON или XML, вместо ориентированного на человека вывода в формате HTML;
- *веб-приложение с HTML-разметкой, разработанное для непосредственного использования пользователями –* если приложение используется пользователями напрямую, как в традиционном веб-

приложении, то Razor Pages отвечает за создание веб-страниц, с которыми взаимодействует пользователь. Они обрабатывают запросы URL-адресов, получают данные, отправленные с помощью форм, и генерируют HTML-код, который пользователи используют для просмотра и навигации по приложению;

- *веб-приложение с HTML-разметкой и API* – также возможно наличие приложений, которые удовлетворяют обеим потребностям, что может позволить обслуживать более широкий круг клиентов, разделяя логику в приложении.

В этой главе вы узнаете, как ASP.NET Core использует Razor Pages для второго из этих вариантов, создания HTML-страниц с отрисовкой на стороне сервера. Мы сразу же начнем с того, что используем шаблон для создания простого приложения Razor Pages и сравним его функции с приложениями с минимальными API, которые вы видели до сих пор. В разделе 13.2 мы рассмотрим более сложный пример страницы Razor.

Далее в разделе 13.3 мы сделаем шаг назад и посмотрим на паттерн проектирования MVC. Я расскажу о некоторых преимуществах использования этого паттерна, и вы узнаете, почему так много веб-фреймворков используют его в качестве модели для создания удобных в сопровождении приложений.

В разделе 13.4 вы узнаете, как паттерн проектирования MVC применяется в ASP.NET Core. Данный паттерн – это общая концепция, которую можно применять в различных ситуациях, но в ASP.NET Core он используется как абстракция пользовательского интерфейса. Вы увидите, как Razor Pages реализует паттерн проектирования MVC и как он строится поверх фреймворка ASP.NET Core MVC.

В этой главе я постараюсь подготовить вас к каждой из предстоящих тем, но, возможно, вы обнаружите, что на данном этапе некоторые действия покажутся вам немного магическими. Страйтесь не слишком заботиться о том, как именно все части Razor Pages связаны друг с другом; сосредоточьтесь на конкретных рассматриваемых концепциях и на том, как они связаны с концепциями, с которыми вы уже познакомились. Мы начнем с создания приложения Razor Pages.

13.1 Наше первое приложение Razor Pages

В этом разделе мы начнем работу с Razor Pages, создав новое приложение на основе шаблона. После того как мы создадим приложение и изучим его, мы рассмотрим сходства и различия по сравнению с приложением с минимальным API. Вы узнаете о дополнительном промежуточном ПО, добавляемом шаблоном по умолчанию, посмотрите, как генерируется HTML-код с помощью Razor Pages, и узнаете о Razor Page-эквиваленте обработчиков конечных точек минимальных API: обработчиках страниц.

13.1.1 Использование шаблона «Веб-приложение»

Использование шаблона – это быстрый способ запустить приложение, поэтому мы воспользуемся этим подходом, используя шаблон

ASP.NET Core «Веб-приложение». Чтобы создать приложение Razor Pages в Visual Studio, выполните следующие действия:

- 1 Выберите **Create a New Project** (Создать новый проект) на заставке или **File > New > Project** (Файл > Создать > Проект) на главном экране Visual Studio.
- 2 В списке шаблонов выберите **ASP.NET Core Web App**, убедившись, что выбран шаблон языка C#.
- 3 На следующем экране введите имя проекта, местоположение и имя решения и нажмите **Next** (Далее). Например, в качестве имени проекта и решения можно использовать `WebApplication1`.
- 4 На следующем экране (рис. 13.1) выполните следующие действия:
 - выберите .NET 7.0. Если этот параметр недоступен, убедитесь, что у вас установлен .NET 7. Подробную информацию о настройке окружения см. в приложении А;
 - убедитесь, что установлен флагок **Configure for HTTPS** (Настроить для HTTPS);
 - убедитесь, что флагок **Enable Docker** (Включить Docker) снят;
 - убедитесь, что флагок **Do Not Use Top-level Statements** (Не использовать операторы верхнего уровня) снят;
 - выберите **Create** (Создать).

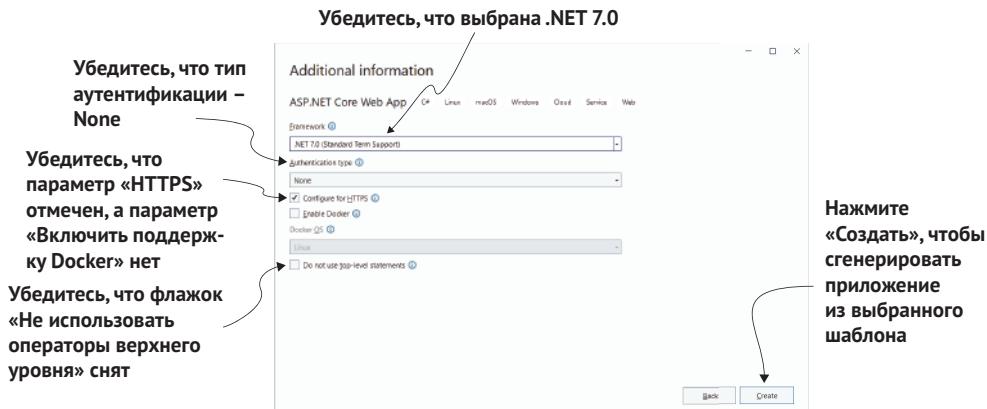


Рис. 13.1 Экран дополнительной информации. Он идет после диалогового окна «Настройка нового проекта» и позволяет настроить шаблон, который генерирует приложение

Если вы не используете Visual Studio, то можете создать аналогичный шаблон с помощью интерфейса командной строки .NET. Создайте папку для хранения нового проекта. Откройте командную строку PowerShell или cmd в папке (в OC Windows) или сеансе терминала (в Linux или macOS) и выполните команды из следующего листинга.

Листинг 13.1. Создание нового приложения Razor Pages с помощью интерфейса командной строки .NET

```
dotnet new sln -n WebApplication1      ← Создает файл решения  

dotnet new razor -o WebApplication1    ← Создает проект ASP.NET Core Razor Pages  

dotnet sln add WebApplication1        ← во вложенной папке WebApplication1
```

Добавляет новый проект в файл решения

Независимо от того, используете вы Visual Studio или интерфейс командной строки .NET, теперь вы можете собрать и запустить приложение. Нажмите клавишу **F5**, чтобы запустить приложение с помощью Visual Studio, или воспользуйтесь командой `dotnet run` в папке проекта. Эта команда открывает соответствующий URL-адрес в веб-браузере и отображает базовую страницу приветствия, как показано на рис. 13.2.

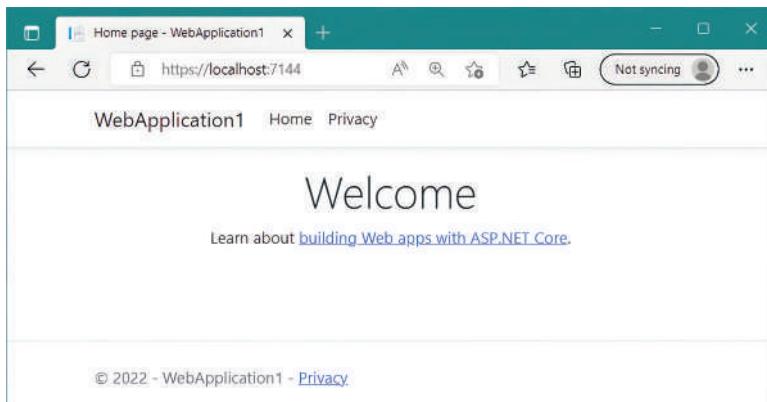


Рис. 13.2 Так выглядит наше новое приложение Razor Pages. Шаблон выбирает случайный порт для URL-адреса, который автоматически открывается в браузере при запуске приложения

По умолчанию на этой странице отображаются простой баннер приветствия и ссылка на официальную документацию Microsoft для ASP.NET Core. Вверху страницы находятся две ссылки: **Home** и **Privacy**. Ссылка **Home** – это страница, на которой вы сейчас находитесь. Нажав на ссылку **Privacy**, вы перейдете на новую страницу, показанную на рис. 13.3. Как вы увидите в разделе 13.1.3, можно использовать Razor Pages в своем приложении для определения этих двух страниц и создания HTML-кода, который они отображают.

На данном этапе вы должны заметить пару вещей:

- заголовок, содержащий ссылки и название приложения, `WebApplication1`, одинаков на обеих страницах;
- заголовок страницы, показанный на вкладке браузера, изменяется в соответствии с текущей страницей. Вы увидите, как реализовать эти возможности, в главе 17, когда мы будем обсуждать отрисовку HTML с использованием шаблонов Razor.

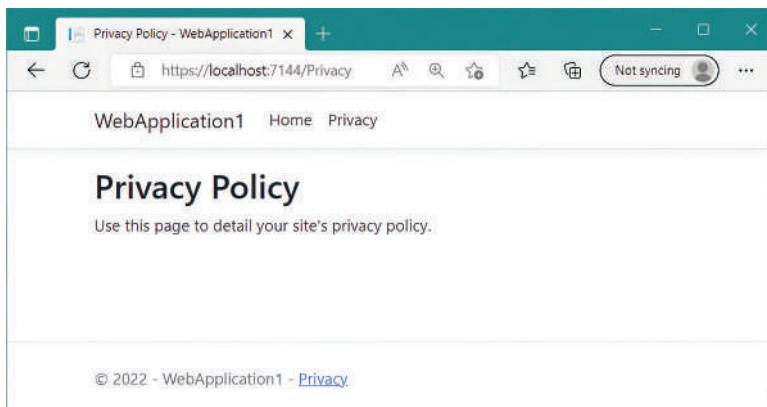


Рис. 13.3 Страница Privacy. Можно перемещаться между двумя страницами приложения, используя ссылки Home и Privacy в заголовке приложения. Приложение генерирует содержимое страниц с помощью Razor Pages

Пока от пользовательского интерфейса приложения больше ничего не требуется. Пощелкайте мышью и, если вас устроит поведение приложения, вернитесь в редактор и просмотрите файлы, включенные в шаблон.

У этого приложения Razor Pages почти такая же структура, как и у приложения минимальных API, которые мы создавали в этой книге, как показано на рис. 13.4. Общая структура идентична, за исключением двух дополнительных папок, которые вы раньше не видели:

- папка *Pages* – эта папка содержит файлы Razor Pages, которые определяют различные страницы веб-приложения, включая страницы **Home** и **Privacy**, которые вы уже видели;
- папка *wwwroot* – эта папка уникальна тем, что это единственная папка в приложении, к которой браузерам разрешен прямой доступ при просмотре веб-приложения. Здесь вы можете хранить файлы CSS, JavaScript, изображения или статические HTML-файлы, а компонент обработки статических файлов будет предоставлять их браузерам по запросу. Шаблон создает вложенные папки в папке *wwwroot*, но вам не обязательно их использовать; вы можете структурировать свои статические файлы в *wwwroot* по своему усмотрению.

Помимо этих дополнительных файлов, единственное различие между приложением Razor Pages и приложением минимальных API – это файл *Program.cs*. В разделе 13.1.2 вы увидите, что приложение Razor Pages использует ту же базовую структуру файла *Program.cs*, но добавляет дополнительные сервисы и промежуточное ПО, используемое в типичном приложении Razor Pages.

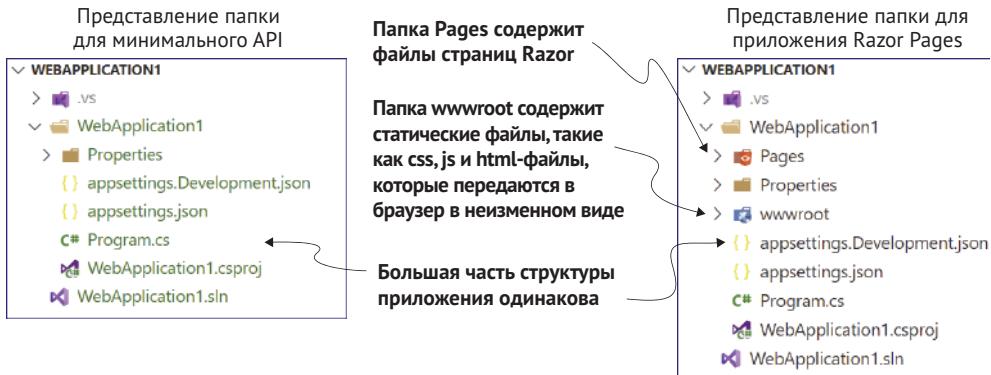


Рис. 13.4 Сравнение структуры проекта приложения минимальных API с приложением Razor Pages. Приложение Razor Pages содержит все те же файлы и папки, а также папку Pages для определений страниц Razor и папку wwwroot для статических файлов, которые передаются непосредственно в браузер

13.1.2 Добавление и настройка сервисов

Одна из приятных особенностей работы с приложениями ASP.NET Core заключается в том, что код установки очень похож даже для совершенно разных моделей приложений. Независимо от того, создаете вы приложение Razor Pages или используете минимальные API, файл Program.cs содержит все те же шесть шагов:

- 1 создание экземпляра `WebApplicationBuilder`;
- 2 регистрация необходимых сервисов в этом экземпляре;
- 3 вызов метода `Build()` экземпляра построителя, чтобы создать экземпляр `WebApplication`;
- 4 добавление промежуточного ПО в `WebApplication`, чтобы создать конвейер;
- 5 привязка конечных точек приложения;
- 6 вызов метода `Run()` класса `WebApplication`, чтобы запустить сервер и обрабатывать запросы.

В следующем листинге показан файл Program.cs для приложения Razor Pages. В этом файле используется гораздо больше компонентов, чем вы видели ранее, но общая структура должна быть вам знакома.

Листинг 13.2. Файл Program.cs для приложения Razor Pages

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages(); <-- Регистрирует необходимые сервисы для использования функции Razor Pages
WebApplication app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

Добавляет промежуточное ПО в зависимости от среды выполнения

```
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();
```

В конвейер промежуточного программного обеспечения можно добавить дополнительное ПО

```
app.MapRazorPages(); ←
app.Run();
```

Регистрирует каждую страницу Razor в качестве конечной точки приложения

В главе 4 вы узнали о промежуточном ПО и важности порядка при добавлении этого ПО в конвейер. В этом примере в конвейер добавляется шесть компонентов, два из которых добавляются только тогда, когда приложение *не* выполняется в окружении разработки:

- `ExceptionHandlerMiddleware` – с этим компонентом мы познакомились в главах 4 и 5. Он перехватывает исключения, создаваемые промежуточным ПО на последующих этапах конвейера, и генерирует понятную страницу ошибок;
- `HstsMiddleware` – этот компонент задает заголовки безопасности в ответе в соответствии с лучшими отраслевыми практиками. Подробную информацию об этом и других компонентах, связанных с безопасностью, см. в главе 28;
- `HttpsRedirectionMiddleware` – данный компонент гарантирует, что ваше приложение отвечает только на безопасные (HTTPS) запросы, и является лучшей отраслевой практикой. Мы рассмотрим протокол HTTPS в главе 28;
- `StaticFileMiddleware` – как было показано в главе 4, этот компонент обслуживает запросы статических файлов (например, файлов с расширениями .css и .js) из папки `wwwroot`;
- `RoutingMiddleware` – компонент маршрутизации отвечает за выбор конечной точки для входящего запроса. `WebApplication` добавляет его по умолчанию, но, как обсуждалось в главе 4, его явное добавление гарантирует, что он будет выполняться после `StaticFileMiddleware`;
- `AuthorizationMiddleware` – этот компонент контролирует, разрешен ли запуск конечной точки в зависимости от пользователя, выполняющего запрос, но требует также настройки аутентификации для вашего приложения. Подробнее об аутентификации рассказывается в главе 23, а об авторизации – в главе 24.

Помимо промежуточного ПО, добавляемого явно, `WebApplication` автоматически добавляет некоторые компоненты (как обсуждалось в главе 4), например `EndpointMiddleware`, которые автоматически добавляются в конец конвейера промежуточного ПО. Как и в случае с минимальными API, `RoutingMiddleware` выбирает, какой обработчик конечной точки выполнить, а `EndpointMiddleware` выполняет обработчик для генерации ответа.

Вместе эта пара отвечает за интерпретацию запроса, определяя, какую страницу Razor вызывать, за чтение параметров из запроса и за создание окончательного HTML-кода. Требуется небольшая настройка; нужно лишь добавить промежуточное ПО в конвейер и указать,

что вы хотите использовать конечные точки Razor Pages, вызвав метод `MapRazorPages`. Для каждого запроса компонент маршрутизации использует URL-адрес запроса, чтобы определить, какую страницу Razor вызвать. Затем компонент конечной точки выполняет страницу Razor для генерации ответа в виде HTML.

Когда приложение настроено, оно может начать обрабатывать запросы. Но *как* оно это делает? В разделе 13.1.3 вы узнаете, как Razor Pages генерирует HTML.

13.1.3 Создание HTML с помощью Razor Pages

Когда приложение ASP.NET Core получает запрос, оно проходит через конвейер промежуточного ПО до тех пор, пока его не обработает нужный компонент. Обычно компонент маршрутизации сопоставляет путь URL-адреса запроса с настроенным маршрутом, который определяет, какую страницу Razor вызвать, а компонент конечной точки вызывает его.

Страницы Razor хранятся в файлах с расширением .cshtml (смеси файлов .cs и .html) в папке Pages вашего проекта. Как правило, компонент маршрутизации сопоставляет пути URL-адресов запросов с одной страницей Razor, выполняя поиск страницы с тем же путем в папке Pages. Например, на рис. 13.3 видно, что страница приложения **Privacy** соответствует пути /Privacy в адресной строке браузера. Если вы заглянете в папку Pages, то найдете там файл Privacy.cshtml, показанный в следующем листинге.

Листинг 13.3. Страница Razor – Privacy.cshtml

```
@page           ←———— Указывает на то, что это страница Razor
@model PrivacyModel ←———— Связывает страницу Razor с определенной моделью страницы
 @{
    ViewData["Title"] = "Privacy Policy"; ←———— Код C#, который не пишется в ответ
}
<h1>@ViewData["Title"]</h1> ←———— HTML-код с динамическими значениями C#, которые пишутся в ответ

```

► <p>Use this page to detail your site's privacy policy.</p>

Автономный статический HTML-код

Страницы Razor используют синтаксис шаблонов под названием *Razor*, сочетающий статический HTML-код с динамическим кодом на языке C# и генерацией HTML-кода. Директива `@page` в первой строке страницы Razor является самой важной. Она всегда должна размещаться в первой строке файла, поскольку сообщает ASP.NET Core, что файл .cshtml – это страница Razor. Без нее вы не сможете правильно просматривать страницу.

Следующая строка определяет, с какой моделью страницы в проекте ассоциирована страница Razor:

```
@model PrivacyModel
```

В данном случае модель страницы называется `PrivacyModel` и следует стандартному соглашению по именованию моделей страниц Razor. Этот класс можно найти в файле `Privacy.cshtml.cs` в папке `Pages` вашего проекта, как показано на рис. 13.5. Visual Studio отображает эти файлы вложенными в файлы страниц Razor с расширением `.cshtml` в обозревателе решений. Мы рассмотрим модель страницы в следующем разделе.

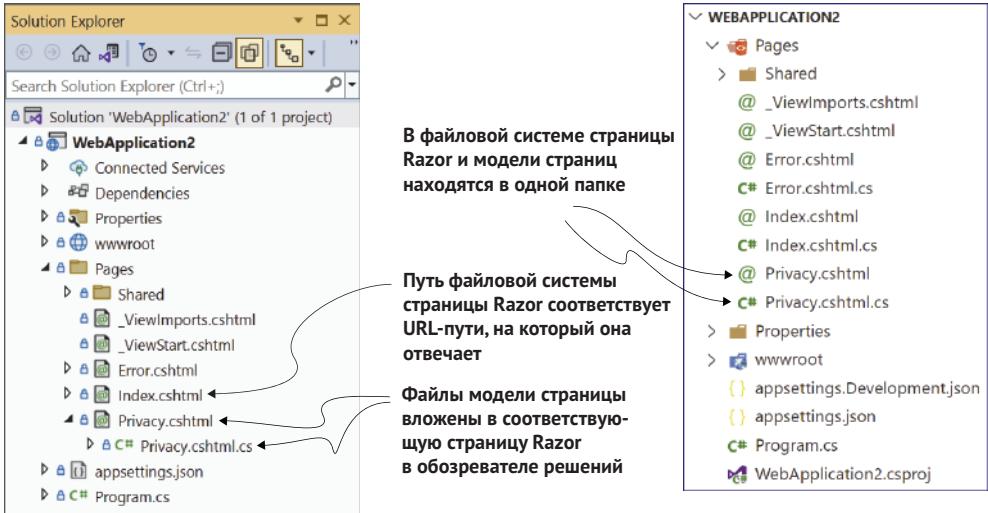


Рис. 13.5 По соглашению модели страниц помещаются в файл с тем же именем, что и страница Razor. При этом к имени добавляется суффикс `.cs`. Visual Studio размещает эти файлы под страницей Razor в Обозревателе решений

Помимо директив `@page` и `@model`, видно, что статический HTML-код всегда является валидным на странице Razor и будет отображаться в ответе «как есть».

```
<p>Use this page to detail your site's privacy policy.</p>
```

Вы также можете писать обычный код на языке C# в шаблонах Razor, используя следующую конструкцию:

```
@{ /* здесь идет код C# */ }
```

Любой код, заключенный в фигурные скобки, будет выполнен, но не будет записан в ответ. В листинге мы задаем заголовок страницы, записывая значение ключа в словарь `ViewData`, но на данный момент мы ничего не пишем в ответ:

```
@{
    ViewData["Title"] = "Privacy Policy";
}
```

Еще одна функция, показанная в этом шаблоне, заключается в том, что вы можете динамически записывать переменные C# в HTML-поток с помощью символа `@`. Способность сочетать динамическую

и статическую разметки – вот в чем сила страниц Razor. В этом примере вы получаете значение "Title" из словаря ViewData с последующей его записью внутри тега <h1>:

```
<h1>@ViewData["Title"]</h1>
```

На данном этапе вас, возможно, немного может сбить с толку шаблон из листинга 13.3, если сравнить его с выводом, показанным на рис. 13.3. Заголовок и статическое HTML-содержимое отображаются как в листинге, так и на рисунке, но некоторые части конечной веб-страницы не видны в шаблоне. Как такое может быть?

Страницы Razor имеют концепцию макетов, которые представляют собой «базовые» шаблоны, определяющие общие элементы приложения, такие как верхний колонтитул (шапка) и нижний (подвал). HTML-код макета в сочетании с шаблоном страницы Razor создает окончательный HTML-код, который отправляется в браузер. Это избавляет вас от необходимости дублировать код шапки и подвала на каждой странице, а также означает, что если вам нужно что-то настроить, достаточно сделать это только в одном месте.

ПРИМЕЧАНИЕ Я подробно расскажу о шаблонах Razor, включая макеты, в главе 17. Макеты можно найти в папке проекта Pages/Shared.

Как вы уже видели, вы можете включить код на языке C# в страницы Razor, используя фигурные скобки @{}, но, как правило, нужно ограничивать код в файле .cshtml только функциями, используемыми в представлении. Сложная логика, код для доступа к таким сервисам, как база данных, и манипулирование данными должны обрабатываться в модели страницы.

13.1.4 Логика обработки запросов с помощью моделей страницы и обработчиков

Как вы уже видели, директива @page в файле с расширением .cshtml помечает страницу как страницу Razor, но большинство страниц Razor также имеют связанную модель страницы. По соглашению она помещается в файл, обычно известный как *файл кода программной части*, который имеет расширение .cs, как вы видели на рис. 13.5. Модели страниц должны наследовать от базового класса PageModel и обычно содержат один или несколько методов, которые называются *обработчиками страниц*. Они определяют, как обрабатывать запросы к странице Razor.

ОПРЕДЕЛЕНИЕ Обработчик страницы – это метод, который запускается в ответ на запрос. Модели страниц Razor должны наследовать от класса PageModel. Они могут содержать несколько обработчиков страниц, хотя обычно там только один или два обработчика. В следующем листинге показана модель страницы для Privacy.cshtml, которая находится в файле Privacy.cshtml.cs.

Листинг 13.4 PrivacyModel в Privacy.cshtml.cs – модель страницы Razor Page

```
public class PrivacyModel: PageModel { ← | Razor Pages должны наследовать от PageModel
{
    private readonly ILogger<PrivacyModel> _logger; ← | Вы можете использовать внедрение зависимостей для предоставления сервисов в конструктор
    public PrivacyModel(ILogger<PrivacyModel> logger)
    {
        _logger = logger;
    }

    public void OnGet() ← | Обработчик страницы по умолчанию – OnGet. Возврат void указывает на то, что должен быть сгенерирован HTML
    {
    }
}
```

Эта очень простая модель страницы, но она демонстрирует несколько важных моментов:

- обработчики страниц именуются по соглашению;
- модели страниц могут использовать внедрение зависимостей для взаимодействия с другими сервисами.

Обработчики страниц обычно именуются по соглашению, на основе HTTP-метода, на который они отвечают. Они возвращают либо `void`, что указывает на необходимость отрисовки шаблона страницы Razor, либо `IActionResult`, содержащий другие инструкции для генерации ответа, например перенаправление пользователя на другую страницу.

`PrivacyModel` содержит единственный обработчик `OnGet`, который указывает, что он должен выполняться в ответ на запросы методом GET. Поскольку метод возвращает `void`, выполнение обработчика выполнит ассоциированный шаблон Razor для страницы, чтобы сгенерировать HTML-код.

ПРИМЕЧАНИЕ Razor Pages ориентированы на создание приложений на основе страниц, поэтому обычно требуется возвращать HTML-код, а не ответ в формате JSON или XML. Однако вы также можете использовать `IActionResult` для возврата любых данных, перенаправления пользователей на новую страницу или для отправки ошибки. Подробнее об `IActionResult` вы узнаете в главе 15.

Внедрение зависимостей используется для внедрения экземпляра `ILogger<PrivacyModel>` в конструктор модели страницы так же, как вы внедряете сервис в обработчик конечной точки минимального API. В этом примере сервис не используется, но все о `ILogger` вы узнаете в главе 26.

Очевидно, что модель страницы `PrivacyModel` в этом случае мало что дает, и вам может быть интересно, зачем она нужна. Если все, что модели делают, – это говорят странице Razor сгенерировать HTML-код, тогда зачем вообще нужны эти модели?

Здесь важно помнить, что теперь у вас есть фреймворк для выполнения произвольно сложных функций в ответ на запрос. Вы можете легко изменить метод обработчика, чтобы загрузить данные из базы данных, отра-

вить электронное письмо, добавить продукт в корзину или создать счет – все в ответ на простой HTTP-запрос. В этой расширяемости и заключается большая часть возможностей страниц Razor (и паттерна MVC в целом).

Другой важный момент заключается в том, что вы отделили выполнение этих методов от генерации самого HTML-кода. Если логика изменится и вам понадобится добавить поведение для обработчика страницы, то не нужно будет ничего менять в коде генерации HTML, поэтому вы с меньшей вероятностью внесете ошибки. И наоборот, если вам нужно немного изменить пользовательский интерфейс, например изменить цвет заголовка, тогда логике вашего метода обработчика ничего не угрожает.

И вот оно, готовое приложение ASP.NET Core, созданное с помощью Razor Pages! Прежде чем продолжить, в последний раз посмотрим, как наше приложение обрабатывает запрос. На рис. 13.6 показан запрос к пути /Privacy, обрабатываемый приложением. Здесь вы уже все видели, поэтому процесс обработки запроса должен быть вам знаком. Он показывает, как запрос проходит через конвейер промежуточного ПО, прежде чем будет обработан компонентом конечной точки. Страница Razor Privacy.cshtml выполняет обработчик `OnGet` и генерирует ответ в виде HTML-кода, который возвращается через промежуточное ПО к веб-серверу ASP.NET Core перед отправкой в браузер пользователя.

На этом данный раздел подошел к концу, и теперь у вас сложилось четкое представление о том, как настраивается все приложение Razor Pages и как оно обрабатывает запросы с помощью Razor Pages. В разделе 13.2 мы продолжим рассматривать базовые страницы Razor в шаблоне по умолчанию и изучим более сложный пример.

13.2 Изучение типичной страницы Razor

Модель программирования Razor Pages была представлена в ASP.NET Core 2.0 как способ создания многостраничных веб-сайтов с отрисовкой на стороне сервера. Она построена на основе инфраструктуры ASP.NET Core, чтобы обеспечить оптимизированный опыт, по возможности с использованием соглашений, дабы уменьшить необходимое количество стандартного кода и конфигурации. В этом разделе мы рассмотрим более сложную модель страницы, чтобы лучше понять общий дизайн Razor Pages.

В листинге 13.4 мы рассмотрели простую страницу Razor. Она не содержала никакой логики, а просто отображала связанное представление Razor. Такой подход может быть распространен, если вы создаете, например, маркетинговый сайт с большим количеством контента, но чаще всего ваши страницы Razor будут содержать некую логику, загружать данные из базы данных или использовать формы, чтобы позволить пользователям отправлять информацию.

Чтобы лучше понять, как работают типичные страницы Razor, в этом разделе мы кратко рассмотрим страницу посложнее. Эту страницу мы взяли из приложения со списком дел. Она используется для отображения всех дел для данной категории. На этом этапе мы не фокусируемся на генерации HTML-кода, поэтому в следующем листинге показан только код программной части модели страницы Razor.

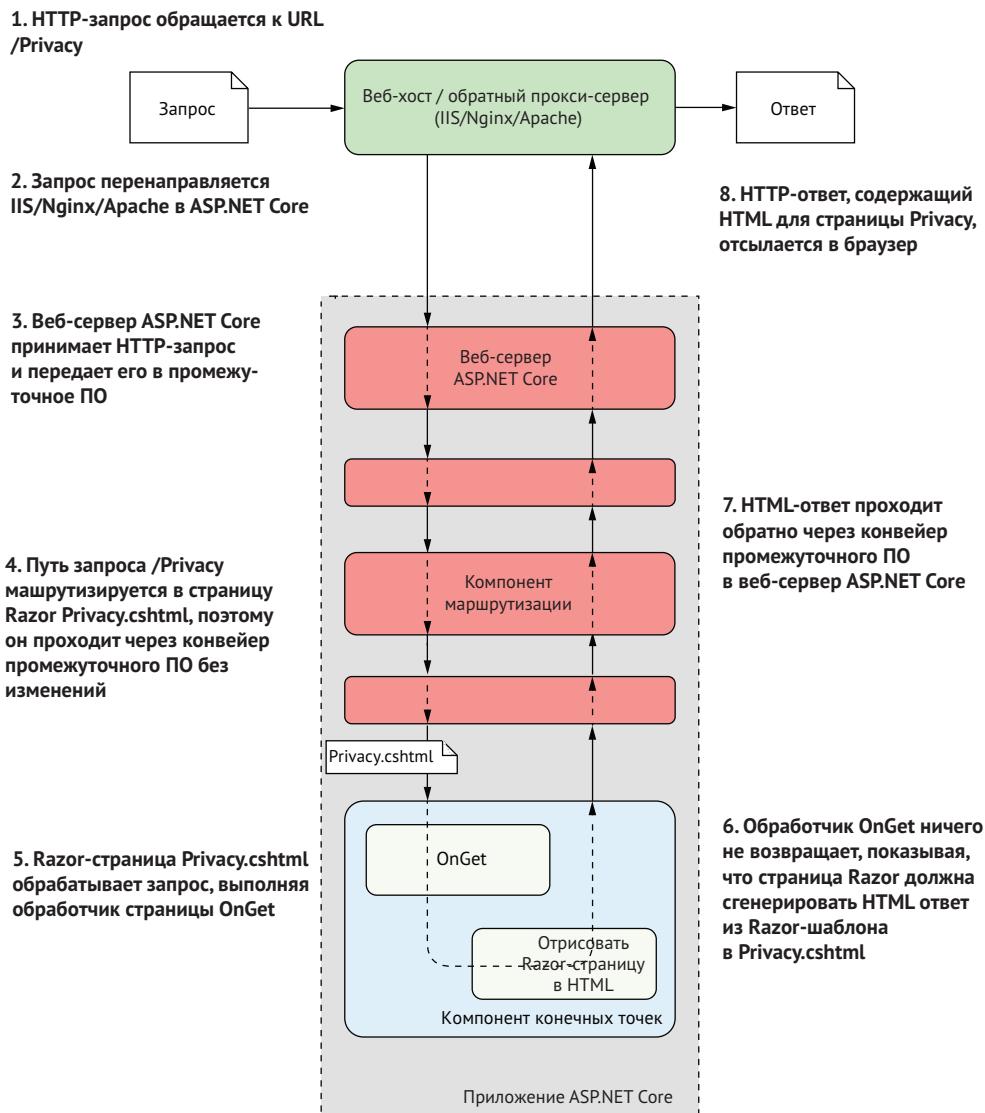


Рис. 13.6 Обзор запроса к URL-адресу /Privacy для приложения ASP.NET, созданного с использованием Razor Pages. Компонент маршрутизации направляет запрос обработчику OnGet файла Privacy.cshtml.cs. Страница Razor генерирует ответ в виде HTML-кода, выполняя шаблон Razor в файле Privacy.cshtml, и передает ответ обратно по конвейеру в браузер

Листинг 13.5. Страница Razor для просмотра всех дел в заданной категории

```
public class CategoryModel : PageModel
{
    private readonly ToDoService _service;
    public CategoryModel(ToDoService service)
```

ToDoService предоставляется в конструктор модели с использованием внедрения зависимостей

Обра-
ботчик
OnGet
прини-
мает па-
раметр
category

```
{
    _service = service;
}

public ActionResult OnGet(string category)
{
    Items = _service.GetItemsForCategory(category);
    return Page();
}

public List<ToDoListModel> Items { get; set; }
}
```

Обработчик вызывает ToDoService для получения данных и задает свойство Items

Возвращає PageResult, указывающий, что должно быть отображено представление Razor

Представление Razor может получить доступ к свойству Items при его визуализации

Это по-прежнему относительно простой пример, однако он демонстрирует множество функций по сравнению с базовым примером из листинга 13.4:

- обработчик страницы OnGet принимает параметр метода category. Этот параметр автоматически заполняется инфраструктурой Razor Pages значениями из входящего запроса посредством привязки модели, аналогично тому, как работает привязка в минимальных API. Привязка модели подробно обсуждается в главе 16;
- обработчик не взаимодействует с базой данных напрямую. Вместо этого он использует предоставленное значение category для взаимодействия с ToDoService, который внедряется как аргумент конструктора с использованием внедрения зависимостей;
- обработчик возвращает Page() в конце метода, чтобы указать, что должно быть отрисовано связанное представление Razor. В этом случае оператор return фактически является необязательным; по соглашению, если обработчик страницы – это метод void, представление Razor все равно будет отображаться, как если бы вы вызвали return Page() в конце метода;
- представление Razor имеет доступ к экземпляру CategoryModel, поэтому оно может получить доступ к свойству Items, установленному обработчиком. Оно использует эти элементы для создания HTML-кода, который в конечном итоге отправляется пользователю.

Паттерн взаимодействий на странице Razor из листинга 13.5 демонстрирует распространенный подход. Обработчик страницы является центральным контроллером страницы Razor. Он получает входные данные от пользователя (параметр метода category), обращается к «мозгам» приложения (ToDoService) и передает данные (предоставляя доступ к свойству Items) представлению Razor, которое генерирует в ответ HTML-код. Если приглядеться, то это похоже на паттерн проектирования «модель–представление–контроллер» (MVC).

В зависимости от вашего опыта в разработке программного обеспечения вы, возможно, ранее сталкивались с этим паттерном в той или иной форме. В веб-разработке MVC является распространенной парадигмой и используется в таких фреймворках, как Django, Rails и Spring MVC. Но поскольку это очень обширная концепция, вы можете най-

ти MVC везде, от мобильных приложений до полнофункциональных клиентских настольных приложений. Надеюсь, это свидетельствует о пользе, которую данный паттерн может принести при правильном использовании! В следующем разделе мы рассмотрим паттерн MVC в общих чертах и то, как он используется ASP.NET Core.

13.3 Паттерн проектирования MVC

MVC – это распространенный паттерн для проектирования приложений с пользовательским интерфейсом. Исходный паттерн MVC имеет множество различных интерпретаций, каждая из которых фокусируется на разных аспектах. Например, исходный паттерн проектирования MVC был определен с учетом полнофункциональных клиентских приложений с графическим интерфейсом пользователя (GUI), а не веб-приложений, поэтому в нем используются терминология и парадигмы, связанные со средой графического интерфейса пользователя. По сути, однако, этот паттерн направлен на отделение управления данными и манипулирования данными от их визуального представления.

Прежде чем мы подробно изучим сам паттерн проектирования, рассмотрим типичный запрос. Представьте, что пользователь запрашивает страницу Razor из листинга 13.5, в котором отображена категория списка дел. На рис. 13.7 показано, как страница Razor обрабатывает различные аспекты запроса, чтобы генерировать окончательный ответ.

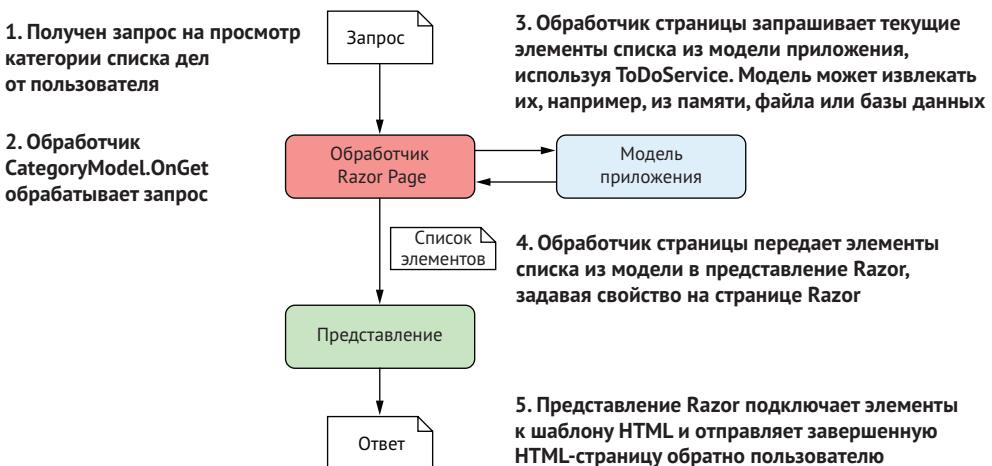


Рис. 13.7 Получение страницы списка дел приложения Razor. Различные компоненты обрабатывают различные аспекты запроса

В целом паттерн проектирования MVC состоит из трех компонентов:

- *модель* – это данные, которые необходимо отобразить, глобальное состояние приложения. Доступ к нему осуществляется через ToDoService в листинге 13.5;
- *представление* – шаблон, отображающий данные, предоставленные моделью;

- *контроллер* – обновляет модель и предоставляет данные для отображения в представлении. Этую роль выполняет обработчик страниц в Razor Pages. Это метод `OnGet` из листинга 13.5.

Каждый компонент в паттерне проектирования MVC отвечает за отдельный аспект всей системы, которые при объединении можно использовать для создания пользовательского интерфейса. В примере со списком дел MVC рассматривается с точки зрения веб-приложения, использующего Razor Pages, но запрос также может быть эквивалентен щелчку кнопки в настольном приложении с графическим интерфейсом пользователя.

В целом порядок событий, когда приложение отвечает на взаимодействие с пользователем или запрос, следующий:

- 1 контроллер (обработчик страницы Razor) получает запрос;
- 2 в зависимости от запроса контроллер либо извлекает запрошенные данные из модели приложения, используя внедренные сервисы, либо обновляет данные, образующие модель;
- 3 контроллер выбирает представление для отображения и передает ему модель представления;
- 4 представление использует данные, содержащиеся в модели, для создания пользовательского интерфейса.

Когда мы описываем MVC в этом формате, контроллер (обработчик страницы Razor) служит точкой входа для взаимодействия. Пользователь связывается с контроллером, чтобы инициировать взаимодействие.

В веб-приложениях это взаимодействие принимает форму HTTP-запроса, поэтому, когда запрос на URL-адрес получен, контроллер обрабатывает его.

В зависимости от характера запроса контроллер может выполнять различные действия, но ключевым моментом является то, что действия выполняются с использованием модели приложения. Модель здесь содержит всю бизнес-логику приложения, поэтому оно может предоставлять запрашиваемые данные или выполнять действия.

ПРИМЕЧАНИЕ В этом описании MVC модель рассматривается как непростой зверь, содержащий всю логику выполнения действия, а также любое внутреннее состояние. Класс Razor Page `PageModel` – это не та модель, о которой мы говорим! К сожалению, как и во всей разработке программного обеспечения, названия – вещь непростая.

Рассмотрим запрос на просмотр страницы товара для приложения электронной торговли. Контроллер получит запрос и будет знать, как связаться с каким-то сервисом товаров, который является частью модели приложения. Сервис может получить сведения о запрашиваемом товаре из базы данных и вернуть их контроллеру.

В качестве альтернативы представьте, что контроллер получает запрос на добавление товара в корзину пользователя. Контроллер получит запрос и, скорее всего, вызовет метод модели, чтобы запросить добавление товара. Затем модель обновит свое внутреннее представление корзины пользователя, добавив, например, новую строку в таблицу базы данных, содержащую данные пользователя.

СОВЕТ Можно рассматривать каждый обработчик страниц Razor как мини-контроллер, ориентированный на одну страницу. Каждый веб-запрос – это независимый вызов контроллера, который координирует получение ответа. Хотя существует много разных контроллеров, все обработчики взаимодействуют с одной и той же моделью приложения.

После обновления модели контроллеру необходимо решить, какой ответ сгенерировать. Одним из преимуществ использования паттерна проектирования MVC является то, что модель, представляющая данные приложения, отделена от окончательного внешнего вида этих данных, называемого *представлением*. Контроллер отвечает за принятие решения о том, должен ли ответ генерировать HTML-представление, нужно ли отправлять пользователя на новую страницу или нужно вернуть страницу с сообщением об ошибке.

Одним из преимуществ независимости модели от представления является то, что это улучшает тестируемость. Код пользовательского интерфейса классически сложно протестировать, поскольку он зависит от окружения – любой, кто писал тесты пользовательского интерфейса, имитирующие нажатие пользователем кнопок и ввод данных в формы, знает, что обычно они хрупкие. Сохраняя модель независимой от представления, вы можете гарантировать, что модель станет легко тестируемой, без каких-либо зависимостей от конструкций пользовательского интерфейса. Поскольку модель часто содержит бизнес-логику приложения, это очевидно хорошо!

Представление может использовать данные, передаваемые ему контроллером, для генерации соответствующего ответа в виде HTML-кода. Представление отвечает только за генерацию окончательного представления данных; оно не участвует ни в какой бизнес-логике.

Это все, что касается паттерна проектирования MVC, если говорить о веб-приложениях. Большая часть путаницы, связанной с MVC, по-видимому, происходит из-за нескольких разных применений этого термина для разных фреймворков и типов приложений. В следующем разделе я покажу, как фреймворк ASP.NET Core использует паттерн MVC с Razor Pages, а также другие примеры этого паттерна в действии.

13.4 Применение паттерна проектирования MVC к Razor Pages

В предыдущем разделе мы обсуждали паттерн MVC, который обычно используется в веб-приложениях; он используется и в Razor Pages. Но в ASP.NET Core также есть фреймворк под названием ASP.NET Core MVC. Этот фреймворк (что неудивительно) очень точно отражает паттерн проектирования MVC, используя контроллеры и методы действий вместо Razor Pages и обработчиков страниц. Razor Pages строится непосредственно поверх базового фреймворка ASP.NET Core MVC, используя под капотом фреймворк MVC для реализации своего поведения.

При желании вы можете полностью отказаться от использования Razor Pages и работать с фреймворком MVC непосредственно в ASP.NET Core. В ранних версиях ASP.NET Core и предыдущей версии ASP.NET это был единственный вариант.

НА ЗАМЕТКУ Выбор между Razor Pages и фреймворком MVC будет более подробно рассматриваться в главе 19.

В этом разделе мы по пунктам рассмотрим, как паттерн проектирования MVC применяется к Razor Pages в ASP.NET Core. Это также поможет прояснить роль различных функций Razor Pages.

MVC или MVVM – что используется в Razor Pages?

Иногда я видел, как некоторые утверждают, что Razor Pages использует паттерн проектирования «модель – представление – модель представления» (MVVM), а не паттерн MVC. Лично я не согласен с этим, но стоит помнить о различиях.

MVVM – это паттерн пользовательского интерфейса, который часто используется в мобильных и настольных приложениях и некоторых клиентских фреймворках. Его отличие от MVC состоит в том, что между представлением и моделью представления существует двунаправленное взаимодействие. Модель представления сообщает представлению, что отображать, но представление также может инициировать изменения непосредственно в модели представления. Он часто используется с двусторонней привязкой данных, когда модель представления привязана к представлению.

Некоторые считают, что в Razor Pages эту роль выполняет модель страницы, но я не уверен. Мне определенно кажется, что Razor Pages основан на паттерне MVC (в конце концов, он основан на фреймворке ASP.NET Core MVC!), и у него нет такой же двусторонней привязки, которую я ожидал бы в случае с MVVM.

Как вы видели в предыдущих главах, ASP.NET Core реализует конечные точки Razor Page, используя сочетание компонентов `RoutingMiddleware` и `EndpointMiddleware`, как показано на рис. 13.8. Как только запрос был обработан более ранним компонентом (и при условии что ни один из них не обработал запрос и не замкнул конвейер), компонент маршрутизации выберет, какой обработчик страницы Razor должен быть выполнен, а компонент конечной точки выполнит обработчик страницы.

Как было показано в предыдущих главах, промежуточное ПО часто обрабатывает сквозную функциональность или узко определенные запросы, например запросы файлов. Для требований, выходящих за рамки этих функций или имеющих много внешних зависимостей, требуется более надежный фреймворк. Razor Pages (и/или ASP.NET Core MVC) может предоставить такой фреймворк, позволяя взаимодействовать с основной бизнес-логикой вашего приложения и создавать пользовательский интерфейс. Он обрабатывает все, от сопоставления запроса с соответствующим контроллером до генерации HTML- или API-ответов.

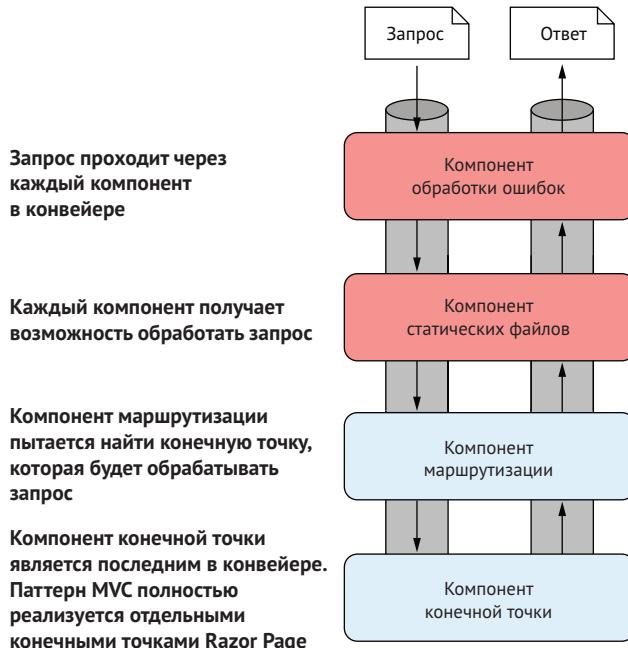


Рис. 13.8 Конвейер промежуточного ПО типичного приложения ASP.NET Core. Запрос обрабатывается каждым компонентом последовательно. Если запрос достигает компонента маршрутизации, выбирается конечная точка, например страница Razor, для выполнения. Компонент конечной точки выполняет выбранную конечную точку

В традиционном описании паттерна проектирования MVC есть только один тип модели, который содержит все данные и поведение, не относящийся к пользовательскому интерфейсу. Контроллер обновляет эту модель соответствующим образом, а затем передает ее представлению, которое использует ее для создания пользовательского интерфейса.

Одна из проблем, возникающих при обсуждении MVC, – это расплывчатые и неоднозначные термины, которые он использует, такие как «контроллер» и «модель». «Модель», в частности, – настолько перегруженный термин, что часто бывает трудно понять, к чему именно он относится – это объект, коллекция объектов или абстрактное понятие? Даже в ASP.NET Core это слово используется для описания нескольких связанных, но разных компонентов, как вы вскоре увидите.

13.4.1 Направление запроса на страницу Razor и создание модели привязки

Первый шаг, когда ваше приложение получает запрос, – это его маршрутизация на соответствующий обработчик страницы Razor. Давайте снова обратимся к странице списка дел с категориями из листинга 13.5. На этой странице отображен список элементов с ярлыком заданной категории. Если вы просматриваете список элементов с категорией Simple, то должны сделать запрос к пути /category/Simple.

Маршрутизация берет заголовки и путь запроса, `/category/Simple`, и сопоставляет их с предварительно зарегистрированным списком паттернов. Каждый из них соответствует пути к одной странице Razor и обработчику страницы. Подробнее о маршрутизации вы узнаете в следующей главе.

НА ЗАМЕТКУ Я использую термин *Razor Page* для обозначения сочетания представления Razor и модели страницы, которая включает в себя обработчик страницы. Обратите внимание, что класс `PageModel` – *не* та «модель», о которой мы говорим при описании паттерна MVC. Как вы увидите позже в данном разделе, у него другая роль.

Когда в компоненте маршрутизации выбран обработчик страницы, запрос продолжает движение по конвейеру промежуточного ПО, пока не достигнет компонента конечной точки, где выполняется страница Razor.

Сначала создается *модель привязки* (если она применима). Эта модель строится из входящего запроса на основе свойств модели страницы, помеченных для привязки, и параметров метода, требуемых обработчиком страницы, как показано на рис. 13.9. Модель привязки обычно представляет собой один или несколько стандартных объектов C# и работает так же, как и в минимальных API, как вы видели в главе 6. Мы подробнее рассмотрим модели привязки Razor Page в главе 16.

1. Запрос получен и проходит через конвейер промежуточного ПО

Запрос

Получение URL-адреса
`/category/Simple`

2. Компонент маршрутизации направляет запрос конкретной странице Razor и обработчику

Компонент
маршрутизации

URL-адрес сопоставляется
со страницей Razor
`CategoryModel.OnGet`

3. Модель привязки строится из деталей, предоставленных в запросе

Модель привязки

`category = "Simple"`

4. Странице Razor передается модель привязки, и выполняется метод обработчика страницы

Обработчик
страницы

Выполняется обработчик
страницы
`OnGet(category)`

Страница Razor

Рис. 13.9 Маршрутизация запроса к контроллеру и построение модели привязки. Запрос URL-адреса `/category/Simple` приводит к выполнению обработчика страницы `CategoryModel.OnGet`, передавая заполненную модель привязки `category`

ОПРЕДЕЛЕНИЕ *Модель привязки* – это один или несколько объектов, которые действуют как контейнер для данных, предоставленных в запросе, – данных, которые требуются обработчику страницы.

В данном случае модель привязки представляет собой простую строку `category`, которой задается значение `"Simple"`. Это значение указывается в пути URL-адреса запроса. Также можно было использовать более сложную модель привязки, в которой несколько свойств были бы заполнены значениями из шаблона маршрута, строки запроса и тела запроса.

ПРИМЕЧАНИЕ Модель привязки для Razor Pages концептуально эквивалентна всем параметрам, передаваемым в конечную точку минимальных API, которые заполняются из запроса.

Модель привязки в данном случае соответствует параметру метода обработчика страницы `OnGet`. Экземпляр страницы Razor создается с помощью конструктора, а модель привязки передается обработчику страницы при ее выполнении, поэтому ее можно использовать для того, чтобы принять решение относительно того, как ответить. В этом примере обработчик страницы использует ее, чтобы решить, какие элементы списка отображать на странице.

13.4.2 Выполнение обработчика с использованием модели приложения

Роль обработчика страницы как контроллера в паттерне MVC заключается в координации генерации ответа на запрос, который он обрабатывает. Это означает, что он должен выполнять только ограниченное количество действий. В частности, он должен:

- убедиться, что данные, содержащиеся в предоставленной модели привязки, являются допустимыми для запроса;
- вызвать соответствующие действия в модели приложения с помощью сервисов;
- выбрать соответствующий ответ для генерации на основе ответа от модели приложения.

На рис. 13.10 показан обработчик страницы, вызывающий соответствующий метод в модели приложения. Здесь видно, что модель приложения – это несколько абстрактная концепция, которая инкапсулирует оставшиеся части приложения, не относящиеся к пользовательскому интерфейсу. Она содержит модель предметной области, ряд сервисов и взаимодействие с базой данных.

1. Обработчик страницы использует категорию, предоставляемую моделью привязки, чтобы определить, какой метод модели приложения вызвать

2. Обработчик страницы вызывает сервисы, которые составляют модель приложения. Также может использоваться модель предметной области, например чтобы определить, нужно ли включать завершенные элементы списка дел

3. Сервисы загружают подробную информацию о списке дел из базы данных и возвращают их в метод действия

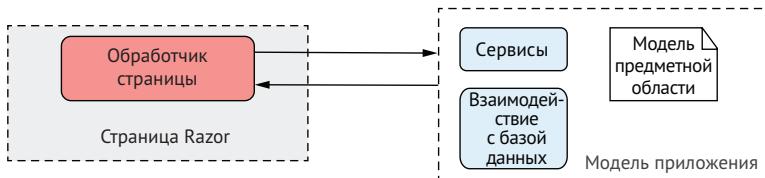


Рис. 13.10 При выполнении действие вызовет соответствующие методы в модели приложения

ОПРЕДЕЛЕНИЕ Модель предметной области инкапсулирует сложную бизнес-логику в набор классов, которые не зависят от какой-либо инфраструктуры и которые можно легко протестировать.

Обработчик страницы обычно вызывает одну точку в модели приложения. В нашем примере просмотра категории списка дел модель приложения может использовать различные сервисы, чтобы проверить, разрешено ли текущему пользователю просматривать определенные элементы, искать элементы в данной категории, загружать сведения из базы данных или картинку, связанную с элементом из файла. Если запрос проходит проверку, модель приложения вернет необходимые данные обработчику страницы. Затем обработчик должен выбрать ответ для генерации.

13.4.3 Генерация HTML-кода с использованием модели представления

После того как обработчик страницы вызвал модель приложения, содержащую бизнес-логику, пора сгенерировать ответ. *Модель представления* фиксирует детали, необходимые для того, чтобы представление сгенерировало ответ.

ОПРЕДЕЛЕНИЕ *Модель представления* в паттерне MVC – это все данные, необходимые представлению для отрисовки пользовательского интерфейса. Обычно это некое преобразование данных, содержащихся в модели приложения, плюс дополнительная информация, необходимая для отрисовки страницы, например ее заголовок.

Термин «модель представления» широко используется в ASP.NET Core MVC, где обычно обозначает отдельный объект, который передается в представление Razor для отрисовки. Однако в Razor Pages представление Razor может напрямую обращаться к классу модели страницы Razor Page. Следовательно, *PageModel* обычно действует как модель представления в Razor Pages, при этом доступ к данным, необходимым для представления Razor, осуществляется через свойства, как вы видели в листинге 13.5.

ПРИМЕЧАНИЕ Razor Pages использует сам класс *PageModel* в качестве модели представления для Razor, предоставляя необходимые данные как свойства.

Представление Razor использует данные, имеющиеся в модели страницы, для генерации окончательного ответа в виде HTML. В итоге ответ отправляется обратно по конвейеру промежуточного ПО и выводится в браузер пользователя, как показано на рис. 13.11.

Важно отметить, что хотя обработчик страницы выбирает, выполнять ли представление и какие данные использовать, он не контролирует, *какой HTML-код генерируется*. Само представление решает, каким будет содержание ответа.

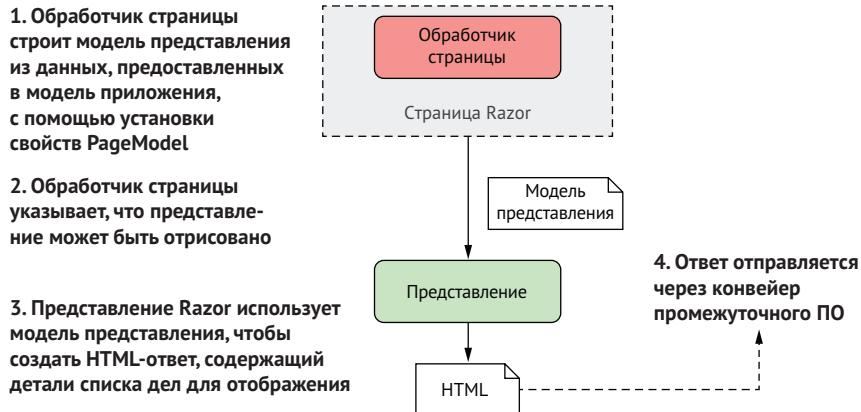


Рис. 13.11 Обработчик страницы создает модель представления, задавая свойства PageModel. Это представление, генерирующее ответ

13.4.4 Собираем все вместе: полный запрос страницы Razor

Теперь, когда вы ознакомились с каждым этапом обработки запроса в ASP.NET Core с помощью Razor Pages, объединим все это от запроса к ответу. На рис. 13.12 показано, как объединяются шаги для обработки запроса на отображение списка дел для категории Simple. Традиционный паттерн MVC все еще виден в Razor Pages. Он состоит из обработчика страницы (контроллера), представления и модели приложения.

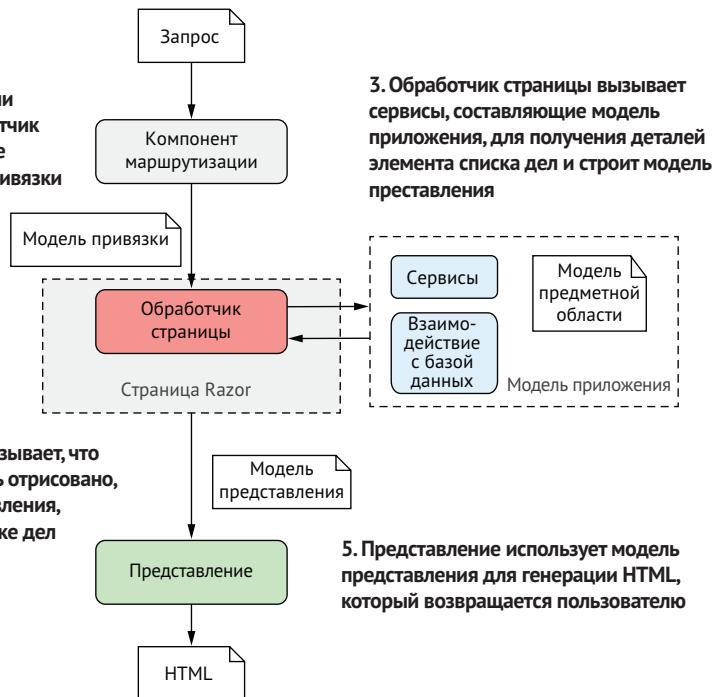
К настоящему времени вы можете подумать, что весь этот процесс кажется довольно запутанным – так много шагов, чтобы отобразить HTML! Почему бы не позволить модели приложения создавать представление напрямую, вместо того чтобы танцевать взад и вперед с методом обработчика страницы? Ключевым преимуществом такого подхода является *разделение ответственности*:

- представление отвечает только за получение некоторых данных и генерацию HTML-кода;
- модель приложения отвечает лишь за выполнение необходимой бизнес-логики;
- обработчик страницы (контроллер) отвечает только за проверку входящего запроса и выбор необходимого ответа на основе выходных данных модели приложения.

Благодаря четко определенным границам легче обновлять и тестировать каждый из компонентов независимо от других. Если логика пользовательского интерфейса изменится, вам не обязательно будет изменять какой-либо из классов бизнес-логики, поэтому вероятность возникновения ошибок в неожиданных местах снижается.

1. Получен запрос на URL
/category/Simple

2. Компонент маршрутизации направляет запрос в обработчик страницы OnGet на странице Category и строит модель привязки



4. Обработчик страницы указывает, что представление должно быть отрисовано, и передает модель представления, содержащую данные о списке дел

3. Обработчик страницы вызывает сервисы, составляющие модель приложения, для получения деталей элемента списка дел и строит модель представления

5. Представление использует модель представления для генерации HTML, который возвращается пользователю

Рис. 13.12 Полный запрос Razor Pages для списка дел в категории Simple

Опасность сильной связности

В целом рекомендуется как можно больше уменьшать связность между логическими частями приложения. Это упрощает его обновление, не вызывая неблагоприятных последствий и не требуя изменений в, казалось бы, несвязанных областях. Применение паттерна MVC – один из способов помочь вам в достижении этой цели.

В качестве примера, когда связность мешает, я вспоминаю случай, произошедший несколько лет назад, когда я работал над небольшим веб-приложением. В спешке мы не отделили должным образом бизнес-логику от кода генерации HTML-кода, но поначалу очевидных проблем не было – код работал, поэтому мы отправили его заказчику!

Несколько месяцев спустя какой-то новый сотрудник начал работать над приложением, и я сразу же «помог» ему, переименовав безобидную орфографическую ошибку в классе на бизнес-уровне. К сожалению, имена этих классов были использованы для генерации HTML-кода, поэтому переименование класса привело к выходу из строя всего сайта в браузерах пользователей! Достаточно сказать, что после этого мы приложили совместные усилия, чтобы применить паттерн MVC и убедиться, что у нас было надлежащее разделение ответственности.

Примеры, показанные в этой главе, демонстрируют основную часть функциональности Razor Pages. У него есть дополнительные функции, такие как конвейер фильтров, о котором я расскажу в главах 21 и 22, и более подробно остановлюсь на моделях связывания в главе 16, но общее поведение системы останется без изменений.

Аналогичным образом в главе 19 я рассматриваю контроллеры MVC и объясняю, почему я не рекомендую использовать их вместо Razor Pages для приложений с отрисовкой на стороне сервера. В главе 20 я обсуждаю, как использовать паттерн проектирования MVC при генерации машиночитаемых ответов с помощью контроллеров веб-API. Этот процесс по сути идентичен паттерну MVC, который вы уже видели.

Я надеюсь, что к этому моменту вы уже увлеклись Razor Pages и принципом его работы с использованием паттерна MVC. Методы обработчика страницы на странице Razor вызываются в ответ на запрос и выбирают тип ответа для генерации, возвращая `IActionResult`.

Аспект, который я затронул лишь вскользь, – это то, как компонент `RoutingMiddleware` решает, какую страницу Razor и какой обработчик вызывать для данного запроса. Вам не нужна страница Razor для каждого URL-адреса в приложении. Например, было бы сложно иметь отдельную страницу для каждого товара в интернет-магазине; каждому товару понадобилась бы своя страница Razor! В главе 14 вы увидите, как определять маршруты для страниц Razor, добавлять ограничения к маршрутам и как они деконструируют URL-адреса, чтобы соответствовать отдельному обработчику.

Резюме

- Страницы Razor расположены в папке проекта Pages и по умолчанию называются в соответствии с URL-путем, который они обрабатывают. Например, `Privacy.cshtml` обрабатывает путь `/Privacy`. Это соглашение позволяет легко и быстро добавлять новые страницы;
- страницы Razor должны содержать директиву `@page` в первой строке файла `.cshtml`. Без этой директивы ASP.NET Core не распознает ее как страницу Razor, и она не будет отображаться как конечная точка в приложении;
- модели страниц являются производными от базового класса `RageModel` и содержат обработчики страниц. Обработчики страниц – это методы, названные с использованием соглашений, указывающих HTTP-метод, который они обрабатывают. Например, `OnGet` обрабатывает команду `GET`. Обработчики страниц эквивалентны обработчикам конечных точек минимальных API; они запускаются в ответ на данный запрос;
- шаблоны Razor могут содержать автономный код C#, автономный HTML-код и динамический HTML-код, созданный на основе значений C#. Объединив их все, можно создавать высокодинамичные приложения;

- паттерн проектирования MVC позволяет разделить задачи между бизнес-логикой приложения, передаваемыми данными и отображением данных в ответе. Это уменьшает связность между различными уровнями приложения;
- Razor Pages должен наследовать от базового класса `PageModel` и содержать обработчики страниц. Компонент маршрутизации выбирает обработчик страницы на основе входящего URL-адреса запроса, HTTP-метода и строки запроса;
- обработчики страниц обычно должны делегировать полномочия сервисам для обработки бизнес-логики, необходимой запросу, вместо того чтобы выполнять изменения самостоятельно. Это обеспечивает четкое разделение ответственности, что помогает при тестировании и улучшает структуру приложения.

14

Сопоставление URL-адресов с Razor Pages с использованием маршрутизации

В этой главе:

- маршрутизация запросов в Razor Pages;
- настройка шаблонов маршрутов Razor Page;
- генерация URL-адресов для страниц Razor.

В предыдущей главе вы узнали о паттерне проектирования MVC и о том, как ASP.NET Core использует его для создания пользовательского интерфейса приложения с помощью Razor Pages. Razor Pages содержат обработчики страниц, которые действуют как мини-контроллеры для запроса. Обработчик страницы вызывает модель приложения для получения или сохранения данных. Затем он передает данные из модели приложения в представление Razor, которое генерирует ответ в виде HTML-страницы.

Хотя это и не является частью паттерна проектирования MVC как такового, одна из важнейших частей Razor Pages – это выбор страницы Razor, вызываемой в ответ на данный запрос. Razor Pages использу-

зует ту же систему маршрутизации, что и минимальные API (представленные в главе 6). В этой главе основное внимание уделяется тому, как маршрутизация работает с Razor Pages.

Я начинаю эту главу с краткого напоминания о том, как работает маршрутизация в ASP.NET Core. Я коснусь двух компонентов промежуточного ПО, которые имеют решающее значение для маршрутизации конечных точек в .NET 7, и подхода, который использует Razor Pages для смешивания соглашений с явными шаблонами маршрутов.

В разделе 14.3 мы рассмотрим поведение маршрутизации Razor Pages по умолчанию, а в разделе 14.4 вы узнаете, как настроить поведение, добавляя или изменяя шаблоны маршрутов.

Razor Pages имеет доступ к тем же функциям шаблона маршрута, о которых вы узнали в главе 6, а в разделе 14.4 вы узнаете, как ими воспользоваться.

В разделе 14.5 я описываю, как использовать систему маршрутизации для создания URL-адресов для Razor Pages. Razor Pages предоставляет несколько вспомогательных методов, упрощающих создание URL-адресов по сравнению с минимальными API, поэтому я сравниваю оба подхода и обсуждаю преимущества каждого.

Наконец, в разделе 14.6 я описываю, как настроить соглашения, используемые Razor Pages, что дает вам полный контроль над URL-адресами в приложении. Вы увидите, как изменить встроенные соглашения, например использовать строчные буквы для URL-адресов, а также как написать собственное соглашение и применить его глобально к своему приложению.

К концу этой главы вы получите гораздо более четкое представление о том, как работает приложение ASP.NET Core. Маршрутизацию можно рассматривать как связующее звено, соединяющее конвейер промежуточного ПО с Razor Pages и фреймворком MVC. Используя промежуточное ПО, Razor Pages и маршрутизацию, вы сможете писать веб-приложения в кратчайшие сроки!

14.1 Маршрутизация в ASP.NET Core

В главе 6 мы подробно рассмотрели маршрутизацию и некоторые ее преимущества, например возможность иметь несколько URL-адресов, указывающих на одну и ту же конечную точку, и извлечение сегментов из URL-адреса. Вы также узнали, как это реализовано в приложениях ASP.NET Core с использованием двух компонентов:

- `EndpointMiddleware` – этот компонент используется для регистрации конечных точек в системе маршрутизации при запуске приложения. Он выполняет одну из конечных точек во время выполнения;
- `RoutingMiddleware` – этот компонент выбирает, какая из конечных точек, зарегистрированных `EndpointMiddleware`, должна выполниться для данного запроса во время выполнения.

`EndpointMiddleware` – то место, где вы регистрируете все конечные точки приложения, включая минимальные API, страницы Razor,

и контроллеры MVC. Все страницы Razor в приложении можно легко зарегистрировать с помощью метода расширения `MapRazorPages()`, как показано в следующем листинге.

Листинг 14.1. Регистрация страниц Razor в Startup.Configure

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages(); ← Добавляет необходимые сервисы
var app = builder.Build();
app.UseStaticFiles();
app.UseRouting(); ← Добавляет RoutingMiddleware
app.UseAuthorization(); ← в конвейер промежуточного ПО
app.MapRazorPages(); ← Регистрирует все страницы Razor в при-
app.Run(); ← ложении в EndpointMiddleware
```

Каждая конечная точка, будь то страница Razor или конечная точка минимального API, имеет связанный *шаблон маршрута*, определяющий, каким URL-адресам должна соответствовать конечная точка. `EndpointMiddleware` сохраняет эти шаблоны маршрутов и конечные точки в словаре, который используется совместно с `RoutingMiddleware`. Во время выполнения `RoutingMiddleware` сравнивает входящий запрос с маршрутами в словаре и выбирает соответствующую конечную точку. Когда запрос достигает `EndpointMiddleware`, промежуточное ПО проверяет, какая конечная точка была выбрана, и выполняет ее, как показано на рис. 14.1.

Как обсуждалось в главе 6, преимущество наличия двух отдельных компонентов для обработки этого процесса состоит в том, что любой компонент, размещенный после `RoutingMiddleware`, может видеть, какая конечная точка будет выполнена, до того, как это произойдет. Вы увидите это преимущество в действии, когда мы будем рассматривать авторизацию в главе 24.

Маршрутизация в ASP.NET Core использует одну и ту же инфраструктуру и промежуточное ПО независимо от того, создаете ли вы минимальные API, страницы Razor или контроллеры MVC, но существуют некоторые различия в том, как вы определяете сопоставление между шаблонами маршрутов и обработчиками в каждом случае. В разделе 14.2 вы узнаете о различных подходах, которые использует каждая парадигма.

14.2 Маршрутизация на основе соглашений и явная маршрутизация

Маршрутизация является ключевой частью ASP.NET Core, поскольку она сопоставляет URL-адрес входящего запроса с конкретной конечной точкой, которую нужно будет выполнить. Существует два способа определить эти сопоставления URL-адресов конечным точкам в приложении:

- использовать глобальную маршрутизацию на основе соглашений;
- использовать явную маршрутизацию, где каждая конечная точка сопоставляется с отдельным шаблоном маршрута.

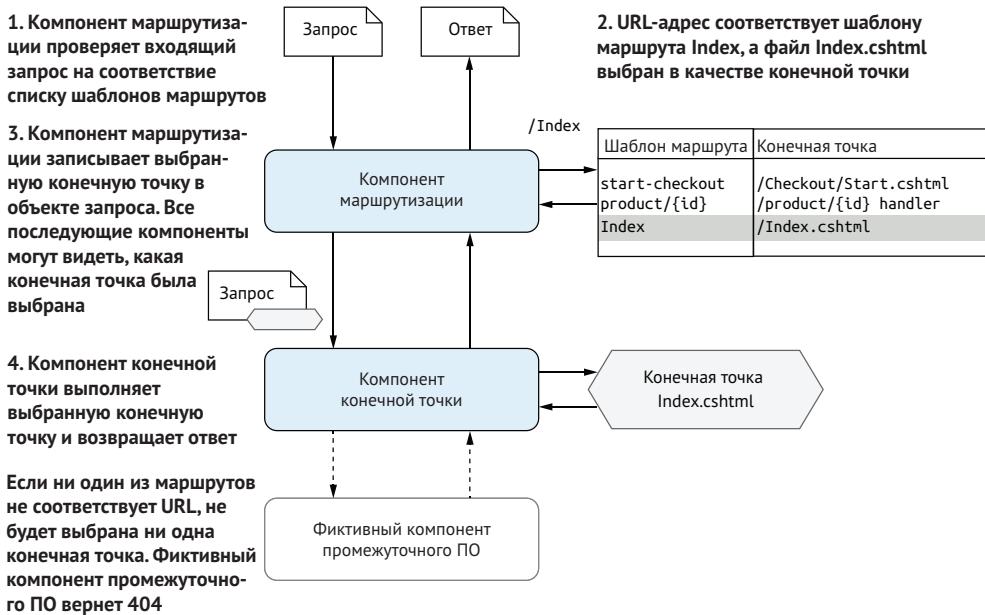


Рис. 14.1 Маршрутизация конечных точек использует двухэтапный процесс. RoutingMiddleware выбирает конечную точку для выполнения, а EndpointMiddleware выполняет ее. Если URL-адрес запроса не соответствует шаблону маршрута, промежуточное ПО конечных точек не сгенерирует ответ

Какой подход вы будете применять, обычно зависит от того, используете ли вы минимальные API, страницы Razor или контроллеры MVC и создаете ли вы API или веб-сайт (с помощью HTML-кода). Как вы вскоре увидите, я склоняюсь к использованию явной маршрутизации.

Маршрутизация на основе соглашений определяется глобально для вашего приложения. Вы можете использовать основанные на соглашениях маршруты для сопоставления конечных точек (конкретнее, методов действия контроллеров MVC) с URL-адресами, но эти контроллеры должны строго придерживаться определенных вами соглашений. Традиционно приложения, использующие контроллеры MVC для генерации HTML-кода, склонны применять этот подход. Но у него есть и обратная сторона. Она состоит в том, что это затрудняет настройку URL-адресов для подмножества контроллеров и действий.

В качестве альтернативы можно использовать явную маршрутизацию для привязки заданного URL-адреса к определенной конечной точке. Вы видели этот подход в минимальных API, где каждая конечная точка напрямую связана с шаблоном маршрута. В случае с контроллерами MVC это включает в себя размещение атрибутов [Route] на самих методах действия, поэтому явную маршрутизацию часто называют *маршрутизацией на основе атрибутов*. Это обеспечивает большую гибкость, поскольку вы можете явно определить, каким должен быть URL-адрес для каждого метода действия. Данный подход, как правило, более подробный, по сравнению с тем, где используется маршрутизация на основе соглашений,

поскольку требует применения атрибутов к каждому методу действия в приложении. Несмотря на это, дополнительная гибкость, предоставляемая им, может быть очень полезной, особенно при создании API.

Несколько сбивает с толку то, что страницы Razor используют *соглашения для создания явных маршрутов!* Во многих отношениях такое сочетание привносит лучшее из обоих миров – вы получаете предсказуемость и краткость маршрутизации на основе соглашений с простой настройкой явной маршрутизации. Как показано в табл. 14.1, для каждого из этих подходов есть свои компромиссы.

Итак, какой же подход следует использовать? Я считаю, что маршрутизация на основе соглашений не стоит усилий в 99 % случаев и следует придерживаться явной маршрутизации. Если вы следете моему совету по использованию Razor Pages для приложений с отрисовкой на стороне сервера, то, значит, вы уже используете явную маршрутизацию. Кроме того, если вы создаете API с помощью минимальных API или контроллеров MVC, данный вид маршрутизации – лучший вариант и рекомендуемый подход.

Таблица 14.1 Преимущества и недостатки стилей маршрутизации, доступных в ASP.NET Core

Вид маршрутизации	Типичное использование	Преимущества	Недостатки
Маршрутизация на основе соглашений	Контроллеры MVC, генерирующие HTML-код	Очень краткое определение в одном месте приложения. Обеспечивает согласованную компоновку контроллеров MVC	Маршруты определяются вне контроллеров. Переопределение соглашений о маршрутах может быть сложным и подвержено ошибкам. Добавляет дополнительный уровень косвенности при маршрутизации запроса
Явная маршрутизация	Конечные точки минимальных API Контроллеры MVC веб-API	Предоставляет полный контроль над шаблонами маршрутов для каждой конечной точки. Маршруты определяются рядом с конечной точкой, которой они соответствуют	Многословно по сравнению с маршрутизацией на основе соглашений. Шаблоны маршрутов легко изменить. Шаблоны маршрутов разбросаны по всему приложению, а не собраны в одном месте
Генерация явных маршрутов на основе соглашений	Razor Pages	Поощряет согласованный набор предоставляемых URL-адресов. Лаконично, когда вы придергиваетесь соглашений. Легко переопределить шаблон маршрута для отдельной страницы. Настройте соглашения глобально, чтобы изменить предоставляемые URL-адреса	Возможна чрезмерная настройка шаблонов маршрутов. Вы должны рассчитать, каким должен быть шаблон маршрута для страницы, вместо того чтобы явно определять его в приложении

Единственный сценарий, в котором традиционно применяется маршрутизация на основе соглашений, – это использование контроллеров MVC для генерации HTML-кода. Но если вы последуете моему совету из главы 13, то будете использовать Razor Pages для приложений, генерирующих HTML-код, и возвращаться к контроллерам MVC только тогда, когда это необходимо. Этот вопрос более подробно обсуждается в главе 19. Для единообразия в данном сценарии я бы по-прежнему придерживался явной маршрутизации.

ПРИМЕЧАНИЕ По указанным выше причинам в этой книге основное внимание уделяется явной маршрутизации или маршрутизации на основе атрибутов. Дополнительные сведения о маршрутизации на основе соглашений см. в документации Microsoft «Маршрутизация для действий контроллера в ASP.NET Core»: <http://mng.bz/ZP00>.

Мы познакомились с маршрутизацией и шаблонами маршрутов в главе 6 в контексте минимальных API. Хорошая новость состоит в том, что в Razor Pages доступны точно такие же шаблоны и функции. Основное отличие от минимальных API заключается в том, что Razor Pages используют соглашения для создания шаблона маршрута для страницы, хотя вы можете легко изменить шаблон для каждой страницы. В разделе 14.3 мы подробно рассмотрим соглашения по умолчанию и то, как маршрутизация сопоставляет URL-адрес запроса со страницей Razor.

14.3 Маршрутизация запросов в Razor Pages

Как я упоминал в разделе 14.2, Razor Pages использует явную маршрутизацию, создавая шаблоны маршрутов на основе соглашений. ASP.NET Core создает шаблон маршрута для каждой страницы Razor в приложении во время его запуска, когда вы вызываете метод `MapRazorPages()` в файле `Program.cs`:

```
app.MapRazorPages();
```

Для каждой страницы Razor в приложении используется путь к файлу страницы относительно корневого каталога страниц `Pages/`, за исключением расширения файла (`.cshtml`). Например, если у вас есть страница, расположенная по пути `Pages/Products/View.cshtml`, фреймворк создает шаблон маршрута со значением `"Products/View"`, как показано на рис. 14.2.

Запросы к URL-адресу `/products/view` соответствуют шаблону маршрута `"Products/View"`, который, в свою очередь, соответствует странице `View.cshtml` из папки `Pages/Products`. `RoutingMiddleware` выбирает страницу `View.cshtml` в качестве конечной точки для запроса, а `EndpointMiddleware` выполняет обработчик страницы, как только запрос доходит до него в конвейере.

ПРИМЕЧАНИЕ Помните, что маршрутизация не чувствительна к регистру, поэтому URL-адрес запроса будет совпадать, даже если его регистр отличается от шаблона маршрута.

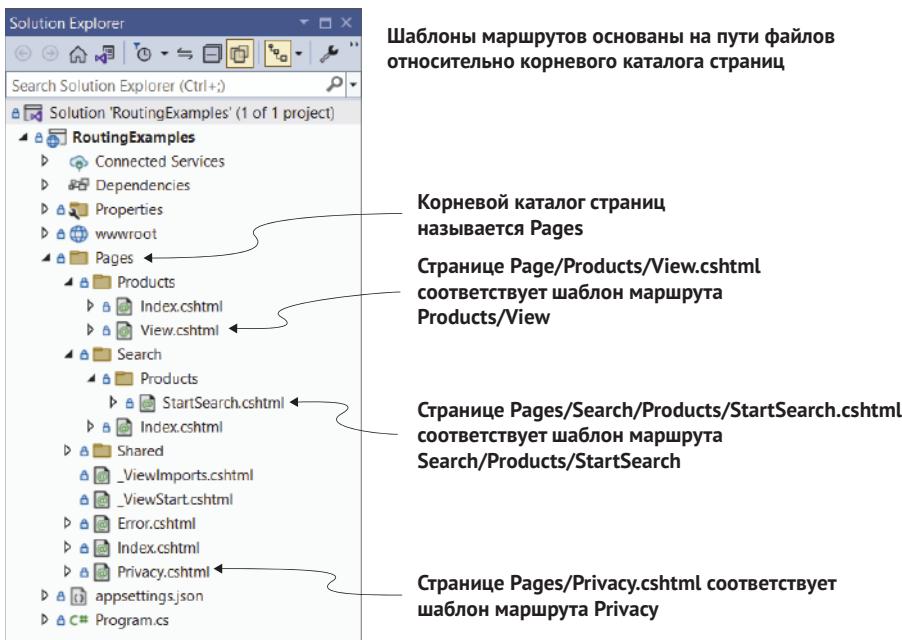


Рис. 14.2 По умолчанию шаблоны маршрутов создаются для страниц Razor на основе пути к файлу относительно корневого каталога, Pages

В главе 13 вы узнали, что обработчики страниц Razor – это методы, которые вызываются на странице Razor, например `OnGet`. Когда мы говорим «страница Razor выполняется», то на самом деле имеем в виду, что «создается экземпляр `PageModel` страницы Razor и вызывается обработчик модели страницы». У страниц Razor может быть несколько обработчиков, поэтому после того, как `RoutingMiddleware` выберет страницу, `EndpointMiddleware` нужно решить, какой обработчик выполнить. Вы узнаете, как фреймворк выбирает обработчик страницы для вызова, в главе 15.

По умолчанию каждая страница Razor создает единый шаблон маршрута на основе пути к файлу. Исключением из этого правила являются страницы `Index.cshtml`. Они создают два шаблона маршрутов, один из которых оканчивается на `"Index"`, а другой не имеет окончания. Например, если у вас есть страница Razor по пути `Pages/ToDo/Index.cshtml`, будет сгенерировано два шаблона маршрута:

- "ToDo";
- "ToDo/Index".

При совпадении любого из этих маршрутов выбирается одна и та же страница `Index.cshtml`. Например, если ваше приложение работает по URL-адресу <https://example.org>, можно просмотреть страницу, выполнив <https://example.org/ToDo> или <https://example.org/ToDo/Index>.

ВНИМАНИЕ! При использовании страниц `Index.cshtml` необходимо следить за перекрытием маршрутов. Например, если вы добавите `Pages/ToDo/Index.cshtml` в приведенном выше при-

мере, не следует добавлять страницу Pages/ToDo.cshtml, так как во время выполнения вы получите исключение при переходе к /todo, как вы увидите в разделе 14.6.

В качестве последнего примера рассмотрим страницы Razor, созданные по умолчанию, при создании приложения с помощью Visual Studio или при выполнении команды `dotnet new web` с помощью интерфейса командной строки .NET, как мы это делали в главе 13. Стандартный шаблон включает в себя три страницы в каталоге Pages:

- Pages/Error.cshtml;
- Pages/Index.cshtml;
- Pages/Privacy.cshtml.

Таким образом создается набор из четырех маршрутов для приложения, определенных следующими шаблонами:

- "" соответствует Index.cshtml;
- "Index" соответствует Index.cshtml;
- "Error" соответствует Error.cshtml;
- "Privacy" соответствует Privacy.cshtml.

На данном этапе маршрутизация, вероятно, кажется до смешного банальной, но это только основы, которые вы получаете бесплатно, используя соглашения Razor Pages по умолчанию. Нередко этого бывает достаточно для большей части веб-сайтов. Однако в какой-то момент вы обнаружите, что вам необходимо что-то более динамичное, например использование параметров маршрута для включения идентификатора в URL-адрес. Именно здесь становится полезной возможность настраивать шаблоны маршрутов страницы Razor.

14.4 Настройка шаблонов маршрутов страницы Razor

Шаблоны маршрута для страницы Razor по умолчанию основаны на пути к файлу, но вы также можете настроить окончательный шаблон для каждой страницы или даже полностью заменить его. В этом разделе я покажу, как настроить шаблоны маршрутов для отдельных страниц, чтобы вы могли настроить URL-адреса своего приложения и сопоставить несколько URL-адресов с одной страницей.

Возможно, вы помните из главы 6, что шаблоны маршрутов состоят из литеральных сегментов и из параметров маршрута, как показано на рис. 14.3. По умолчанию на страницах Razor есть URL-адреса, состоящие из ряда литеральных сегментов, например "ToDo/Index".

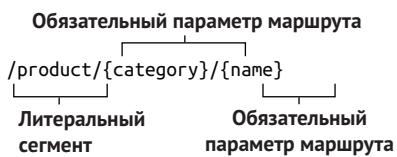


Рис. 14.3 Простой шаблон маршрута с литеральным сегментом и двумя обязательными параметрами маршрута

Литеральные сегменты и параметры маршрута – это два краеугольных камня шаблонов маршрутов ASP.NET Core, но как настроить страницу Razor на использование одного из этих шаблонов? В разделе 14.4.1 вы увидите, как добавить сегмент в конец шаблона маршрута страницы Razor, а в разделе 14.4.2 – как полностью заменить шаблон маршрута.

14.4.1 Добавление сегмента в шаблон маршрута

Чтобы настроить шаблон маршрута страницы Razor, обновите директиву `@page` в верхней части файла `.cshtml`. Эта директива – первое, что должно быть указано в файле для правильной регистрации страницы Razor. Чтобы добавить дополнительный сегмент в шаблон маршрута страницы Razor, добавьте пробел, за которым следует нужный шаблон маршрута, после инструкции `@page`. Например, чтобы добавить "Extra" к шаблону маршрута страницы Razor, используйте

```
@page "Extra"
```

Так предоставленный шаблон маршрута будет добавлен к шаблону по умолчанию, созданному для страницы Razor. Например, шаблон маршрута по умолчанию для страницы Razor в `Pages/Privacy.html` – это `"Privacy"`. С помощью предыдущей директивы новый шаблон маршрута для страницы будет выглядеть так: `"Privacy/Extra"`.

Наиболее частой причиной такой настройки шаблона маршрута страницы Razor является добавление параметра маршрута. Например, у вас есть одна страница Razor для отображения продуктов на сайте для онлайн-торговли по пути `Pages/Products.cshtml` и вы можете использовать параметр маршрута в директиве `@page`:

```
@page "{category}/{name}"
```

Так вы получаете окончательный шаблон маршрута `Products/{category}/{name}`, который будет соответствовать всем перечисленным ниже URL-адресам:

- `/products/bags/white-rucksack;`
- `/products/shoes/black-size9;`
- `/Products/phones/iPhoneX.`

ПРИМЕЧАНИЕ Можно использовать те же функции маршрутизации, о которых вы узнали в главе 6 с Razor Pages, включая необязательные параметры, параметры по умолчанию и ограничения.

Очень часто сегменты маршрута добавляются в шаблон страницы Razor таким образом, но что, если этого недостаточно? Возможно, вы не хотите, чтобы сегмент `/products` присутствовал в начале предыдущих URL-адресов, или вам нужно использовать полностью настраиваемый URL-адрес страницы. К счастью, этого легко добиться.

14.4.2 Полная замена шаблона маршрута

Работа со страницами Razor будет наиболее продуктивной, если вы сможете по возможности придерживаться соглашений о маршрутизации по умолчанию, добавляя при необходимости дополнительные сегменты для параметров маршрута. Но иногда вам просто нужно больше контроля. Это часто происходит с важными страницами приложения, например страницей оформления заказа, или даже страницами продуктов, как вы видели в предыдущем разделе.

Чтобы указать настраиваемый маршрут для страницы Razor, используйте префикс `/` в директиве `@page`. Например, чтобы удалить префикс `"product/"` из шаблонов маршрутов в предыдущем разделе, используйте эту директиву:

```
@page "/{category}/{name}"
```

Обратите внимание, что данная директива содержит символ `/` в начале маршрута, указывая на то, что это *настраиваемый* шаблон маршрута, а не *добавление*. Шаблон маршрута для этой страницы будет выглядеть так: `"{category}/{name}"`, независимо от того, к какой странице Razor он применяется. Точно так же можно создать статический настраиваемый шаблон для страницы, поставив в начало шаблона символ `/` и используя только лiteralные сегменты. Например:

```
@page "/checkout"
```

Где бы вы ни разместили свою страницу оформления заказа в папке Pages, использование этой директивы гарантирует, что она всегда будет иметь шаблон маршрута `"checkout"`, поэтому всегда будет соответствовать URL-адресу запроса `/checkout`.

СОВЕТ Также можно рассматривать настраиваемые шаблоны маршрутов, которые начинаются с символа `/` как *абсолютные* шаблоны маршрутов, в то время как другие шаблоны являются *относительными* к их местоположению в файловой иерархии.

Важно отметить, что при настройке шаблона маршрута для страницы Razor как при добавлении к значению по умолчанию, так и при его замене настраиваемым маршрутом *шаблон по умолчанию становится недействительным*. Например, если вы используете шаблон маршрута `"checkout"` по адресу `Pages/Payment.cshtml`, то можете получить к нему доступ, только используя URL-адрес `/checkout`; URL-адрес `/Payment` уже будет недействителен и не будет выполнять страницу Razor.

СОВЕТ Настройка шаблона маршрута для страницы Razor с помощью директивы `@page` заменяет шаблон маршрута по умолчанию. В разделе 14.6 я покажу, как добавить дополнительные маршруты, сохранив шаблон маршрута по умолчанию.

В этом разделе вы узнали, как настроить шаблон маршрута для страницы Razor. По большей части маршрутизация на страницы Razor ра-

ботает как в минимальных API – основное отличие состоит в том, что шаблоны маршрутов создаются с использованием соглашений. Когда дело доходит до другой половины маршрутизации – генерации URL-адресов, – Razor Pages и минимальные API также схожи, но Razor Pages предоставляет ряд отличных помощников.

14.5 Генерация URL-адресов для страниц Razor

В этом разделе вы узнаете, как создавать URL-адреса для страниц Razor с помощью `IUrlHelper`, который является частью типа `PageModel`. Вы также научитесь использовать класс `LinkGenerator`, который вы видели в главе 6, для генерации URL-адресов в минимальных API.

Одним из преимуществ использования маршрутизации на основе соглашений в Razor Pages является тот факт, что ваши URL-адреса могут быть гибкими. Если вы переименуете страницу Razor, то URL-адрес, связанный с этой страницей, также изменится. Например, переименование страницы `Pages/Cart.cshtml` в `Pages/Basket/View.cshtml` приводит к изменению URL-адреса, который вы используете для доступа к странице, и вместо `/Cart` вы получаете `/Basket/View`.

Чтобы отслеживать эти изменения (и избегать неработающих ссылок), можно использовать инфраструктуру маршрутизации для создания URL-адресов, которые выводятся в HTML-код страницы Razor и включаются в HTTP-ответы. В главе 6 вы видели, как генерировать URL-адреса для конечных точек минимальных API, а в этом разделе вы увидите, как сделать то же самое для страниц Razor. Я также описываю, как генерировать URL-адреса для контроллеров MVC, поскольку этот механизм практически идентичен тому, что используется в Razor Pages.

14.5.1 Генерация URL-адресов для страницы Razor

Вам нужно будет генерировать URL-адреса в различных местах приложения, и одно из распространенных мест – это страницы Razor и контроллеры MVC. В следующем листинге показано, как создать ссылку на страницу `Pages/Currency/View.cshtml`, используя экземпляр `IUrlHelper` из базового класса `PageModel`.

Листинг 14.2. Генерация URL-адреса с использованием `IUrlHelper` и имени страницы Razor

```
public class IndexModel : PageModel {  
    public void OnGet()  
    {  
        var url = Url.Page("Currency/View", new { code = "USD" });  
    }  
}
```

Наследование от `PageModel`
дает доступ к свойству `Url`

Вы указываете относительный путь к странице Razor
вместе со всеми дополнительными значениями маршрута

Свойство `Url` – это экземпляр `IUrlHelper`, который позволяет легко создавать URL-адреса для приложения, ссылаясь на другие страницы Razor по пути к их файлу.

ПРИМЕЧАНИЕ IUrlHelper – это обертка класса LinkGenerator, о котором вы узнали в главе 6. IUrlHelper добавляет несколько ярлыков для генерации URL-адресов на основе текущего запроса.

IUrlHelper предоставляет метод `Page()`, которому вы передаете имя страницы Razor и любые дополнительные данные маршрута в качестве анонимного объекта.

СОВЕТ Можно указать *относительный* путь к файлу для страницы Razor, как показано в листинге 14.2. В качестве альтернативы вы можете указать *абсолютный* путь к файлу (относительно папки Pages), начав путь с символа "/", например `"/Currency/View"`.

IUrlHelper имеет несколько различных перегруженных вариантов метода `Page`. Некоторые из этих методов позволяют указать конкретный обработчик страницы, другие – сгенерировать абсолютный URL-адрес вместо относительного, а иные позволяют передавать дополнительные значения маршрута.

В листинге 14.2, помимо указания пути к файлу, я передал анонимный объект `new {code = "USD"}`. Этот объект предоставляет дополнительные значения маршрута при генерации URL-адреса, в данном случае задавая для параметра `code` значение `"USD"`, как вы это делали при создании URL-адресов для минимальных API с помощью `LinkGenerator` в главе 6. Как и раньше, значение кода используется непосредственно в URL-адресе, если оно соответствует параметру маршрута. В противном случае он добавляется в качестве дополнительных данных в строку запроса.

Генерировать URL-адреса на основе страницы, которую вы хотите выполнить, удобно, и в большинстве случаев это обычный подход. Если вы используете контроллеры MVC для своих API, процесс во многом такой же, как и для страниц Razor, хотя методы немного отличаются.

14.5.2 Генерация URL-адресов для контроллера MVC

Создание URL-адресов для контроллеров MVC очень похоже на то, как это делается для страниц Razor. Основное отличие состоит в том, что вы используете метод `Action` в `IUrlHelper` и предоставляете имя контроллера MVC и действия вместо пути к странице.

ПРИМЕЧАНИЕ Я рассмотрел контроллеры MVC лишь вскользь, поскольку обычно не рекомендую использовать их вместо Razor Pages или минимальных API, поэтому не беспокойтесь о них слишком сильно. Мы вернемся к контроллерам MVC в главах 19 и 20; основная причина их упоминания здесь – указать, насколько контроллеры MVC похожи на Razor Pages.

В следующем листинге показан контроллер MVC, генерирующий ссылку от одного метода действия к другому с помощью `IUrlHelper` из базового класса `Controller`.

Листинг 14.3 Создание URL-адреса с использованием IUrlHelper и имени метода действия

```

public class CurrencyController : Controller
{
    [HttpGet("currency/index")]
    public IActionResult Index()
    {
        var url = Url.Action("View", "Currency",
            new { code = "USD" });
        return Content($"The URL is {url}");
    }

    [HttpGet("currency/view/{code}")]
    public IActionResult View(string code)
    {
        /* Реализация метода */
    }
}

```

Сгенерированный URL-адрес ведет к этому методу действия

Наследование от класса Controller дает доступ к свойству Url

Явные шаблоны маршрутов с использованием атрибутов

Предоставляем имя действия и контроллера для создания URL, а также все дополнительные значения маршрута

Возвращает строку "The URL is /Currency/View/USD"

Можно вызывать методы Action и Page в IUrlHelper как из Razor Pages, так и из контроллеров MVC, чтобы при необходимости создавать ссылки между ними. Важный вопрос: каков пункт назначения URL-адреса? Если нужный вам URL-адрес относится к странице Razor, используйте метод Page(). Если речь идет о действии MVC, применяйте метод Action().

СОВЕТ Вместо строк для имени метода действия используйте оператор C# 6 nameof, чтобы сделать значение безопасным для рефакторинга, например nameof(View).

Если вы выполняете маршрутизацию к действию в том же контроллере, то можете использовать другой перегруженный вариант метода Action(), который опускает имя контроллера при генерации URL-адреса. IUrlHelper использует *внешние значения* из текущего запроса и заменяет их конкретными значениями, которые вы предоставляетете.

ОПРЕДЕЛЕНИЕ *Внешние значения* – это значения маршрута для текущего запроса. Они включают в себя Controller и Action при вызове из контроллера MVC и Page при вызове из страницы Razor. Внешние значения также могут включать дополнительные значения маршрута, которые были заданы при первоначальном обнаружении действия или страницы Razor с помощью маршрутизации. См. статью «Маршрутизация в ASP.NET Core» для получения дополнительных сведений: <http://mng.bz/OxoE>.

IUrlHelper может упростить создание URL-адресов за счет повторного использования внешних значений из текущего запроса, но это также добавляет сложности, поскольку одни и те же аргументы метода

могут давать разные сгенерированные URL-адреса в зависимости от страницы, из которой вызывается метод.

Если вам нужно создать URL-адреса из частей приложения за пределами страницы Razor или инфраструктуры MVC, вы не сможете использовать `IUrlHelper`. Вместо этого можно использовать класс `LinkGenerator`.

14.5.3 Генерация URL-адресов с помощью класса `LinkGenerator`

В главе 6 я рассказал, как генерировать ссылки на конечные точки минимальных API с помощью класса `LinkGenerator`. В отличие от `IUrlHelper`, `LinkGenerator` требует, чтобы вы всегда предоставляли достаточные аргументы для однозначного определения создаваемого URL-адреса. Это делает его более многословным и в то же время более последовательным, а также предоставляет преимущество, которое состоит в том, что его можно использовать в любом месте приложения. Данный класс отличается от `IUrlHelper`, который следует использовать только в контексте запроса.

Если вы пишете страницы Razor и контроллеры MVC, следуя советам из главы 13, постарайтесь делать свои страницы относительно простыми. Для этого требуется, чтобы вы выполняли бизнес-логику приложения и логику предметной области в отдельных классах и сервисах.

По большей части URL-адреса, используемые приложением, не должны быть частью логики предметной области. Это упрощает развитие приложения с течением времени или даже полное его изменение. Например, можно создать мобильное приложение, которое повторно использует бизнес-логику из приложения ASP.NET Core. В этом случае применение URL-адресов в бизнес-логике не имеет смысла, поскольку они не будут правильными, когда логика вызывается из мобильного приложения!

СОВЕТ По возможности старайтесь держать клиентскую часть приложения отдельно от бизнес-логики. Этот шаблон обычно известен как принцип инверсии зависимостей.

К сожалению, иногда такое разделение невозможно или оно значительно усложняет ситуацию. Один из примеров – когда вы создаете электронные письма в фоновом сервисе: вероятно, вам нужно будет включить ссылку на свое приложение в электронное письмо. Класс `LinkGenerator` позволяет сгенерировать этот URL-адрес, чтобы он автоматически обновлялся при изменении маршрутов в приложении.

Как показано в главе 6, класс `LinkGenerator` доступен в любой части приложения, поэтому можно использовать его внутри промежуточного ПО, конечных точек минимальных API и любых других сервисов. Вы также можете использовать его из Razor Pages и MVC, если хотите, но `IUrlHelper` обычно проще и скрывает некоторые детали применения `LinkGenerator`.

Вы уже видели, как генерировать ссылки на конечные точки минимальных API с помощью `LinkGenerator`, используя такие методы, как `GetPathByName()` и `GetUriByName()`. `LinkGenerator` имеет различные аналогичные методы для создания URL-адресов для Razor Pages и действий MVC, например `GetPathByPage()`, `GetPathByAction()` и `GetUriByPage()`, как показано в следующем листинге.

Листинг 14.4. Генерация URL-адресов с помощью класса LinkGenerator

```
public class CurrencyModel : PageModel
{
    private readonly LinkGenerator _link;
    public CurrencyModel(LinkGenerator linkGenerator)
    {
        _link = linkGenerator;
    }

    public void OnGet()
    {
        var url1 = Url.Page("Currency/View", new { id = 5 });
        var url3 = _link.GetPathByPage(
            HttpContext,
            "/Currency/View",
            values: new { id = 5 });
        var url2 = _link.GetPathByPage(
            "/Currency/View",
            values: new { id = 5 });
        var url4 = _link.GetUriByPage(
            page: "/Currency/View",
            handler: null,
            values: new { id = 5 },
            scheme: "https",
            host: new HostString("example.com"));
    }
}
```

Вы можете создавать относительные пути, используя Url.Page.
Вы можете использовать относительные или абсолютные пути к страницам

Доступ к LinkGenerator можно получить с помощью внедрения зависимостей

GetPathByPage эквивалентен Url.Page и генерирует относительный URL-адрес
Другие перегруженные варианты не требуют HttpContext

GetUriByPage генерирует абсолютный URL-адрес вместо относительного URL-адреса

ВНИМАНИЕ! Как всегда, нужно быть осторожным при создании URL-адресов, независимо от того, используете вы IUrlHelper или LinkGenerator. Если вы ошиблись – используете неправильный путь или не указали обязательный параметр маршрута, – сгенерированный URL-адрес будет пустым.

На данном этапе мы рассмотрели сопоставление URL-адресов запросов с Razor Pages и создание URL-адресов, но большинство адресов, которые мы использовали, были довольно уродливыми. Если вас беспокоят заглавные буквы в адресах, то следующий раздел для вас. В разделе 14.6 мы настраиваем соглашения, которые приложение использует для формирования шаблонов маршрутов.

14.6 Настройка соглашений с помощью Razor Pages

Razor Pages основан на ряде соглашений, призванных уменьшить объем шаблонного кода, который нужно писать. В этом разделе вы увидите способы настройки этих условных соглашений. Настраивая соглашения, используемые Razor Pages в приложении, вы получаете полный контроль над URL-адресами приложения без необходимости вручную настраивать каждый шаблон маршрута Razor Page.

По умолчанию ASP.NET Core генерирует URL-адреса, которые очень точно соответствуют именам файлов страниц Razor. Например, страница Razor, расположенная по пути `Pages/Products/ProductDetails.cshtml`, будет соответствовать шаблону маршрута `Products/ProductDetails`.

В наши дни нечасто можно увидеть заглавные буквы в URL-адресах, а слова в адресах обычно разделяются с помощью дефиса, а не прописной буквы, например `product-details` вместо `ProductDetails`. Наконец, также часто бывает, что URL-адреса оканчиваются косой чертой, скажем `/product-details/` вместо `/product-details`. Razor Pages дает вам полный контроль над соглашениями, которые приложение использует для создания шаблонов маршрутов, но вот два распространенных изменения, которые часто вношу я.

В главе 6 вы видели, как внести некоторые из этих изменений, настроив объект `RouteOptions` для своего приложения. Вы можете написать URL-адреса строчными буквами и убедиться, что они уже имеют косую черту в конце, как показано в следующем листинге.

Листинг 14.5. Настройка соглашений о маршрутизации с использованием объекта `RouteOptions` в файле `Program.cs`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages();
builder.Services.Configure<RouteOptions>(o =>
{
    o.LowercaseUrls = true;
    o.LowercaseQueryStrings = true;
    o.AppendTrailingSlash = true;
});

WebApplication app = builder.Build();

app.MapRazorPages();

app.Run();
```

Изменяет соглашения, используемые для создания URL-адресов. По умолчанию эти свойства имеют значение `false`

Чтобы использовать дефисы для своего приложения, нужно создать преобразователь настраиваемых параметров, а это раздражает. Это довольно сложная тема, но в данном случае такой вариант относительно просто реализовать. В следующем листинге показано, как создать преобразователь параметров, который использует регулярное выражение для замены значений с прописными буквами в сгенерированном URL-адресе на дефисы.

Листинг 14.6 Создание преобразователя параметров для замены на дефисы

```
public class KebabCaseParameterTransformer : IOutboundParameterTransformer
```

Создаем класс, реализующий интерфейс `IOutboundParameterTransformer`

```

{
    public string TransformOutbound(object? value)
    {
        if (value is null) return null; ← | Защита от нулевых значе-
                                         | ний, чтобы избежать исключе-
                                         |ний во время выполнения
        return Regex.Replace(value.ToString(),
            "([a-z])([A-Z])", "$1-$2").ToLower(); | Регулярное выра-
                                         |жение заменяет ша-
                                         |блонны PascalCase на
                                         |дефисы (kebab-case)
    }
}

```

Генераторы исходного кода в .NET 7

Одной из интересных функций, представленных в C# 9, были генераторы исходного кода. Генераторы исходного кода – это функция компилятора, позволяющая проверять код по мере его компиляции и на лету генерировать новые файлы C#, которые включаются в компиляцию. Генераторы исходного кода могут значительно сократить шаблонный код, необходимый для некоторых функций, и повысить производительность, полагаясь на анализ во время компиляции, а не на рефлексию во время выполнения.

В .NET 6 появилось несколько реализаций генератора исходного кода, например высокопроизводительный API журналирования, о котором я рассказываю в этом посте: <http://mng.bz/Y1GA>. Даже компилятор Razor, используемый для компиляции файлов с расширением .cshtml, был переписан для применения генераторов исходного кода!

В .NET 7 было добавлено множество новых генераторов исходного кода. Одним из таких генераторов является генератор регулярных выражений, который может улучшить производительность экземпляров Regex, например, в листинге 14.6. Фактически если вы используете IDE, например Visual Studio, вы должны увидеть исправление кода, предлагающее использовать новый шаблон. После применения исправления кода листинг 14.6 должен выглядеть следующим образом, который функционально идентичен, но, вероятно, будет работать быстрее:

```

partial class KebabCaseParameterTransformer : IOutboundParameterTransformer
{
    public string? TransformOutbound(object? value)
    {
        if (value is null) return null;

        return MyRegex().Replace(value.ToString(), "$1-$2").ToLower();
    }

    [GeneratedRegex("([a-z])([A-Z])")]
    private static partial Regex MyRegex();
}

```

Если вы хотите узнать больше о том, как работает этот генератор исходного кода и как он может повысить производительность, прочтите пост на странице <http://mng.bz/GyEO>. Если вы хотите узнать больше о генераторах исходного кода или даже написать свои собственные, прочтите мою серию статей об этом процессе: <http://mng.bz/zX4Q>.

Можно зарегистрировать преобразователь параметров в своем приложении с помощью метода расширения `AddRazorPagesOptions()` в файле `Program.cs`. Он идет после метода `AddRazorPages()` и может использоваться для настройки соглашений, используемых Razor Pages. В следующем листинге показано, как зарегистрировать преобразователь для дефисов, а также как добавить соглашение о дополнительном маршруте для данной страницы Razor.

Листинг 14.7. Регистрация преобразователя параметров с помощью `RazorPagesOptions`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages()
    .AddRazorPagesOptions(opts => {
        opts.Conventions.Add(
            new PageRouteTransformerConvention(
                new KebabCaseParameterTransformer()));
        opts.Conventions.AddPageRoute(
            "/Search/Products/StartSearch", "/search-products");
    });
WebApplication app = builder.Build();
app.MapRazorPages();
app.Run();
```

`builder.Services.AddRazorPages()` Метод расширения `AddRazorPagesOptions` можно использовать для настройки соглашений, используемых Razor Pages

`opts.Conventions.Add(new PageRouteTransformerConvention(new KebabCaseParameterTransformer()));` Регистрирует преобразователь параметров как соглашение, используемое всеми страницами Razor Pages

`opts.Conventions.AddPageRoute("/Search/Products/StartSearch", "/search-products");` Метод `AddPageRoute` добавляет дополнительный шаблон маршрута для Pages/Search/Products/StartSearch.cshtml

Соглашение `AddPageRoute` добавляет альтернативный способ выполнения страницы Razor. В отличие от того, когда вы настраиваете шаблон маршрута для страницы Razor с помощью директивы `@page`, при использовании `AddPageRoute` для страницы добавляется дополнительный шаблон маршрута вместо замены того, что есть по умолчанию. Это означает, что есть два шаблона маршрута, по которым можно получить доступ к странице.

COBET Даже имя корневой папки `Pages` – это соглашение, которое можно настроить! Это можно сделать, задав свойство `RootDirectory` в лямбда-функции конфигурации `AddRazorPageOptions()`.

Если вам нужен еще больший контроль над шаблонами маршрутов Razor Pages, можно реализовать собственное соглашение, реализовав интерфейс `IPageRouteModelConvention` и зарегистрировав его как собственное соглашение. `IPageRouteModelConvention` – это один из трех мощных интерфейсов Razor Pages, которые позволяют настраивать работу приложения Razor Pages:

- `IPageRouteModelConvention` – используется для настройки шаблонов маршрутов для всех страниц Razor в приложении;
- `IPageApplicationModelConvention` – используется для настройки спо-

соба обработки страницы Razor, например для автоматического добавления фильтров на страницу Razor. Вы узнаете о фильтрах в Razor Pages в главах 21 и 22;

- `IPageHandlerModelConvention` – используется для настройки способа обнаружения и выбора обработчиков страниц.

Это мощные интерфейсы, поскольку они предоставляют доступ ко всем внутренним компонентам соглашений и конфигурации страницы Razor. Можно использовать `IPageRouteModelConvention`, например, чтобы переписать все шаблоны маршрутов для страниц Razor или автоматически добавлять маршруты. Это особенно полезно, если вам нужно локализовать приложение, чтобы можно было использовать URL-адреса на нескольких языках, которые сопоставляются с одной и той же страницей Razor.

В листинге 14.8 показан простой пример с `IPageRouteModelConvention`, который добавляет фиксированный префикс "page" ко всем маршрутам в приложении. Если у вас есть страница `Pages/Privacy.cshtml` с шаблоном маршрута по умолчанию "Privacy", то после добавления следующего соглашения она также будет иметь шаблон маршрута "page/Privacy".

Листинг 14.8. Создание пользовательской IPageRouteModelConvention

```
public class PrefixingPageRouteModelConvention : IPageRouteModelConvention
{
    public void Apply(RouteModel model)
    {
        var selectors = model.Selectors
            .Select(selector => new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Template = AttributeRouteModel.CombineTemplates(
                    "page",
                    selector.AttributeRouteModel!.Template)
            }
        })
        .ToList();
        foreach(var newSelector in selectors)
        {
            model.Selectors.Add(newSelector);
        }
    }
}
```

Вы можете добавить соглашение в приложение в вызове метода `AddRazorPagesOptions()`. Следующее соглашение применяется ко всем страницам:

```
builder.Services.AddRazorPages().AddRazorPagesOptions(opts =>
{
    opts.Conventions.Add(new PrefixingPageRouteModelConvention());
});
```

Существует множество способов настройки соглашений для приложений Razor Pages, но в большинстве случаев в этом нет необходимости. Если вы обнаружите, что вам необходимо каким-либо образом настроить все страницы приложения, то в документации Microsoft «Соглашения и маршруты приложений Razor Pages в ASP.NET Core» содержится дополнительная информация: <http://mng.bz/A0BK>.

Соглашения – ключевая особенность Razor Pages, и следует опираться на них всякий раз, когда это возможно. Хотя вы можете определить шаблоны маршрутов для отдельных страниц Razor вручную, как показано в предыдущих разделах, я не рекомендую этим злоупотреблять. В частности:

- *избегайте* замены шаблона маршрута на абсолютный путь в директиве страницы `@page`;
- *избегайте* добавления литеральных сегментов в директиву `@page`. Вместо этого используйте файловую иерархию;
- *избегайте* добавления дополнительных шаблонов маршрутов на страницу Razor с помощью соглашения `AddPageRoute()`. Наличие нескольких URL-адресов для доступа к странице иногда может сбивать с толку;
- *добавьте* параметры маршрута в директиву `@page`, чтобы сделать свои маршруты динамическими, например `@page "{name}"`;
- *рассмотрите* возможность использования глобальных соглашений, если хотите изменить шаблоны маршрутов для всех своих страниц Razor, например используя дефисы, как в предыдущем разделе.

Вкратце: эти правила сводятся к «соблюдению соглашений». Опасность, если вы не придерживаетесь этих правил, заключается в том, что вы можете случайно создать две страницы Razor с шаблонами маршрутов, которые будут перекрывать друг друга. К сожалению, если вы попадете в такую ситуацию, то *не* получите ошибку во время компиляции. Вместо этого вы получите исключение во время выполнения, когда ваше приложение получит запрос, соответствующий нескольким шаблонам маршрутов, как показано на рис. 14.4.

Сейчас мы рассмотрели практически все, что касается маршрутизации в Razor Pages. По большей части маршрутизация в Razor Pages работает как в минимальных API. Основное отличие состоит в том, что шаблоны маршрутов создаются с использованием соглашений. Когда дело доходит до другой половины маршрутизации – генерации URL-адресов, – Razor Pages и минимальные API также похожи, но Razor Pages предоставляет вам несколько отличных помощников.

Поздравляю, вы прошли все этапы подробного обсуждения маршрутизации! Надеюсь, вас не слишком смущили отличия от маршрутизации в минимальных API. Мы еще вернемся к маршрутизации, когда я буду описывать, как создавать веб-API в главе 20, но будьте уверены, что мы уже рассмотрели все хитрые подробности в этой главе!

Маршрутизация управляет привязкой входящих запросов к странице Razor, но мы не видели, как в этом участвуют обработчики страниц. В главе 15 вы узнаете все об обработчиках страниц: как они выбираются, как генерируют ответы и как корректно обрабатывать ответы об ошибках.

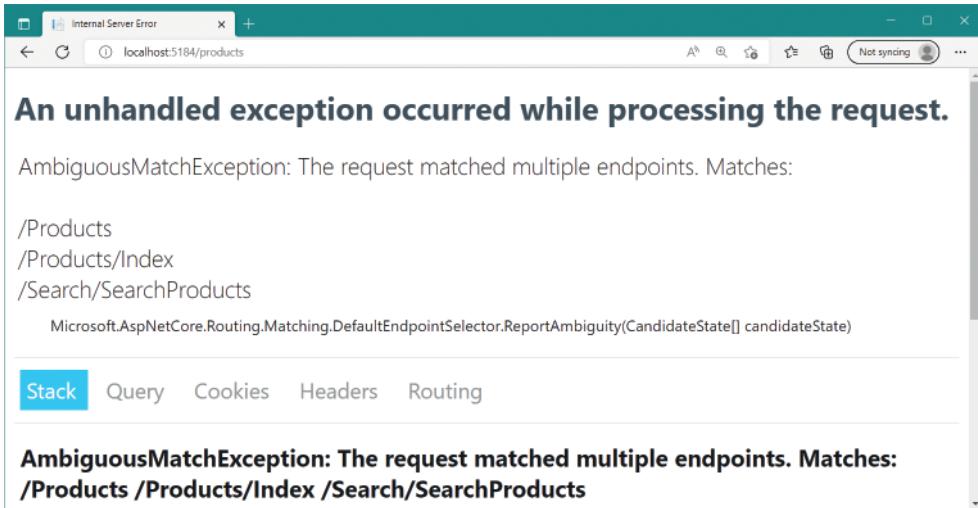


Рис. 14.4 Если несколько страниц Razor зарегистрированы с перекрывающимися шаблонами маршрутов, вы получите исключение во время выполнения, когда маршрутизатор не сможет решить, какой из них выбрать

Резюме

- Маршрутизация – это процесс сопоставления URL-адреса входящего запроса с конечной точкой, которая будет выполняться для генерации ответа. Каждая страница Razor является конечной точкой, и для каждого запроса выполняется один обработчик страницы;
- вы можете определить сопоставление между URL-адресами и конечной точкой в своем приложении, используя маршрутизацию на основе соглашений или явную маршрутизацию. Минимальные API используют явную маршрутизацию, где каждая конечная точка имеет соответствующий шаблон маршрута;
- контроллеры MVC часто используют традиционную маршрутизацию, при которой один шаблон соответствует нескольким контроллерам, но также могут использовать явную маршрутизацию или маршрутизацию на основе атрибутов. Razor Pages находится посередине; он использует соглашения для создания явных шаблонов маршрутов для каждой страницы;
- по умолчанию у каждой страницы Razor есть отдельный шаблон маршрута, который соответствует ее пути в папке Pages, поэтому страница Pages/Products/View.cshtml имеет шаблон маршрута Products/View. Эти настройки по умолчанию на основе файлов позволяют легко визуализировать URL-адреса, предоставляемые приложением;
- у страницы Index.cshtml два шаблона маршрутов: один с суффиксом /Index, а другой – без. Например, у страницы Pages/Products/Index.cshtml два шаблона: Products/Index и Products. Это соответ-

ствует общепринятым поведению файлов index.html в традиционных HTML-приложениях;

- вы можете добавить сегменты в шаблон страницы Razor, добавив его к директиве @page, например @page "{id}". Любые дополнительные сегменты добавляются к шаблону маршрута по умолчанию. Вы можете включать как литеральные сегменты шаблонов, так и сегменты шаблонов маршрутов, которые можно использовать для придания динаминости страницам Razor. Вы можете заменить шаблон маршрута для страницы Razor, начав его с символа "/", как в @page "/contact";
- вы можете использовать IUrlHelper для создания URL-адресов в виде строки на основе имени действия или страницы Razor. IUrlHelper можно использовать только в контексте запроса, и он использует внешние значения маршрутизации из текущего запроса. Это упрощает создание ссылок для Razor Pages в той же папке, что и выполняющийся в данный момент запрос, но также добавляет несогласованности, поскольку один и тот же вызов метода генерирует разные URL-адреса в зависимости от того, где он вызывается;
- класс LinkGenerator можно использовать для создания URL-адресов из других сервисов приложения, где у вас нет доступа к объекту HttpContext. Методы LinkGenerator более многословные, чем их эквиваленты в IUrlHelper, но они однозначны, поскольку не используют внешние значения из текущего запроса;
- вы можете управлять соглашениями о маршрутизации, используемыми ASP.NET Core, настроив объект RouteOptions, например задав все URL-адреса в нижнем регистре или всегда добавляя косую черту в конце;
- вы можете добавить дополнительные соглашения о маршрутизации для страниц Razor, вызвав метод AddRazorPagesOptions() после метода AddRazorPages() в файле Program.cs. Эти соглашения могут контролировать отображение параметров маршрута и добавлять дополнительные шаблоны маршрутов для определенных страниц Razor.

15

Создание ответов с помощью обработчиков страниц в Razor Pages

В этой главе:

- выбор вызываемого обработчика страницы Razor для запроса;
- возврат объекта IActionResult из обработчика страницы;
- установка кода состояния ошибок с помощью компонента StatusCode-PagesMiddleware.

В главе 14 вы узнали, как система маршрутизации выбирает страницу Razor для выполнения на основе связанного с ней шаблона маршрута и URL-адреса запроса, но каждая страница Razor может иметь несколько обработчиков страницы. В этой главе вы узнаете все об обработчиках страниц, их обязанностях и о том, как отдельная страница Razor выбирает, какой обработчик выполнить для запроса.

В разделе 15.3 мы рассмотрим некоторые способы получения значений из HTTP-запроса в обработчике страницы. Как и в минимальных API, обработчики страниц могут принимать аргументы метода, привязанные к значениям в HTTP-запросе, но Razor Pages также может привязывать запрос к свойствам модели страницы.

В разделе 15.4 вы узнаете, как возвращать объекты `IActionResult` из обработчиков страниц. Затем мы рассмотрим некоторые распространенные типы `IActionResult`, которые вы будете возвращать из обработчиков страниц для генерации HTML и ответов перенаправления.

Наконец, в разделе 15.5 вы узнаете, как использовать `StatusCodesMiddleware` для улучшения ответов с кодами состояния ошибок в конвейере промежуточного ПО. Данный компонент перехватывает ответы об ошибках, например базовые ответы с кодом состояния 404, и повторно запускает конвейер, чтобы сгенерировать красивый HTML-ответ для ошибки. Это позволяет пользователям получить более приятные впечатления, когда они сталкиваются с ошибкой при работе с приложением Razor Pages. Мы начнем с краткого рассмотрения обязанностей обработчика страницы, а затем перейдем к рассмотрению того, как инфраструктура Razor Pages выбирает, какой обработчик страницы выполнять.

15.1 Razor Pages и обработчики страниц

В главе 13 я описал паттерн проектирования MVC и его связь с ASP.NET Core. В этом паттерне «контроллер» получает запрос и является точкой входа для генерации пользовательского интерфейса. Для Razor Pages точкой входа является обработчик страницы, который находится в модели страницы Razor. *Обработчик страницы* – это метод, который выполняется в ответ на запрос.

Ответственность обработчика страницы обычно состоит из трех задач:

- подтвердить, что входящий запрос действителен;
- вызвать бизнес-логику, соответствующую входящему запросу;
- выбрать подходящий *тип* ответа, который нужно вернуть.

Обработчику страницы необязательно выполнять все эти действия, но он должен по крайней мере выбрать тип ответа, который нужно вернуть. Обработчики страниц обычно возвращают одно из трех:

- *объект PageResult* – заставляет связанное представление Razor генерировать ответ в виде HTML-кода;
- *ничего* (обработчик возвращает `void` или `Task`) – аналогично предыдущему случаю, заставляя представление Razor генерировать ответ в виде HTML-кода;
- *RedirectToPageResult* – указывает на то, что пользователь должен быть перенаправлен на другую страницу в приложении.

Это наиболее часто используемые результаты для Razor Pages, но я опишу ряд дополнительных вариантов в разделе 15.4.

Важно понимать, что обработчик страницы не генерирует ответ напрямую; он выбирает тип ответа и готовит для него данные. Например, возвращая `PageResult`, в этот момент он не генерирует никакого HTML-кода; он просто указывает на то, что представление должно быть визуализировано. Это соответствует паттерну проектирования MVC, где ответ генерирует *представление*, а не *контроллер*.

Также стоит помнить, что обработчики страниц обычно не должны выполнять бизнес-логику напрямую. Вместо этого они должны вызы-

вать соответствующие сервисы в модели приложения для обработки запросов. Если обработчик страницы получает запрос на добавление продукта в корзину пользователя, он не должен напрямую манипулировать базой данных или пересчитывать общую сумму покупки. Он должен вызвать еще один класс для обработки деталей. Такой подход разделения ответственности гарантирует, что ваш код легко можно будет тестировать и сопровождать, по мере того как он будет расти.

СОВЕТ Обработчик страницы отвечает за выбор типа ответа, который нужно вернуть; *дэйлжок представлений* в MVC использует результат для генерации ответа.

15.2 Выбор обработчика страницы для вызова

В главе 14 я сказал, что маршрутизация заключается в сопоставлении URL-адресов с обработчиком. Для страниц Razor это означает обработчик страницы, но я несколько раз упоминал, что эти страницы могут содержать несколько обработчиков страниц. В этом разделе вы узнаете, как `EndpointMiddleware` выбирает, какой обработчик страницы вызывать при выполнении страницы Razor.

Как показано в главе 14, путь к странице Razor на диске управляет шаблоном маршрута по умолчанию для страницы Razor. Например, страница Razor по пути `Pages/Products/Search.cshtml` имеет шаблон маршрута по умолчанию – `Products/Search`. При получении запроса с URL-адресом `/products/search` `RoutingMiddleware` выбирает эту страницу Razor, и запрос проходит через конвейер промежуточного ПО к `EndpointMiddleware`. На этом этапе `EndpointMiddleware` должен выбрать, какой обработчик страницы выполнить, как показано на рис. 15.1.

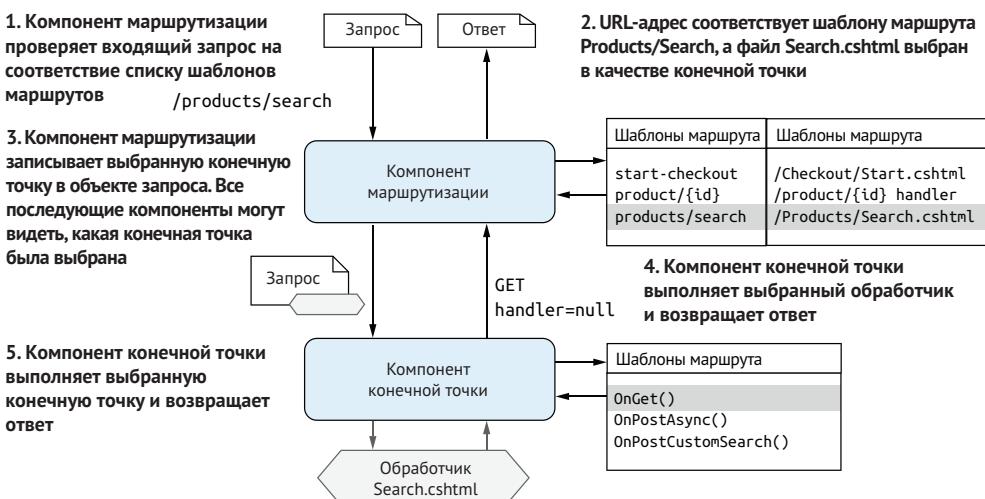


Рис. 15.1 Компонент маршрутизации выбирает страницу Razor для выполнения на основе URL-адреса входящего запроса. Затем компонент конечной точки выбирает конечную точку для выполнения на основе HTTP-метода запроса и наличия (или отсутствия) значения обработчика в маршруте

Рассмотрим SearchModel страницы Razor, показанной в следующем листинге. У этой страницы три обработчика: OnGet, OnPostAsync и OnPostCustomSearch. Тела методов обработчика не показаны, поскольку нас интересует только то, как EndpointMiddleware выбирает, какой обработчик вызывать.

Листинг 15.1. Страница Razor с несколькими обработчиками страниц

```
public class SearchModel : PageModel
{
    public void OnGet() ← Обрабатывает GET-запросы
    {
        // Реализация обработчика;
    }

    public Task OnPostAsync() ← Обрабатывает POST-запросы. Суффикс
    {                               async не является обязательным
        // Реализация обработчика;      и игнорируется при маршрутизации
    }

    public void OnPostCustomSearch() ← Обрабатывает запросы POST,
    {                               в которых значение обра-
        // Реализация обработчика;      ботчика в маршруте равно
    }                               CustomSearch
}
```

Razor Pages может содержать любое количество обработчиков, но в ответ на заданный запрос выполняется только один. Когда EndpointMiddleware выполняет выбранную страницу Razor, то выбирает обработчик страницы, который нужно будет вызвать на основе двух переменных:

- HTTP-метод, используемый в запросе (например, GET, POST или DELETE);
- значение обработчика (`handler`) в маршруте.

Значение маршрута `handler` обычно берется из значения строки запроса в URL-адресе запроса, например `/Search?Handler=CustomSearch`. Если вам не нравится внешний вид строк запроса (мне он не нравится!), то можно включить параметр маршрута `{handler}` в шаблон маршрута страницы Razor. Например, для страницы поиска из листинга 15.2 можно обновить директиву страницы:

```
@page "{handler?}"
```

Так вы получите полный шаблон маршрута что-то вроде "Search/{handler?}", который будет соответствовать таким URL-адресам, как `/Search` и `/Search/CustomSearch`.

EndpointMiddleware использует значение обработчика из маршрута и HTTP-метод вместе со стандартным соглашением об именах, чтобы определить, какой обработчик страницы выполнить, как показано на рис. 15.2. Параметр обработчика является необязательным и обычно предоставляется как часть строки запроса или как параметр маршру-

та, как описано выше. Суффикс `async` также является необязательным и часто применяется, когда обработчик использует конструкции асинхронного программирования, например `Task` или `async/await`.

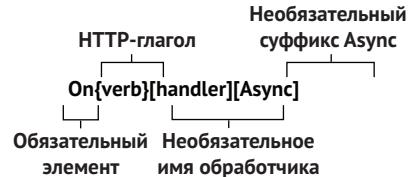


Рис. 15.2 Обработчики страницы Razor со-поставляются с запросом на основе HTTP-метода и необязательного параметра обра-ботчика

ПРИМЕЧАНИЕ Соглашение об именах асинхронных суффиксов предложено корпорацией Microsoft, хотя оно и непопулярно среди некоторых разработчиков. NServiceBus приводит обоснованный аргумент (вместе с советом от Microsoft) на странице <http://mng.bz/e59P>.

Основываясь на этом соглашении, теперь можно определить, какой тип запроса соответствует каждому обработчику страницы из листинга 15.1:

- `OnGet` – вызывается для GET-запросов, в которых не указано значение обработчика;
- `OnPostAsync` – вызывается для запросов методом POST, в которых не указано значение обработчика. Он возвращает `Task`, поэтому использует суффикс `Async`, который игнорируется при маршрутизации;
- `OnPostCustomSearch` – вызывается для запросов методом POST, в которых указано значение обработчика "CustomSearch".

Страница Razor из листинга 15.1 определяет три обработчика, поэтому может обрабатывать только три пары метод–обработчик. Но что произойдет, если вы получите запрос, который не соответствует им, например запрос с использованием команды `DELETE`, GET-запрос с непустым значением `handler` или POST-запрос с нераспознанным значением `handler`?

Во всех этих случаях `EndpointMiddleware` выполняет неявный обработчик страницы. Неявные обработчики страниц не содержат логики; они просто визуализируют представление Razor. Например, если вы отправили запрос с использованием команды `DELETE` на страницу Razor в листинге 15.1, то будет выполнен неявный обработчик. Неявный обработчик страницы эквивалентен следующему коду:

```
public void OnDelete() { }
```

ОПРЕДЕЛЕНИЕ Если обработчик страницы не соответствует HTTP-методу запроса и значению обработчика, выполняется **неявный обработчик страницы**, который отображает ассоциированное представление Razor. Неявные обработчики страниц участвуют в привязке модели и используют фильтры страниц, но не выполняют никакой логики.

Из правила неявного обработчика страниц есть одно исключение: если в запросе используется команда `HEAD` и нет соответствующего обработчика `OnHead`, Razor Pages выполнит обработчик `OnGet` (если он существует).

ПРИМЕЧАНИЕ Запросы с командой HEAD обычно отправляются браузером автоматически и не возвращают тело ответа. Как вы увидите в главе 28, они часто используются в целях безопасности.

Теперь, когда мы знаем, как выбирается обработчик страницы, можно подумать о том, как он выполняется.

15.3 Прием параметров в обработчиках страниц

В главе 7 вы узнали о тонкостях привязки модели в обработчиках конечных точек минимальных API. Как и минимальные API, обработчики страниц Razor могут использовать привязку модели, чтобы с легкостью извлекать значения из запроса. Подробности привязки модели Razor Page мы обсудим в главе 16; в этом разделе вы узнаете об основных механизмах привязки модели в Razor Pages и основных доступных опциях.

При работе с Razor Pages часто требуется извлечь значения из входящего запроса. Если запрос относится к странице поиска, запрос может содержать поисковый запрос и номер страницы в строке запроса. Если запрос отправляет форму в приложение, например когда пользователь выполняет вход со своим именем пользователя и паролем, эти значения могут быть закодированы в теле запроса. В других случаях значений не будет, например когда пользователь запрашивает домашнюю страницу приложения.

ОПРЕДЕЛЕНИЕ Процесс извлечения значений из запроса и преобразования их в типы .NET называется *привязкой модели*. Мы подробно обсудим ее в главе 16.

ASP.NET Core может привязывать в Razor Pages:

- *аргументы метода* – если у обработчика страницы есть параметры метода, аргументы привязываются и создаются на основе значений в запросе;
- *свойства, отмеченные атрибутом [BindProperty]* – любые свойства модели страницы, отмеченные этим атрибутом, будут привязаны к запросу. По умолчанию этот атрибут ничего не делает для GET-запросов.

Привязанные значения могут быть простыми типами, например строками и целыми числами, или сложными, как показано в следующем листинге. Если любое из значений, указанных в запросе, не привязано к свойству или аргументу обработчика страницы, дополнительные значения останутся неиспользованными.

Листинг 15.2 Пример обработчиков страниц Razor

```
public class SearchModel : PageModel
{
    private readonly SearchService _searchService;
    public SearchModel(SearchService searchService)
    {
        _searchService = searchService;
    }
}
```

SearchService внедряется из контейнера внедрения зависимостей для использования в обработчиках страниц

Обработчику страницы не нужно проверять допустимость модели. Возврат void визуализирует представление

```
[BindProperty]
public BindingModel Input { get; set; }
public List<Product> Results { get; set; } <-- Свойства, декорированные атрибутом [BindProperty], привязываются к модели
public void OnGet()
{
    } Параметр max привязывается к модели и использует значения из запроса
}
public IActionResult OnPost(int max) <-- Недекорированные свойства не привязываются к модели
{
    if (ModelState.IsValid)
    {
        Results = _searchService.Search(Input.SearchTerm, max);
        return Page();
    }
    return RedirectToAction("./Index");
}
}
```

Если запрос недействителен, метод указывает, что пользователя следует перенаправить на страницу индекса

В данном примере обработчик `OnGet` не требует каких-либо параметров, и это простой метод – он возвращает `void`, а это означает, что будет отображено связанное представление Razor. Он также мог бы вернуть `PageResult`; эффект был бы таким же. Обратите внимание, что этот обработчик предназначен для HTTP-запросов методом GET, поэтому свойство `Input`, декорированное атрибутом `[BindProperty]`, не привязывается.

СОВЕТ Чтобы привязать свойства и для запросов методом GET, используйте свойство атрибута `SupportsGet`, например `[BindProperty(SupportsGet=true)]`.

Обработчик `OnPost`, наоборот, принимает в качестве аргумента параметр `max`. В данном случае это простой тип `int`, но он также может быть сложным объектом. Кроме того, поскольку этот обработчик соответствует HTTP-запросу методом POST, свойство `Input` также привязывается к запросу.

ПРИМЕЧАНИЕ В отличие от большинства классов .NET, нельзя использовать перегрузку методов, чтобы иметь на странице Razor несколько обработчиков с одним и тем же именем.

Когда метод действия использует привязанные свойства или параметры, он всегда должен проверять валидность предоставленной модели с помощью свойства `ModelState.IsValid`. `ModelState` предоставляется как свойство в базовом классе `PageModel` и может использоваться для проверки достоверности всех привязанных свойств и параметров. Вы увидите, как работает этот процесс, в главе 16, когда мы будем изучать валидацию.

Как только обработчик страницы установит, что параметры метода, предоставленные для действия, являются валидными (допустимыми), он может выполнить соответствующую бизнес-логику и обработать запрос. В случае обработчика `OnPost` это включает в себя вызов предоставленного сервиса `SearchService` и установку результата в свой-

ство `Results`. Наконец, обработчик возвращает объект `PageResult`, вызывая вспомогательный метод базового класса `PageModel`.

```
return Page();
```

Если модель не является валидной, у вас не будет результатов для отображения! В этом примере действие возвращает объект `RedirectToActionResult` с помощью вспомогательного метода `RedirectToPage`. При выполнении этот результат отправит пользователю ответ с кодом состояния `302`, который заставит браузер перейти на страницу `Index`.

Обратите внимание, что метод `OnGet` возвращает `void` в сигнатуре метода, тогда как метод `OnPost` возвращает объект `IActionResult`. Это необходимо в методе `OnPost`, чтобы позволить C# выполнить компиляцию (поскольку вспомогательные методы `Page()` и `RedirectToPage()` возвращают разные типы), но не меняет окончательного поведения методов. С таким же успехом можно было бы вызвать `Page` в методе `OnGet` и вернуть объект `IActionResult`. Поведение было бы идентичным.

COBET Если вы возвращаете несколько типов результатов из обработчика страницы, необходимо убедиться, что ваш метод возвращает объект `IActionResult`.

В листинге 15.2 я использовал методы `Page()` и `RedirectToPage()` для генерации возвращаемого значения. Экземпляры `IActionResult` можно создавать и возвращать с использованием оператора `new` языка C#:

```
return new PageResult()
```

Однако базовый класс Razor Pages, `PageModel`, также предоставляет несколько вспомогательных методов для генерации ответов, которые представляют собой тонкую обертку синтаксиса `new`. Обычно для генерации соответствующего экземпляра `PageResult` используется метод `Page()`, метод `RedirectToPage()` для создания экземпляра `RedirectToPageResult` или метод `NotFound()` для создания экземпляра `NotFoundResult`.

COBET Большинство реализаций `IActionResult` имеют вспомогательный метод базового класса `PageModel`. Обычно они называются `Type`, а сгенерированный результат называется `TypeResult`. Например, метод `StatusCode()` возвращает экземпляр `StatusCodeResult`.

В следующем разделе мы более подробно рассмотрим некоторые распространенные типы `IActionResult`.

15.4 Возврат ответов `IActionResult`

В предыдущем разделе я подчеркнул, что обработчики страниц решают, какой тип ответа возвращать, но сами не генерируют ответ. Этот `IActionResult`, возвращаемый обработчиком страницы, при выполнении инфраструктурой Razor Pages с использованием движка представлений будет генерировать ответ.

ВНИМАНИЕ! Обратите внимание, что тип интерфейса – IActionResult, а не IResult. IResult используется в минимальных API, и его, как правило, следует избегать в Razor Pages (и контроллерах MVC). В .NET 7 типы IResult, возвращаемые из Razor Pages или контроллеров MVC, работают, как и ожидалось, но у них нет тех же функций, что и у IActionResult, поэтому в Razor Pages следует отдавать предпочтение IActionResult.

Такой подход является ключом к следованию паттерну проектирования MVC. Он отделяет решение о том, какой ответ отправить, от генерации ответа, что позволяет с легкостью протестировать логику метода действий, чтобы подтвердить отправку правильного типа ответа. Затем можно отдельно проверить, например, что данный объект IActionResult генерирует ожидаемый HTML-код.

В ASP.NET Core есть много разных типов IActionResult:

- **PageResult** – генерирует HTML-представление для связанной страницы в Razor Pages и возвращает ответ с кодом состояния 200;
- **ViewResult** – генерирует HTML-представление для заданного представления Razor при использовании контроллеров MVC и возвращает ответ с кодом состояния 200;
- **PartialViewResult** – отображает часть HTML-страницы с использованием заданного представления Razor и возвращает ответ с кодом состояния 200; обычно используется с контроллерами MVC и запросами AJAX;
- **RedirectToPageResult** – отправляет ответ с кодом состояния 302 для автоматического перенаправления пользователя на другую страницу;
- **RedirectResult** – отправляет ответ с кодом состояния 302 для автоматического перенаправления пользователя на указанный URL-адрес (это необязательно должна быть страница Razor);
- **FileResult** – возвращает ответ в виде файла. Это базовый класс с несколькими производными типами:
- **FileContentResult** – возвращает byte[] в виде ответа файла браузеру;
- **FileStreamResult** – возвращает содержимое потока в виде ответа файла браузеру;
- **PhysicalFileResult** – возвращает содержимое файла на диске в качестве ответа браузеру;
- **ContentResult** – возвращает предоставленную строку в качестве ответа;
- **StatusResult** – отправляет код состояния в качестве ответа, не обязательно со связанным содержимым тела ответа;
- **NotFoundResult** – в качестве ответа отправляет код состояния 404.

Каждый из них, когда выполняется Razor Pages, генерирует ответ для обратной отправки через конвейер промежуточного ПО и передачи пользователю.

СОВЕТ При работе с Razor Pages некоторые из этих результатов действий обычно не используются, например ContentResult

и `StatusCodeResult`. Тем не менее полезно знать о них, поскольку вы, вероятно, будете использовать их при создании веб-API с помощью контроллеров MVC, как будет показано в главе 20.

В разделах 15.4.1–15.4.3 я приведу краткое описание наиболее распространенных типов `IActionResult`, которые вы будете использовать с Razor Pages.

15.4.1 `PageResult` и `RedirectToPageResult`

При создании традиционного веб-приложения с помощью Razor Pages обычно используется `PageResult`, который генерирует ответ в виде HTML-кода с помощью Razor. Мы подробно рассмотрим, как это происходит, в главе 17.

Вы также будете часто использовать различные результаты на основе переадресации, чтобы перенаправить пользователя на новую веб-страницу. Например, когда вы размещаете заказ на сайте онлайн-магазина, то обычно перемещаетесь по нескольким страницам, как показано на рис. 15.3. Веб-приложение отправляет HTTP-перенаправления всякий раз, когда нужно перейти на другую страницу, например когда пользователь отправляет форму. Ваш браузер автоматически следует запросам перенаправления, создавая плавный поток процесса оформления заказа.

В этом потоке всякий раз, когда вы возвращаете HTML-код, используется `PageResult`; при перенаправлении на новую страницу используется `RedirectToPageResult`.

СОВЕТ Страницы Razor, как правило, не имеют состояния, поэтому если вы хотите сохранить данные между несколькими страницами, то необходимо поместить их в базу данных или подобное хранилище. Если вы просто хотите сохранить данные для одного запроса, то можно использовать `TempData`, который хранит небольшие объемы данных в файлах cookie для одного запроса. Подробности см. в документации: <http://mng.bz/XdXp>.

15.4.2 `NotFoundResult` и `StatusResult`

Помимо HTML-кода и переадресации, иногда нужно будет отправлять определенные коды состояния HTTP. Если вы запрашиваете страницу для просмотра продукта в приложении онлайн-торговли, а этого продукта не существует, браузер возвращается код состояния 404, и вы видите веб-страницу «Не найдено». Для этого Razor Pages возвращает `NotFoundResult`, который вернет код состояния 404. Аналогичного результата можно добиться, используя `StatusResult` и явно задав для кода состояния значение 404.

Обратите внимание, что `NotFoundResult` генерирует не HTML-разметку, а только код состояния 404 и возвращает его через конвейер промежуточного ПО. Обычно это не очень удобно для пользователя, поскольку браузер чаще всего отображает страницу по умолчанию, как, например, ту, что показана на рис. 15.4.

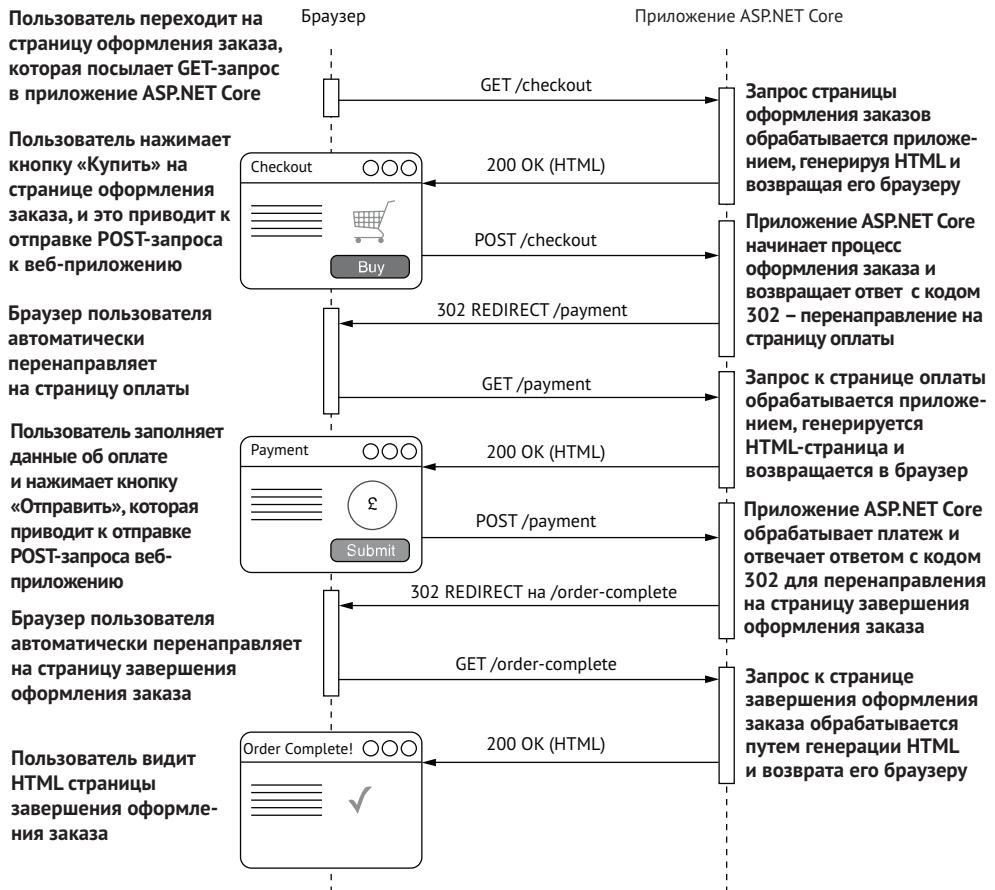


Рис. 15.3 Типичный поток с использованием POST, REDIRECT, GET. Пользователь отправляет свою корзину покупок на страницу оформления заказа, которая проверяет ее содержимое и перенаправляет на страницу оплаты без необходимости вручную изменять URL-адрес

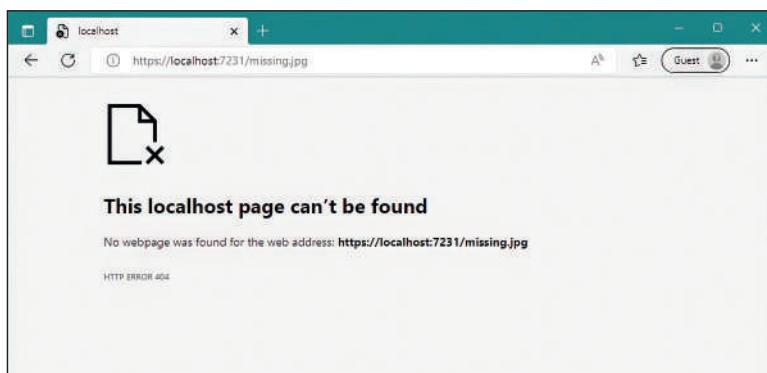


Рис. 15.4 Если вы вернете код состояния 404 без HTML, браузер отобразит стандартную страницу по умолчанию. Такое сообщение не особо полезно для пользователей и может привести многих из них в замешательство, или они могут подумать, что ваше веб-приложение не работает

Возврат такого рода кодов состояния удобен при создании API, но для приложения Razor Pages этого редко бывает достаточно. В разделе 15.5 вы узнаете, как перехватить код состояния 404, после того как он был сгенерирован, и вместо этого предоставить удобный для пользователя ответ в формате HTML.

15.5 Обработка кодов состояния с помощью `StatusCodePagesMiddleware`

В главе 4 мы обсуждали компонент обработки ошибок, предназначенный для перехвата исключений, сгенерированных в любом месте конвейера промежуточного ПО, их перехвата и генерации удобного для пользователя ответа. В этом разделе вы узнаете об аналогичном компоненте, который перехватывает коды состояния ошибок HTTP: `StatusCodePagesMiddleware`.

Ваше приложение может возвращать широкий спектр кодов состояния HTTP, указывающих на состояние ошибки. Вы уже видели, что код состояния 500 («ошибка сервера») отправляется при возникновении необработанного исключения, а ошибка 404 («файл не найден») отправляется, когда URL-адрес не обрабатывается каким-либо компонентом. В частности, очень распространены ошибки 404, возникающие, когда пользователь вводит недопустимый URL-адрес.

СОВЕТ Помимо указания полностью необработанного URL-адреса, такие ошибки часто используются, чтобы указать на то, что конкретный запрашиваемый объект не был найден. Например, запрос сведений о продукте с идентификатором 23 может вернуть ошибку 404, если такого продукта не существует. Они также генерируются автоматически, если ни одна конечная точка в приложении не соответствует URL-адресу запроса.

Возврат «необработанных» кодов состояния без дополнительного контента обычно допустим, если вы создаете приложение минимальных API или веб-API. Но, как упоминалось ранее, для приложений, используемых непосредственно пользователями, таких как приложения Razor Pages, это может привести к проблемам между пользователем и сайтом. Если вы не обработаете эти коды состояния, пользователи увидят стандартную страницу с ошибкой, как показано на рис. 15.4. Это может привести многих пользователей в замешательство, и они подумают, что ваше приложение не работает. Лучше будет обработать эти коды ошибок и вернуть страницу с ошибкой, которая соответствует остальной части приложения или по крайней мере не создает впечатления, что оно неисправно.

Компания Microsoft предоставляет компонент `StatusCodePagesMiddleware` для обработки этого варианта использования. Как и все компоненты для обработки ошибок, его следует добавить в начало конвейера, поскольку он будет обрабатывать только ошибки, сгенерированные более поздними компонентами.

Вы можете использовать его различными способами в своем приложении. Самый простой подход – добавить компонент в конвейер без каких-либо дополнительных настроек:

```
app.UseStatusCodePages();
```

С помощью этого метода компонент будет перехватывать любой ответ, который имеет код состояния HTTP, начинающийся с 4xx или 5xx и не имеющий тела ответа. В простейшем случае, когда вы не предоставляете никакой дополнительной конфигурации, компонент добавит тело ответа в виде простого текста с указанием типа и имени ответа, как показано на рис. 15.5. На данный момент это, возможно, хуже, чем сообщение по умолчанию, но это отправная точка для обеспечения более единообразного взаимодействия с пользователями.

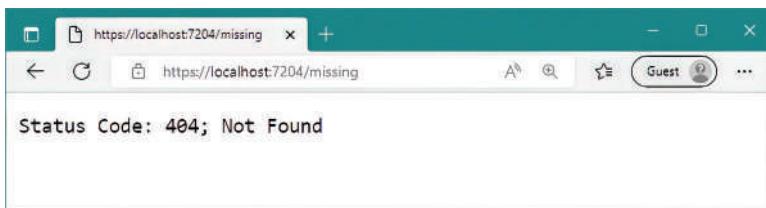


Рис. 15.5 Страница кода состояния для ошибки 404. Вы вряд ли будете использовать эту версию в промышленном окружении, поскольку она не обеспечивает надлежащего взаимодействия с пользователем, но демонстрирует, что коды ошибок перехватываются правильно

Более типичным подходом к использованию `StatusCodePagesMiddleware` в промышленном окружении является повторное выполнение конвейера при перехвате ошибки с использованием техники, аналогичной `ExceptionHandlerMiddleware`. Это позволяет вам иметь динамические страницы ошибок, которые соответствуют остальной части вашего приложения. Чтобы использовать данный метод, замените вызов `UseStatusCodePages` следующим методом расширения:

```
app.UseStatusCodePagesWithReExecute("/{0}");
```

Этот метод расширения настраивает `StatusCodePagesMiddleware` для повторного выполнения конвейера всякий раз, когда обнаруживается код ответа 4xx или 5xx, используя предоставленный путь обработки ошибок. Это похоже на то, как `ExceptionHandlerMiddleware` повторно выполняет конвейер, как показано на рис. 15.6.

Обратите внимание, что путь обработки ошибок "`{0}`" содержит маркер строки форматирования, `{0}`. При повторном выполнении пути компонент заменит этот маркер на номер кода состояния. Например, ошибка 404 приведет к повторному выполнению пути `/404`. Обработчик пути (обычно это страница Razor) имеет доступ к коду состояния и при желании может изменить ответ в зависимости от кода состояния. Вы можете выбрать любой путь обработки ошибок, если ваше приложение знает, как его обработать.

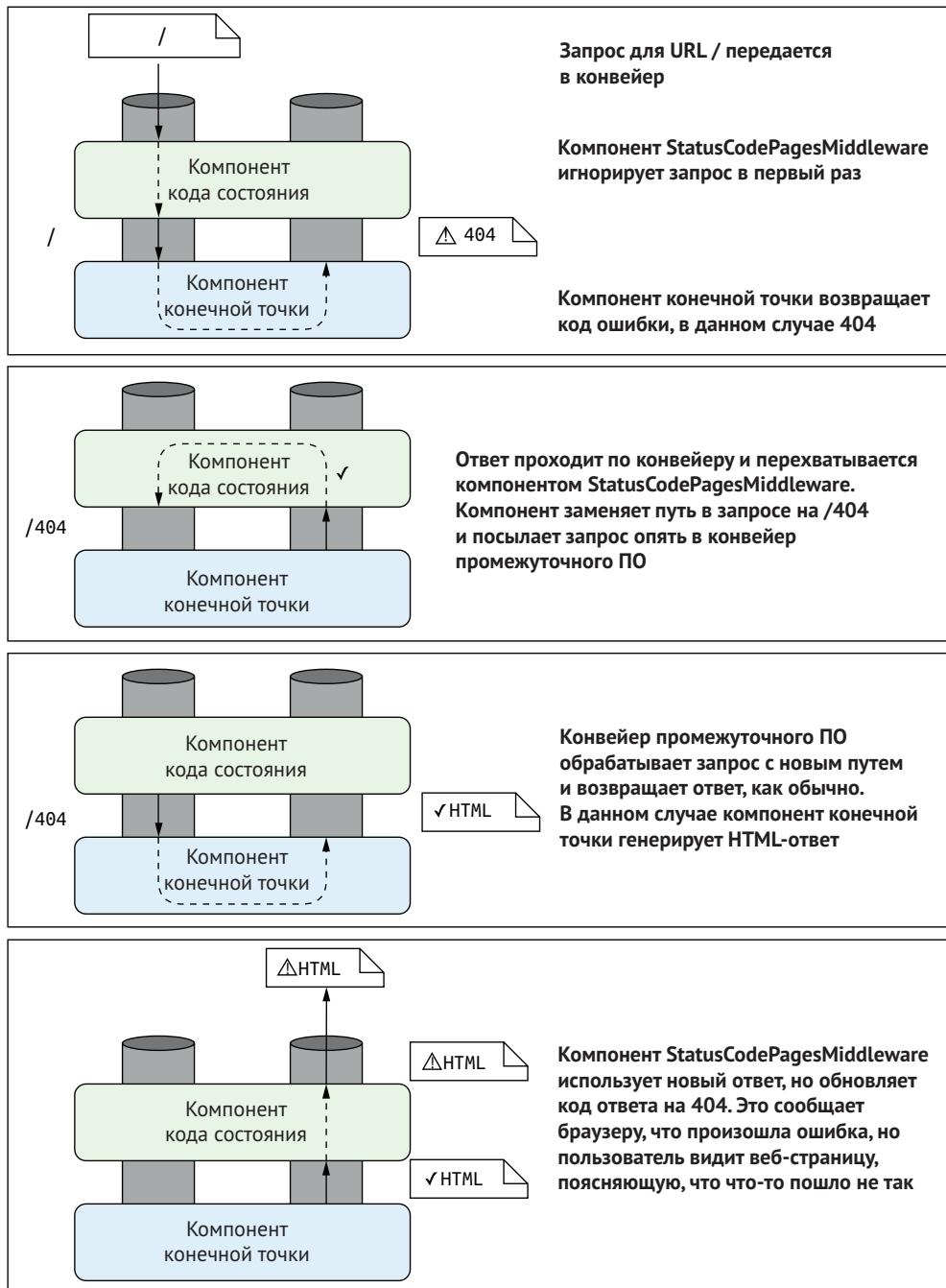


Рис. 15.6 StatusCodePagesMiddleware повторно выполняет конвейер для генерации HTML-тела ответа 404. Запрос к пути / возвращает ответ 404, который обрабатывается компонентом кода состояния. Конвейер повторно запускается с использованием пути /404 для генерации ответа в виде HTML

При таком подходе можно создавать разные страницы для разных кодов ошибок, например страницу с ошибкой 404, показанную на рис. 15.7. Этот метод гарантирует, что ваши страницы ошибок согласованы с остальной частью приложения, включая любое динамически генерируемое содержимое, а также позволяет адаптировать сообщение для распространенных ошибок.

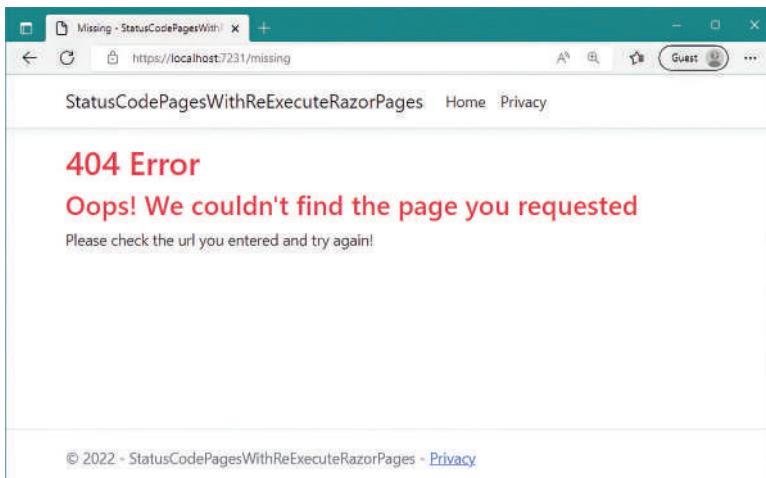


Рис. 15.7 Страница для отсутствующего файла. При обнаружении кода ошибки (в этом случае это ошибка 404) конвейер промежуточного ПО повторно запускается для генерации ответа. Это позволяет динамическим частям вашей веб-страницы оставаться согласованными

ВНИМАНИЕ! Как я упоминал в главе 4, если путь обработки ошибок генерирует ошибку, пользователь увидит стандартную ошибку браузера. Чтобы предотвратить это, часто лучше использовать статическую страницу ошибок, которая всегда будет работать, а не динамическую страницу, которая рискует выдать больше ошибок.

Метод `UseStatusCodePagesWithReExecute()` отлично подходит для возврата удобной страницы с ошибкой, когда в запросе что-то идет не так, но есть второй способ использования `StatusCodePagesMiddleware`. Вместо повторного выполнения конвейера для генерации ответа об ошибке можно перенаправить пользователя на страницу ошибки, вызвав

```
app.UseStatusCodePagesWithRedirects("/{0}");
```

Как и версия с повторным выполнением, этот метод принимает строку формата, которая определяет URL-адрес для генерации ответа. Однако, в то время как версия с повторным выполнением генерирует ответ об ошибке для исходного запроса, версия с перенаправлением изначально возвращает ответ с кодом состояния 302, указывая браузеру отправить второй запрос, на этот раз для URL-адреса ошибки, как показано на рис. 15.8. Этот второй запрос генерирует ответ страницы ошибки, возвращая его с кодом состояния 200.

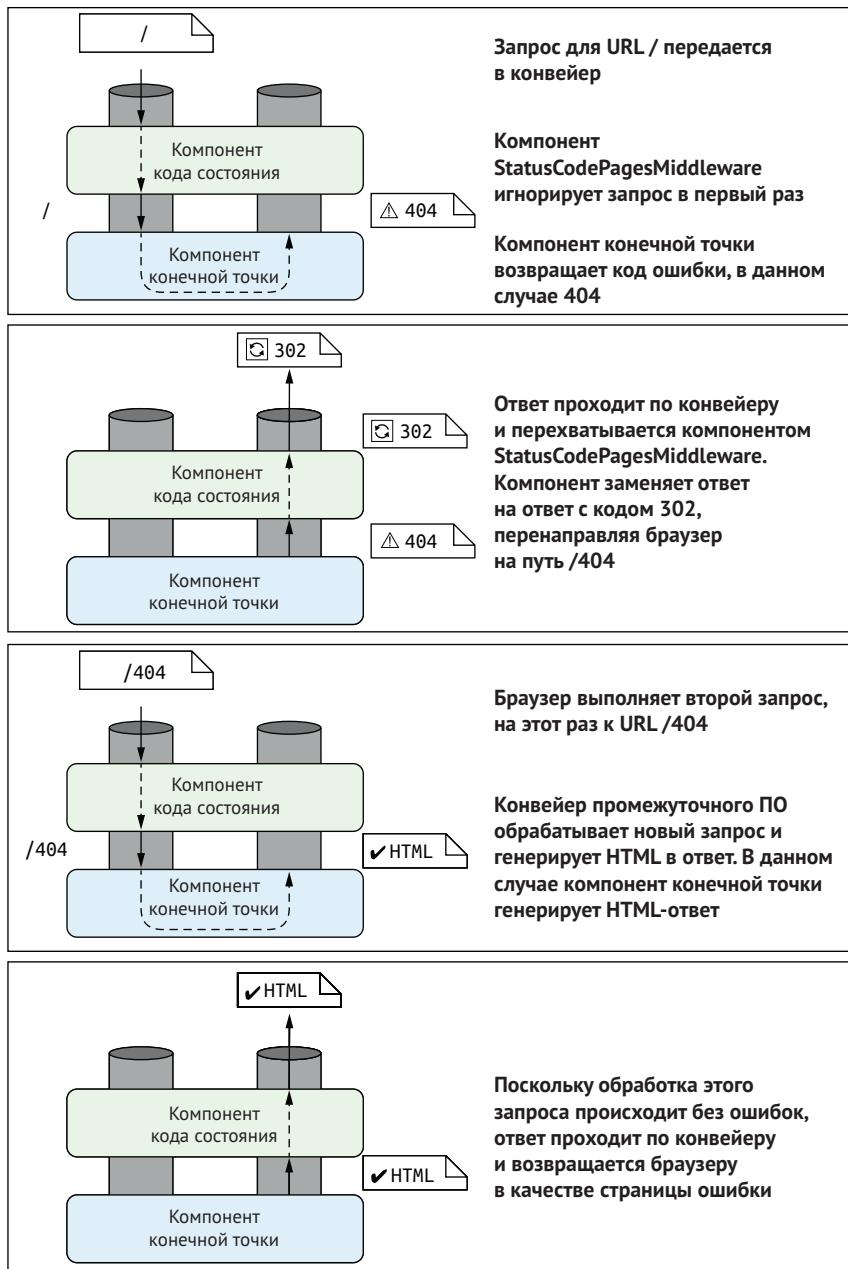


Рис. 15.8 StatusCodePagesMiddleware возвращает перенаправления для создания страниц ошибок. Запрос к пути `/` возвращает ответ `404`, который перехватывается компонентом кода состояния и преобразуется в ответ `302`. Браузер выполняет второй запрос, используя путь `/404`, для генерации ответа в формате `HTML`

Независимо от того, используете ли вы метод повторного выполнения или перенаправления, браузер в конечном итоге получает, по

существу, один и тот же HTML-код. Однако есть некоторые важные различия:

- при повторном выполнении исходный код состояния (например, 404) сохраняется. Браузер воспринимает HTML-код страницы ошибки как ответ на исходный запрос. Если пользователь обновляет страницу, браузер делает второй запрос исходного пути;
- при использовании перенаправления исходный код состояния теряется. Браузер воспринимает перенаправление и второй запрос как два отдельных запроса и не «знает» об ошибке. Если пользователь обновляет страницу, браузер отправляет запрос на тот же путь ошибки; он не отправляет повторно исходный запрос.

В большинстве случаев я считаю, что подход с повторным выполнением более полезен, поскольку он сохраняет исходную ошибку и обычно ведет себя так, как ожидают пользователи. Однако в некоторых случаях метод перенаправления может быть полезен, например когда совершенно другое приложение генерирует страницу с ошибкой.

СОВЕТ Отдавайте предпочтение использованию метода `UseStatusCodePagesWithReExecute`, а не метода перенаправления, когда одно и то же приложение создает HTML-код страницы ошибки для приложения.

Вы можете использовать `StatusCodePagesMiddleware` в сочетании с другим компонентом для обработки исключений, добавив их в конвейер. `StatusCodePagesMiddleware` изменяет ответ только в том случае, если тело ответа не было записано. Поэтому если другой компонент, например `ExceptionHandlerMiddleware`, возвращает тело сообщения вместе с кодом ошибки, оно не будет изменено.

ПРИМЕЧАНИЕ `StatusCodePagesMiddleware` имеет дополнительные перегруженные варианты, которые позволяют запускать настраиваемые компоненты при возникновении ошибки вместо повторного выполнения конвейера промежуточного ПО. Об этом подходе можно прочитать на странице <http://mng.bz/OK66>.

Обработка ошибок важна при разработке любого веб-приложения; ошибки случаются, и вам нужно корректно их обрабатывать. `StatusCodePagesMiddleware` практически необходим для любого рабочего приложения Razor Pages.

В главе 16 мы подробно рассмотрим привязку модели. Вы увидите, как значения маршрута, сгенерированные во время маршрутизации, привязываются к параметрам обработчика страницы и, возможно, что более важно, как проверять предоставленные вам значения.

Резюме

- Обработчик страницы Razor Page – это метод в классе модели страницы, который выполняется, когда страница Razor обрабатывает запрос;

- обработчики страниц должны убедиться, что входящий запрос действителен, вызвать соответствующие сервисы предметной области для обработки запроса, а затем выбрать тип возвращаемого ответа. Обычно они не генерируют ответ напрямую; вместо этого они описывают, как сгенерировать ответ;
- обработчики страниц обычно должны делегировать сервисам обработку бизнес-логики, необходимой для запроса, а не выполнять изменения самостоятельно. Это обеспечивает четкое разделение задач, что облегчает тестирование и улучшает структуру приложения;
- при выполнении страницы Razor вызывается обработчик одной страницы на основе HTTP-метода запроса и значения маршрута обработчика. Если обработчик страницы не найден, вместо него используется «неявный» обработчик, просто отображающий содержимое страницы Razor;
- обработчики страниц могут иметь параметры, значения которых берутся из свойств входящего запроса в ходе процесса, называемого *привязкой модели*. Свойства, декорированные атрибутом `[BindProperty]`, также могут быть привязаны к запросу. Это канонические способы чтения значений из HTTP-запроса на странице Razor;
- по умолчанию свойства, декорированные атрибутом `[BindProperty]`, не привязываются к запросам с методом GET. Чтобы активировать привязку, используйте `[BindProperty(SupportsGet = true)]`;
- обработчики страниц могут возвращать `PageRoute` или `void` для генерации ответа в формате HTML. Инфраструктура страниц Razor использует связное представление Razor для создания HTML и возвращает ответ `200 OK`;
- вы можете отправить пользователей на другую страницу Razor, используя `RedirectToPageResult`. Обычно пользователи отправляются на новую страницу в рамках потока POST-REDIRECT-GET для обработки пользовательского ввода через формы;
- базовый класс `PageModel` предоставляет множество вспомогательных методов для создания `IActionResult`, например `Page()`, который создает `PageRoute`, и `RedirectToPage()`, который создает `RedirectToPageResult`. Эти методы представляют собой простые обертки для вызова `new` соответствующего типа `IActionResult`;
- `StatusCodesMiddleware` позволяет предоставлять удобные сообщения об ошибках, когда конвейер возвращает необработанный код состояния ответа на ошибку. Это важно для обеспечения единобразного взаимодействия с пользователем при возврате кода состояния ошибок, например ошибок 404, когда URL-адрес не соответствует конечной точке.

16

Привязка и валидация запросов с помощью Razor Pages

В этой главе:

- использование значений запроса для создания моделей привязки;
- настройка процесса привязки модели;
- валидация пользовательского ввода с помощью атрибутов DataAnnotations.

В главе 7 мы рассмотрели процесс привязки и валидации модели в минимальных API. В этой главе мы рассмотрим эквивалент этого в Razor Pages: извлечение значений из запроса с использованием привязки модели и проверки пользовательского ввода.

В первой половине этой главы мы рассмотрим применение моделей привязки для получения этих параметров из запроса, чтобы можно было использовать их на страницах Razor, создавая объекты C#. Эти объекты передаются обработчикам страницы Razor в качестве параметров метода или задаются как свойства в модели (`PageModel`) страницы Razor.

Когда код выполняется в методе обработчика страниц, нельзя просто использовать модель привязки без каких-либо дополнительных размышлений. Каждый раз, когда вы используете данные, предоставленные

ные пользователем, необходимо их проверять! Вторая половина главы посвящена валидации моделей привязки с помощью Razor Pages.

Мы рассмотрели привязку моделей и валидацию в минимальных API в главе 7, и с концептуальной точки зрения привязка и валидация в Razor Pages аналогичны. Однако их детали и механизмы совершенно разные.

Модели привязки, заполняемые инфраструктурой Razor Pages, передаются обработчикам страниц при выполнении. После запуска обработчика страницы все готово для использования моделей вывода в реализации «модель–представление–контроллер» (MVC): модели представления и модели API. Они используются для генерации ответа на запрос пользователя. Мы рассмотрим их в главах 19 и 20.

Прежде чем продолжить, вспомним паттерн проектирования MVC и то, как модели привязки вписываются в ASP.NET Core.

16.1 Модели в Razor Pages и MVC

В этом разделе описано, как модели привязки вписываются в паттерн проектирования MVC, который мы рассматривали в главе 13. Я описы-ваю разницу между моделями привязки и другими концепциями «моде-ли» в паттерне MVC и то, как каждая из них используется в ASP.NET Core.

MVC – это разделение ответственности. Смысл в том, что, изолируя каж-дый аспект приложения, чтобы сосредоточиться на единственной ответ-ственности, он уменьшает взаимозависимости в системе. Такое разделение упрощает внесение изменений, не влияя на другие части приложения.

Классический паттерн проектирования MVC состоит из трех неза-висимых компонентов:

- *модель* – отображаемые данные и методы обновления;
- *представление* – отображает представление данных, составляю-щих модель;
- *контроллер* – вызывает методы модели и выбирает представление.

В этом паттерне есть только одна модель, модель приложения, ко-торая представляет всю бизнес-логику приложения, а также способы обновления и изменения его внутреннего состояния. В ASP.NET Core есть несколько моделей, в которых принцип единственной ответ-ственности продвинулся на шаг вперед, по сравнению с некоторыми представлениями MVC.

В главе 13 мы рассмотрели пример приложения со списком дел, ко-торое может отображать все задачи для данной категории и имени поль-зователя. С помощью этого приложения вы отправляете запрос на URL-адрес, который маршрутизируется, используя шаблон `todo/listcategory/{category}/{username}`. После этого вы получаете ответ, по-казывающий все соответствующие задачи, как видно на рис. 16.1.

В приложении используются те же конструкции MVC, которые вы уже видели, например маршрутизация к обработчику страницы Razor, а также ряд различных моделей. На рис. 16.2 показано, как запрос к этому приложению сопоставляется с паттерном проектирования MVC и как он генерирует окончательный ответ, включая дополнительные сведения о привязке модели и проверке (валидации) запроса.

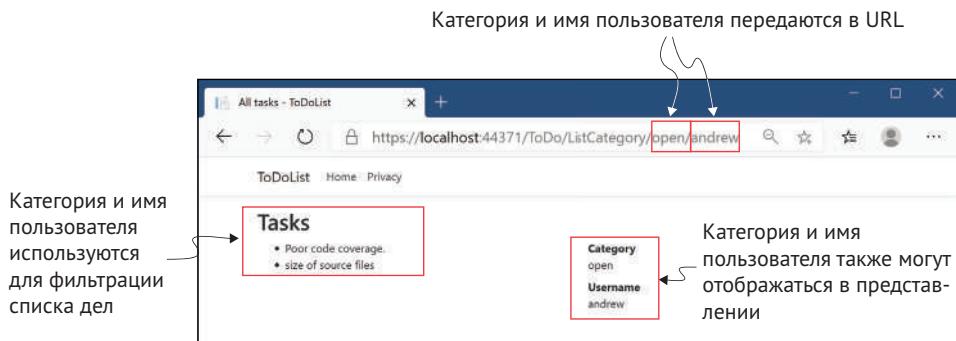


Рис. 16.1 Базовое приложение со списком дел, которое отображает элементы списка дел. Пользователь может фильтровать список элементов, изменив параметры category и username в URL-адресе

Razor Pages использует несколько разных моделей, большинство из которых являются объектами POCO, и модель приложения, которая скорее представляет собой концепцию набора сервисов. Каждая из моделей в ASP.NET Core отвечает за обработку разных аспектов общего запроса:

- **модель привязки** – это вся информация, предоставляемая пользователем при выполнении запроса, а также дополнительные контекстные данные. Сюда входят такие вещи, как параметры маршрута, извлеченные из URL-адреса, строка запроса и данные формы или данные в формате JSON в теле запроса. Сама модель привязки – это один или несколько определяемых вами объектов POCO. Модели привязки в Razor Pages обычно определяются путем создания публичного свойства в модели страницы и декорирования его с помощью атрибута `[BindProperty]`. Их также можно передать обработчику страницы в качестве параметров.

В этом примере модель привязки будет включать имя категории `open` и имя пользователя `andrew`. Инфраструктура Razor Pages проверяет модель привязки перед выполнением обработчика страницы, чтобы проверить, действительны ли предоставленные значения, хотя обработчик страницы будет выполняться, даже если они таковыми не являются. Вы увидите это, когда мы будем обсуждать валидацию модели в разделе 16.3;

- **модель приложения** – на самом деле модель приложения не является моделью ASP.NET Core. Как правило, это скорее концепция, целая группа различных сервисов и классов – все, что необходимо для выполнения какого-либо бизнес-действия в приложении. Она может включать в себя как модель предметной области (которая представляет то, что ваше приложение пытается описать) и модели базы данных (которые представляют данные, хранящиеся в базе данных), так и любые другие, дополнительные сервисы. В приложении со списком дел модель приложения будет содержать полный список дел, который, вероятно, хранится в базе данных, и она знает, как найти только задачи в категории `open`, назначенные `Andrew`;



Рис. 16.2 Паттерн MVC в ASP.NET Core, обрабатывающий запрос на просмотр подмножества элементов в приложении Razor Pages

■ **модель страницы** – `PageModel` страницы Razor выполняет две основные функции: она действует как контроллер для приложения, предоставляя методы обработчика страницы, и как модель представления для представления Razor. Все данные, необходимые представлению для генерации ответа, представлены в модели страницы, например список дел в категории `open`, назначенной `andrew`.

Базовый класс `PageModel`, от которого вы наследуете свои страницы Razor, содержит различные вспомогательные свойства и методы. Одно из них, свойство `ModelState`, содержит результат валидации модели в виде серии пар «ключ–значение». Подробнее о валидации и свойстве `ModelState` вы узнаете в разделе 16.3.

Эти модели составляют основу любого приложения Razor Pages, обрабатывая ввод, бизнес-логику и вывод каждого обработчика стра-

ницы. Представьте, что у вас есть приложение для онлайн-торговли, позволяющее пользователям искать одежду, отправляя запросы на URL-адрес `/search/{query}`, где `{query}` содержит поисковый запрос:

- *модель привязки* – принимает параметр маршрута `{query}` из URL-адреса и любые значения, присланные в теле запроса (возможно, порядок сортировки или число элементов для показа), и привязывает их к классу C#, который обычно действует как класс передачи данных. Объект этого класса передачи данных будет доступен как свойство модели страницы при вызове обработчика страницы;
- *модель приложения* – это сервисы и классы, выполняющие логику. При вызове обработчиком страницы эта модель загружает всю одежду, соответствующую запросу, применяя необходимую сортировку и фильтры, и возвращает результаты контроллеру;
- *модель страницы* – значения, предоставляемые моделью приложения, будут заданы как свойства `PageModel` страницы Razor вместе с другими метаданными, такими как общее количество доступных элементов или возможность для пользователя оформить заказ в данный момент. Представление Razor будет использовать эти данные для визуализации представления Razor в HTML-код.

Важный момент всех этих моделей состоит в том, что их обязанности четко определены и разделены. Их разделение и предотвращение повторного использования помогают обеспечить гибкость и простоту обновления приложения.

Очевидным исключением из этого разделения является модель страницы, поскольку именно здесь определяются модели привязки и обработчики страниц, а также хранятся данные, необходимые для визуализации представления. Некоторые могут посчитать очевидное отсутствие разделения кощунством, но на самом деле это не проблема. Демаркационные линии вполне очевидны. Если вы, например, не пытаетесь вызвать обработчик страницы из представления Razor, то у вас не должно возникнуть никаких проблем!

Теперь, когда вы должным образом познакомились с различными моделями в ASP.NET Core, пора сосредоточиться на том, как их использовать. В этой главе рассматриваются модели привязки, которые строятся из входящих запросов, – как они создаются и откуда берутся значения?

16.2 От запроса к модели: делаем запрос полезным

В этом разделе вы узнаете:

- как ASP.NET Core создает модели привязки из запроса;
- как привязать простые типы, такие как `int` и `string`, а также сложные классы;
- как выбрать, какие части запроса используются в модели привязки.

К настоящему времени вы должны быть знакомы с тем, как ASP.NET Core обрабатывает запрос, выполняя обработчик страницы для стра-

ницы Razor. Обработчики страниц – это обычные методы C#, поэтому у фреймворка ASP.NET Core должна быть возможность вызывать их обычным способом. Процесс извлечения значений из запроса и создания из них объектов C# называется *привязкой модели*.

Любые общедоступные свойства в модели страницы Razor (в файле .cshtml.cs страницы Razor), декорированные атрибутом [BindProperty], создаются из входящего запроса с использованием привязки модели, как показано в листинге 16.1.

Аналогично, если у метода обработчика страницы имеются какие-либо параметры, они также создаются с использованием привязки модели.

ВНИМАНИЕ! Свойства, декорированные атрибутом [BindProperty], должны иметь публичный метод set; в противном случае привязка завершится ошибкой.

Листинг 16.1. Запросы привязки модели к свойствам на странице Razor

```
public class IndexModel : PageModel
{
    [BindProperty]
    public string Category { get; set; } | Свойства, декорированные
                                            атрибутом [BindProperty],
                                            участвуют в привязке модели

    [BindProperty(SupportsGet = true)]
    public string Username { get; set; } | Свойства не привязываются
                                            к модели для запросов GET, если
                                            вы не используете SupportsGet

    public void OnGet()
    {
    }

    public void OnPost(ProductModel model) <--| Параметры обработчи-
                                                ков страниц также привя-
                                                зываются к модели, если
                                                этот обработчик выбран
    {
    }
}
```

Как показано в главе 15 и предыдущем листинге, свойства модели страницы *не* привязываются к модели для запросов методом GET, даже если вы добавите атрибут [BindProperty].

По соображениям безопасности привязываются только запросы, использующие такие методы, как POST и PUT. Если вы хотите привязать запросы с методом GET, то можете задать свойство SupportsGet в атрибуте [BindProperty], чтобы разрешить привязку модели.

Какая часть является моделью привязки?

В листинге 16.1 показана страница Razor, использующая несколько моделей привязки: свойства Category, Username и ProductModel (в обработчике OnPost) привязываются к модели.

Использование нескольких моделей таким образом – это нормально, но я предпочитаю подход, при котором все привязки модели хранятся

в одном вложенном классе, который я часто называю `InputModel`. При таком подходе страницу Razor из листинга 16.1 можно было бы написать следующим образом:

```
public class IndexModel : PageModel
{
    [BindProperty]
    public InputModel Input { get; set; }
    public void OnGet()
    {
    }

    public class InputModel
    {
        public string Category { get; set; }
        public string Username { get; set; }
        public ProductModel Model { get; set; }
    }
}
```

У такого подхода есть некоторые организационные преимущества, о которых вы узнаете подробнее в разделе 16.4.

ASP.NET Core автоматически заполняет модели привязки за вас, используя свойства запроса, такие как URL-адрес запроса, любые заголовки, отправленные в HTTP-запросе, данные, явно отправленные с помощью метода POST в теле запроса, и т. д.

ПРИМЕЧАНИЕ В этой главе описано, как привязывать модели к входящему запросу, но не показано, как Razor Pages использует модели привязки для генерации этого запроса с помощью HTML-форм. В главе 17 вы узнаете о синтаксисе Razor, который отображает HTML-код, а в главе 18 познакомитесь с тег-хелперами Razor, которые генерируют поля формы на основе модели привязки.

По умолчанию ASP.NET Core использует три разных источника привязки при создании модели привязки. Он просматривает каждый из них по порядку и принимает первое значение, которое находит (если оно есть), соответствующее имени модели:

- *значения формы* – отправляются в теле HTTP-запроса, когда форма отправляется на сервер, с помощью метода POST;
- *значения маршрута* – получаются из сегментов URL-адреса или значений по умолчанию после сопоставления маршрута, как было показано в главе 14;
- *значения строки запроса* – передаются в конце URL-адреса и не используются во время маршрутизации.

ВНИМАНИЕ! Несмотря на концептуальную схожесть, процесс привязки Razor Page работает совершенно иначе, чем подход, используемый минимальными API.

Процесс привязки модели показан на рис. 16.3. Связыватель модели проверяет каждый источник привязки, чтобы проверить, содержит ли он значение, которое можно задать в модели. В качестве альтернативы модель также может выбрать конкретный источник, из которого должно поступать значение. Вы увидите это в разделе 16.2.3. После привязки каждого свойства модель валидируется и присваивается в свойство объекта `PageModel` или передается в качестве параметра обработчику страницы. С процессом валидации модели вы познакомитесь во второй половине этой главы.

ПРИМЕЧАНИЕ В Razor Pages разные свойства сложной модели можно привязывать к разным источникам. Это отличается от минимальных API, где весь объект будет привязан из одного источника и «частичная» привязка невозможна. Страницы Razor также по умолчанию привязываются к телам форм, тогда как минимальные API не могут этого сделать. Отчасти эти различия объясняются историческими причинами, а отчасти это связано с тем, что в такой ситуации минимальные API отдают предпочтение производительности, а не удобству.

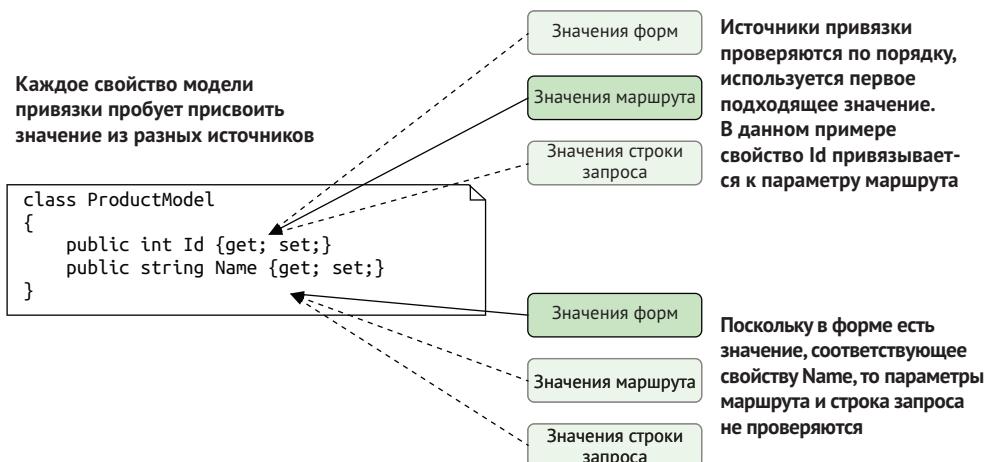


Рис. 16.3 Привязка модели включает в себя присваивание значений из источников привязки, которые соответствуют разным частям запроса

Свойства `PageModel` или параметры обработчика страницы?

Привязку модели в Razor Pages можно использовать тремя способами:

- декорировать свойства модели страницы атрибутом `[BindProperty]`;
- добавить параметры в свой метод обработчика страницы;
- декорировать всю модель страницы атрибутом `[BindProperties]`.

Какой из этих подходов выбрать?

Ответ на этот вопрос – во многом дело вкуса. Задание свойств в модели страницы и пометка их атрибутом `[BindProperty]` – это подход,

который вы чаще всего будете встречать в примерах. Если вы воспользуетесь им, то сможете получить доступ к модели привязки при визуализации представления, как показано в главах 17 и 18.

Альтернативный подход, при котором к методам обработчика страницы добавляются параметры, обеспечивает больше разделения между различными этапами MVC, поскольку вы не сможете получить доступ к параметрам за пределами обработчика страницы. С другой стороны, если вам действительно нужно отобразить эти значения в представлении Razor, вам придется вручную скопировать параметры в свойства, к которым можно получить доступ в представлении.

Я избегаю последнего подхода, декорируя модель страницы атрибутом `[BindProperties]`. При таком варианте каждое свойство `PageModel` участвует в привязке модели. Мне не нравится косвенность, которую это дает, и существует риск случайного связывания свойств, которые я не хотел привязывать к модели.

Подход, который выбираю я, обычно зависит от конкретной страницы Razor, которую я создаю. Если я создаю форму, то выберу подход, использующий атрибут `[BindProperty]`, поскольку обычно мне нужен доступ к значениям запроса внутри представления Razor. Для простых страниц, где привязка модели – это идентификатор продукта, я предпочитаю подход с параметрами обработчика страницы из-за его простоты, особенно если обработчик предназначен для запроса методом GET. Более конкретные советы по своему подходу я даю в разделе 16.4.

На рис. 16.4 показан пример запроса, создающего аргумент метода `ProductModel`, используя привязку модели для примера, показанного в начале этого раздела:

```
public void OnPost(ProductModel product)
```

На рис. 16.4 показан пример запроса, создающего аргумент метода `ProductModel`, используя привязку модели для примера, показанного в начале этого раздела:

```
public void OnPost(ProductModel product)
```

Во время привязки модели создается новый объект `ProductModel`, используя значения, полученные в запросе: значения маршрута из URL и данные из тела запроса

Locals	Search (Ctrl+F)	Search Depth: 3	A
Name	Value	Type	
this	(ExampleBinding.Pages.EditProductModel)	ExampleBinding.Pages.EditProductModel	
product	(ExampleBinding.ProductModel)	ExampleBinding.ProductModel	
Id	5	int	
Name	"This is my name"	string	
SellPrice	25.43	decimal	

HTTP-запрос получен и направлен маршрутизацией в страницу `EditProduct`. Свойство типа `ProductModel` декорировано атрибутом `[BindProperty]`

Рис. 16.4 Применение привязки модели для создания экземпляра модели, которая используется для выполнения страницы Razor

Свойство `Id` было привязано из параметра маршрута URL, а свойства `Name` и `SellPrice` – из тела запроса. Существенное преимущество использования привязки модели заключается в том, что вам не нужно писать код для парсинга запросов и самостоятельно сопоставлять данные. Такой код обычно является стереотипным и подвержен ошибкам, поэтому использование встроенного традиционного подхода позволяет сосредоточить внимание на важных аспектах приложения: бизнес-требованиях.

СОВЕТ Привязка модели отлично подходит для уменьшения стереотипного кода. Используйте ее всегда, когда это возможно, и вам редко придется обращаться к объекту `Request` напрямую.

Если нужно, то есть возможности, позволяющие полностью настроить способ работы привязки модели, но вам довольно редко придется заходить так далеко. В большинстве случаев все работает как есть, и вы увидите это в оставшейся части раздела.

16.2.1 Связывание простых типов

Мы начнем наше путешествие по привязке модели с рассмотрения простого обработчика страницы Razor. В следующем листинге показана простая страница Razor, в которой в качестве параметра метода берется одно число и возводится в квадрат.

Листинг 16.2 Страница Razor, принимающая простой параметр

```
public class CalculateSquareModel : PageModel
{
    public void OnGet(int number) {
        Square = number * number;
    }

    public int Square { get; set; }
}
```

Параметр метода – это модель привязки

В более сложном примере эта работа будет выполняться во внешнем сервисе, в модели приложения

Результат предоставляется как свойство и используется представлением для генерации ответа

В главах 6 и 14 вы узнали, что такая маршрутизация и как она выбирает, какую страницу Razor выполнить. Вы можете обновить шаблон маршрута для страницы Razor на `"CalculateSquare/{number}"`, добавив сегмент `{number}` к директиве `@page` страницы Razor в файле с расширением `.cshtml`:

```
@page "{number}"
```

Когда клиент запрашивает URL-адрес `/CalculateSquare/5`, Razor Page использует маршрутизацию для парсинга параметров маршрута. В результате получается пара значений маршрута:

```
number=5
```

Обработчик страницы `OnGet` страницы Razor содержит единственный параметр – целое число `number`, – который является моделью привязки. Когда ASP.NET Core будет выполнять этот метод обработчика

страницы, то обнаружит ожидаемый параметр. Он просмотрит значения маршрута, ассоциированные с запросом, и найдет пару `number=5`. Затем он может привязать параметр `number` к этому значению маршрута и выполнить метод. Сам метод обработчика страницы не заботится о том, откуда взялось это значение; он идет своим путем, вычисляя квадрат значения и задавая его свойству `Square`.

Главное, что необходимо понять, – вам не нужно было писать дополнительный код, чтобы пытаться извлечь число из URL-адреса при выполнении метода. Все, что нужно было сделать, – создать параметр метода (или открытое свойство) с правильным именем и позволить привязке модели творить свою магию.

Значения маршрута – не единственные значения, которые связыватель модели может использовать для создания привязки модели. Как вы уже видели ранее, фреймворк будет просматривать три источника привязки по умолчанию, чтобы найти соответствие моделям привязки:

- значения формы;
- значения маршрута;
- значения строки запроса.

Каждый из этих источников привязки содержит значения в виде пары «имя–значение». Если ни один из источников привязки не содержит нужного значения, для модели привязки по умолчанию создается новый экземпляр типа. Точное значение модели привязки в этом случае зависит от типа переменной:

- для типов значений будет использоваться значение `default(T)`. Для параметра `int` это будет `0`, а для `bool` – `false`;
- для ссылочных типов тип создается с помощью конструктора по умолчанию (без параметров). Для пользовательских типов, таких как `ProductModel`, будет создан новый объект. Для типов, допускающих значение `null`, таких как `int?` или `bool?`, значение будет `null`;
- для строковых типов значение будет `null`.

ВНИМАНИЕ! Важно учитывать поведение обработчика страницы, когда привязка модели не может связать параметры метода. Если ни один из источников привязки не содержит значение, то значение, передаваемое методу, может быть `null` или неожиданно иметь значение по умолчанию (для типов значений).

В листинге 16.2 показано, как привязать один-единственный параметр метода. Сделаем следующий логический шаг и посмотрим, как привязать несколько параметров.

В предыдущей главе мы обсуждали маршрутизацию для создания приложения для конвертера валют.

Допустим, вы создаете приложение для конвертации валют. В качестве первого шага необходимо создать метод, в котором пользователь предоставляет значение в одной валюте, а вы должны конвертировать его в другую. Сначала вы создаете страницу Razor под названием `Convert.cshtml`, а потом настраиваете шаблон маршрута для

страницы с помощью директивы `@page`, чтобы использовать абсолютный путь, содержащий два значения маршрута:

```
@page "/{currencyIn}/{currencyOut}"
```

Затем вы создаете обработчик страницы, который принимает три необходимых вам значения, как показано в следующем листинге.

Листинг 16.3. Обработчик страницы Razor, принимающий несколько параметров привязки

```
public class ConvertModel : PageModel
{
    public void OnGet(
        string currencyIn,
        string currencyOut,
        int qty
    )
    {
        /* Реализация метода; */
    }
}
```

Как видите, здесь есть три разных параметра для привязки. Вопрос в том, откуда берутся значения и как они будут заданы. Ответ: все зависит от обстоятельств! В табл. 16.1 показано множество возможных вариантов. Во всех этих примерах используется один и тот же шаблон маршрута и обработчик страницы, но в зависимости от отправленных данных будут привязаны разные значения. Фактические значения могут отличаться от ожидаемых, поскольку доступные источники привязки предлагают противоречивые значения!

URL (значения маршрута)	Данные тела HTTP (значения формы)	Привязанные значения параметров
/GBP/USD		currencyIn=GBP currencyOut=USD qty=0
/GBP/USD?currencyIn=CAD	QTY=50	currencyIn=GBP currencyOut=USD qty=50
/GBP/USD?qty=100	qty=50	currencyIn=GBP currencyOut=USD qty=50
/GBP/USD?qty=100	currencyIn=CAD& currencyOut=EUR&qty=50	currencyIn=CAD currencyOut=EUR qty=50

Для каждого примера убедитесь, что вы понимаете, *почему* связанные значения имеют те значения, которые у них есть. В первом примере значения `qty` нет в данных формы, значении маршрута или в строке запроса, поэтому он имеет значение по умолчанию 0. В других примерах запрос содержит одно или несколько повторяющихся значений; в этих случаях важно помнить о порядке, в котором связыватель модели проверяет источники привязки. По умолчанию значе-

ния формы имеют приоритет над другими источниками привязки, включая значения маршрута!

ПРИМЕЧАНИЕ По умолчанию связыватель модели не чувствителен к регистру, поэтому значение QTY=50 будет успешно привязано к параметру qty.

Хотя это может показаться немного сложным, относительно необычно связывать все эти источники сразу. Чаще всего все ваши значения поступают из тела запроса в виде значений формы, возможно, с идентификатором из значений маршрута URL. Данный сценарий служит скорее поучительным рассказом о том, какое странное поведение вы можете встретить, если не уверены, как все работает под капотом.

В этих примерах вы успешно связали целочисленное свойство qty с входящими значениями, но, как я упоминал ранее, все значения, хранящиеся в источниках привязки, являются строками. В какие типы можно преобразовать строку?

Связыватель модели преобразует практически любой примитивный тип .NET, например int, float, decimal (и, очевидно, string), любой пользовательский тип, у которого есть метод TryParse (как в минимальных API, как вы видели в главе 7), а также все, что имеет преобразователь типов TypeConverter.

ПРИМЕЧАНИЕ TypeConverter можно найти в пакете System.ComponentModel.TypeConverter. Подробнее о них можно прочитать в документации Microsoft «Преобразование типов в .NET»: <http://mng.bz/AOGK>.

Есть несколько других особых случаев, когда можно выполнить преобразование из строки, например Type, но, рассматривая их только как примитивы, вы далеко продвинетесь!

16.2.2 Связывание сложных типов

Если кажется, что возможность привязывать только простые примитивные типы несколько ограничивает вас, то вы правы! К счастью, это не относится к связывателю моделей. Хотя он может только преобразовывать строки непосредственно в эти примитивные типы, он также может связывать сложные типы, просматривая любые свойства, предоставляемые моделями привязки, и привязывая каждое из этих свойств к строкам.

Если вас это не впечатлило, тогда посмотрим, как вам пришлось бы создавать обработчики страниц, если бы простые типы были вашим единственным вариантом. Представьте себе, что пользователь приложения конвертера валют дошел до страницы оформления заказа и собирается обменять валюту. Здорово! Все, что вам нужно сейчас, – это взять его имя, адрес электронной почты и номер телефона. К сожалению, ваш метод обработчика страницы должен выглядеть как-то так:

```
public IActionResult OnPost(
    string firstName, string lastName,
    string phoneNumber, string email)
```

Фу! Хотя может показаться, что четыре параметра – это не так уж плохо, но что произойдет, когда требования изменятся и вам понадобятся другие сведения? Сигнатура метода продолжит расти. Привязка модели довольно успешно привязывает значения, но это не совсем чистый код. Использование атрибута [BindProperty] тоже не спасает – вам все равно придется загромождать модель страницы множеством свойств и атрибутов!

Упрощение параметров метода привязкой к сложным объектам

Обычным паттерном для любого кода C#, когда у вас много параметров метода, является извлечение класса, который инкапсулирует данные, необходимые методу. Если необходимо добавить дополнительные параметры, то можно добавить в этот класс новое свойство. Класс станет вашей моделью привязки, которая может выглядеть примерно так.

Листинг 16.4 Модель привязки для сбора сведений о пользователе

```
public class UserBindingModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string PhoneNumber { get; set; }
}
```

ПРИМЕЧАНИЕ В этой книге я в основном использую классы вместо записей (record) для своих моделей привязки, но вы можете использовать записи, если хотите. Я считаю, что краткость, которую обеспечивает позиционный синтаксис записей, теряется, если вы хотите добавить атрибуты к свойствам, например атрибуты валидации, как вы увидите в разделе 16.3. Необходимый синтаксис для атрибутов позиционных свойств можно увидеть в документации на странице <http://mng.bz/KexO>.

С помощью этой модели вы теперь можете обновить сигнатуру метода обработчика страницы:

```
public IActionResult OnPost(UserBindingModel user)
```

Или, в качестве альтернативы, используя атрибут [BindProperty], создайте свойство для модели страницы:

```
[BindProperty]
public UserBindingModel User { get; set; }
```

Теперь можно еще больше упростить сигнатуру обработчика страницы:

```
public IActionResult OnPost()
```

Функционально связыватель модели несколько иначе трактует этот новый сложный тип. Вместо того чтобы искать параметры со значением, которое соответствует имени параметра (`user` или `User` для свойства), он создает новый экземпляр модели, используя `new UserBindingModel()`.

ПРИМЕЧАНИЕ Не обязательно использовать отдельные классы для своих методов; все зависит от ваших требований. Если обработчику страницы требуется только одно целое число, то имеет смысл выполнить привязку к простому параметру.

После этого связыватель модели перебирает все свойства модели привязки, например `FirstName` и `LastName` в листинге 16.4. Для каждого из этих свойств он обращается к коллекции источников привязки и пытается найти совпадающую пару «имя–значение». Если он ее находит, то задает значение свойства и идет дальше.

СОВЕТ Хотя имя модели в этом примере не является обязательным, связыватель модели также будет искать свойства с префиксом имени свойства, такие как `user.FirstName` и `user.LastName` для свойства `User`. Вы можете использовать этот подход, когда у вас есть несколько сложных параметров для обработчика страницы или несколько сложных свойств `[BindProperty]`. В целом для простоты по возможности следует избегать такой ситуации. Как и для остальных случаев привязки моделей, регистр префикса значения не имеет.

После того как все свойства, которые можно привязать, будут заданы, модель передается обработчику страницы (или задается свойство `[BindProperty]`), а обработчик выполняется как обычно. С этого момента поведение идентично ситуации, когда у вас много отдельных параметров, – вы получите те же значения, которые были заданы в модели привязки, но код будет чище, и с ним будет легче работать.

СОВЕТ Чтобы класс был привязан к модели, он должен иметь открытый конструктор по умолчанию. Привязывать можно только те свойства, которые являются открытыми и доступными для записи.

С помощью этой техники можно привязать сложные иерархические модели, свойства которых сами по себе являются сложными моделями. Пока каждое свойство предоставляет тип, который может быть привязан к модели, связыватель может с легкостью привязать его.

Привязка коллекций и словарей

Помимо обычных пользовательских классов и примитивов, можно выполнять привязку к коллекциям, спискам и словарям. Представьте, что у вас есть страница, на которой пользователь выбрал все валюты, которые его интересуют; их котировки можно отобразить так, как показано на рис. 16.5.

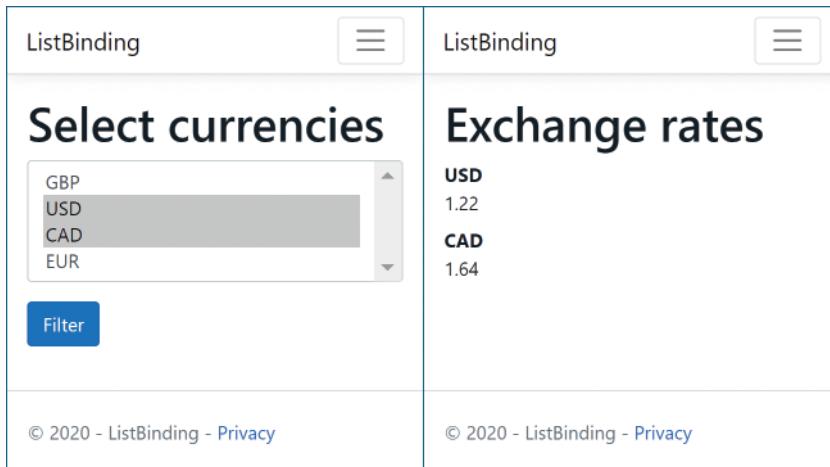


Рис. 16.5 Список выбора валют в приложении конвертера отправит список выбранных валют в приложение. Привязка модели может привязать выбранные валюты и настроить представление, чтобы пользователь видел эквивалентный курс

Для этого можно создать обработчик страницы, который принимает тип `List<string>`:

```
public void OnPost(List<string> currencies);
```

Затем можно отправить данные POST в этот метод, предоставив значения в нескольких разных форматах:

- `currencies[index]`, где `currencies` – это имя параметра для привязки, а `index` – индекс элемента для привязки, например `currencies[0]=GBP¤cies[1]=USD`;
- `[index]` – если вы выполняете привязку только к одному списку (как в этом примере), то имя параметра можно не указывать, например `[0]=GBP&[1]=USD`;
- `currencies` – вы также можете опустить `index` и отправить `currencies` в качестве ключа для каждого значения, например `currencies=GBP¤cies=USD`.

Значения ключей могут поступать из значений маршрута и запроса, но гораздо чаще их отправляют через форму, используя метод POST. Словари могут использовать аналогичную привязку, где ключ словаря заменяет индекс как при именовании параметра, так и при его опущении.

COBET В предыдущем примере я показал коллекцию, используя встроенный тип `string`, но вы также можете привязывать коллекции сложного типа, например `List<UserBindingModel>`.

Если все это кажется немного запутанным, не волнуйтесь. Если вы создаете традиционное веб-приложение и используете представления

Razor для генерации HTML-кода, фреймворк позаботится о создании правильных имен за вас. Как вы увидите в главе 18, представление Razor гарантирует, что любые данные формы, которые вы отправляете, будут сгенерированы в правильном формате.

Привязка загрузки файлов с помощью IFormFile

Razor Pages поддерживает отправку файлов пользователями, предоставляя интерфейсы `IFormFile` и `IFormFileCollection`. Вы можете использовать эти интерфейсы в качестве модели привязки, либо в качестве параметра метода обработчика страницы, либо с помощью подхода с атрибутом `[BindProperty]`, и они будут заполнены подробностями загрузки файла:

```
public void OnPost(IFormFile file);
```

Если нужно принять несколько файлов, то можно использовать `IFormFileCollection`, `IEnumerable<IFormFile>` или `List<IFormFile>`:

Вы уже узнали, как использовать `IFormFile`, в главе 7, когда мы рассматривали привязку минимальных API. Что касается Razor Pages, то здесь процесс тот же. Повторю один момент: если вам не нужно, чтобы пользователи загружали файлы, отлично! При работе с файлами существует так много потенциальных угроз – от вредоносных атак до случайных уязвимостей типа «отказ в обслуживании», – что я избегаю этого, когда это возможно.

Для подавляющего большинства страниц Razor конфигурация привязки модели по умолчанию для простых и сложных типов работает отлично, но вы можете столкнуться с ситуациями, когда вам потребуется больше контроля. К счастью, это вполне возможно, и при необходимости можно полностью переопределить процесс, заменив используемые внутри фреймворка связыватели модели.

Однако такой уровень настройки требуется редко – я обнаружил, что вместо этого чаще всего нужно указать, какой источник привязки следует использовать для модели привязки страницы.

16.2.3 Выбор источника привязки

Как вы уже видели, по умолчанию связыватель модели ASP.NET Core будет пытаться привязать модели привязки из трех разных источников: данные формы, данные маршрута и строка запроса.

Иногда вам может потребоваться специально объявить, к какому источнику привязки нужно выполнить привязку. В других случаях этих трех источников вообще будет недостаточно. Наиболее распространенные сценарии – когда нужно привязать параметр метода к значению заголовка запроса или когда тело запроса содержит данные в формате JSON, которые вы хотите привязать к параметру. В этих случаях можно декорировать модели привязки атрибутами, которые говорят, откуда выполнять привязку, как показано в следующем листинге.

Листинг 16.5 Выбор источника привязки для привязки модели

```
public class PhotosModel: PageModel
{
    public void OnPost(
        [FromHeader] string userId, ← UserId будет привязан к HTTP-
        [FromBody] List<Photo> photos) ← заголовку в запросе
    {
        /* Реализация метода; */
    }
}
```

Список объектов Photo будет привязан к телу запроса, обычно в формате JSON

В этом примере обработчик страницы обновляет коллекцию фотографий по идентификатору пользователя. Здесь есть параметры метода для идентификатора пользователя, которого нужно отметить на фотографиях, `userId`, и список объектов `Photo` для пометки, `photos`.

Вместо того чтобы выполнить привязку этих параметров метода с помощью стандартных источников привязки, я добавил к каждому параметру атрибуты, указывая на источник привязки, который будет использоваться. Атрибут `[FromHeader]` применен к параметру `userId`. Так мы сообщаем связывателю модели привязать значение к значению заголовка HTTP-запроса, `userId`.

Мы также привязываем список фотографий к телу HTTP-запроса с помощью атрибута `[FromBody]`. Это указывает связывателю читать данные в формате JSON из тела запроса и выполнять привязку к параметру метода `List<Photo>`.

ВНИМАНИЕ! Разработчикам, знакомым с предыдущей версией платформы .NET и предыдущей версией ASP.NET, следует обратить внимание на то, что при привязке к запросам в формате JSON в Razor Pages явно требуется атрибут `[FromBody]`. Это отличается от предыдущего поведения ASP.NET, где атрибут не требовался.

Вы не ограничены привязкой данных JSON из тела запроса – можно использовать и другие форматы, в зависимости от того, какие форматеры ввода вы настраиваете для работы с фреймворком. По умолчанию настроен только форматер ввода JSON. Вы увидите, как добавить форматер XML, в главе 20, когда мы будем обсуждать веб-API.

СОВЕТ Автоматическое связывание нескольких форматов из тела запроса является одной из функций, специфичных для Razor Pages и контроллеров MVC, которые отсутствуют в минимальных API.

Можно использовать несколько разных атрибутов, чтобы переопределить значения по умолчанию и указать источник привязки для каждой модели привязки (или каждого свойства модели привязки). Это те же атрибуты, которые мы использовали в главе 7 с минимальными API:

- `[FromHeader]` – привязка к значению заголовка;
- `[FromQuery]` – привязка к значению строки запроса;

- `[FromRoute]` – привязка к параметрам маршрута;
- `[FromForm]` – привязка к данным формы, размещенным в теле запроса. Этот атрибут недоступен в минимальных API;
- `[FromBody]` – привязка к содержимому тела запроса.

Вы можете применить каждый из них к любому количеству параметров или свойств метода обработчика, как вы уже видели в листинге 16.5, за исключением атрибута `[FromBody]` – этим атрибутом можно декорировать только одно значение. Кроме того, при отправке данных формы в теле запроса атрибуты `[FromBody]` и `[FromForm]` фактически являются взаимоисключающими.

СОВЕТ Только один параметр может использовать атрибут `[FromBody]`. Этот атрибут будет использовать данные тела входящего запроса, поскольку тело HTTP-запроса можно безопасно прочитать только один раз.

Помимо этих атрибутов для указания источников привязки, есть еще несколько атрибутов для дальнейшей настройки процесса привязки:

- `[BindNever]` – связыватель модели пропустит этот параметр. Можно использовать атрибут `[BindNever]` для предотвращения массового назначения, как описано в двух постах моего блога: <http://mng.bz/QvfG> и <http://mng.bz/Vd90>;
- `[BindRequired]` – если параметр не был указан или был пустым, связыватель добавит ошибку валидации модели;
- `[FromServices]` – используется, чтобы указать, что параметр должен быть предоставлен с применением внедрения зависимостей. В большинстве случаев этот атрибут не требуется, поскольку .NET 7 достаточно умна, чтобы знать, что параметр – это сервис, зарегистрированный в контейнере внедрения зависимостей, но при желании вы можете указать это явно.

Кроме того, у вас есть атрибут `[ModelBinder]`, который переводит вас в «режим Бога», что касается привязки модели. С помощью этого атрибута можно указать точный источник привязки, переопределить имя параметра для привязки и указать тип привязки, который нужно выполнить. Вы вряд ли будете часто его использовать.

Объединив все эти атрибуты, вы обнаружите, что можно настроить привязку модели для привязки практически ко всем данным запроса, которые хочет использовать обработчик страницы. Однако в целом, вероятно, вам редко придется их использовать; в большинстве случаев вам подойдут значения по умолчанию.

На этом мы подошли к концу раздела, посвященного привязке модели. В конце процесса привязки модели обработчик страницы должен иметь доступ к заполненной модели привязки и быть готовым выполнить свою логику. Но, прежде чем использовать вводимые пользователем данные для чего-либо, вы всегда должны *проверять* их, а этому и посвящена вторая половина данной главы. Razor Pages автоматически выполняет валидацию по умолчанию, но вам придется фактически проверять ее результат.

16.3 Валидация моделей привязки

В этом разделе я расскажу, как работает валидация в Razor Pages. В главе 7 вы уже узнали, насколько важно проверять вводимые пользователем данные, а также как использовать атрибуты `DataAnnotation` для декларативного описания требований к валидации модели. В этом разделе вы узнаете, как повторно использовать эти знания для валидации моделей привязки Razor Page. Хорошая новость состоит в том, что в Razor Pages валидация является встроенной.

16.3.1 Валидация в Razor Pages

В главе 7 вы узнали, что валидация является важной частью любого веб-приложения. Тем не менее в минимальных API валидация в рамках фреймворка не поддерживается напрямую; вам придется накладывать ее поверх, используя фильтры и дополнительные пакеты.

В Razor Pages валидация является встроенной. Она происходит автоматически после привязки модели, но до выполнения обработчика страницы, как показано на рис. 16.2. На рис. 16.6 представлено более компактное представление того, какое место в этом процессе занимает валидация модели, и продемонстрировано, как привязывается и проверяется запрос на страницу оформления заказа, запрашивающий личные данные пользователя.

1. Получен запрос на URL

`/checkout/saveuser`, и компонент маршрутизации выбирает страницу `SaveUser` в каталоге `Checkout`

2. Фреймворк создает `UserBindingModel` из деталей запроса

3. `UserBindingModel` валидируется согласно атрибутам `DataAnnotation` на свойствах

4. `UserBindingModel` и модель состояния (`ModelState`) валидации устанавливаются в свойства страницы `SaveUser`, и выполняется обработчик страницы

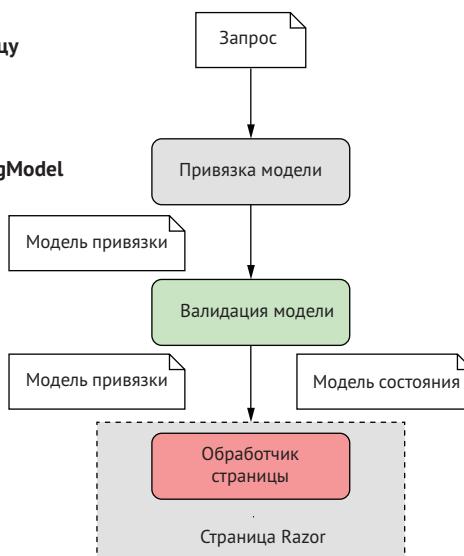


Рис. 16.6 Валидация модели происходит после привязки, но до выполнения обработчика страницы. Обработчик страницы выполняется независимо от того, прошла проверка успешно или нет

Как обсуждалось в главе 7, валидация заключается не только в защите от угроз безопасности. Важно также убедиться в том, что:

- данные правильно отформатированы (поля адресов электронной почты имеют допустимый формат);
 - числа находятся в определенном диапазоне (нельзя купить -1 копию товара);
 - некоторые значения обязательны, а другие – нет (может потребоваться имя, но номер телефона указывать не обязательно);
 - значения соответствуют бизнес-требованиям (нельзя конвертировать валюту в ту же валюту, ее нужно конвертировать в другую).

Может показаться, что что-то из этого можно легко обработать в браузере. Например, если пользователь выбирает валюту для конвертации, не позволять ему выбирать ту же валюту; и все мы видели сообщения «введите действующий адрес электронной почты».

К сожалению, хотя эта *валидация на стороне клиента* полезна для пользователей, поскольку дает им мгновенный отклик, на нее никогда нельзя полагаться, так как эти средства защиты браузера всегда можно обойти. Всегда необходимо проверять данные по мере их поступления в приложение, используя *валидацию на стороне сервера*.

ВНИМАНИЕ! Всегда проверяйте данные, введенные пользователем на стороне сервера.

Если вам это кажется лишним, например вам нужно будет дублировать логику и код, тогда боюсь, что вы правы. Это один из прискорбных аспектов веб-разработки; дублирование – необходимое зло. К счастью, ASP.NET Core предоставляет несколько функций, которые могут попробовать уменьшить это бремя.

COBET Blazor, новый фреймворк C# для создания одностраничных приложений, обещает решить некоторые из этих проблем. Для получения дополнительной информации см. <http://mng.bz/9D51> и книгу Криса Сейнти «Blazor в действии».

Если бы вам пришлось каждый раз писать новый код валидации для каждого приложения, это было бы утомительно и, вероятно, привело бы к появлению ошибок. К счастью, можно использовать атрибуты `DataAnnotations` для декларативного описания требований валидации для своих моделей привязки. В следующем листинге, впервые продемонстрированном в главе 7, показано, как декорировать модель привязки различными атрибутами валидации. Это расширенный вариант примера, который вы видели ранее в листинге 16.4.

Листинг 16.6. Добавление DataAnnotations в модель привязки для представления метаданных

```

[Required]
[StringLength(100)]
[Display(Name = "Last name")]
public string LastName { get; set; }

[Required]
[EmailAddress]           ←
public string Email { get; set; }

}

[Phone]           ←
[Display(Name = "Phone number")]
public string PhoneNumber { get; set; }
}

```

Проверяет, что значение Email является валидным адресом электронной почты

Проверяет, что значение PhoneNumber имеет валидный телефонный формат

Что касается требований валидации, где атрибуты не подходят, например когда достоверность одного свойства зависит от значения другого, можно реализовать `IValidatableObject`, как описано в главе 7. В качестве альтернативы можно использовать другой фреймворк валидации, например `FluentValidation`, как вы увидите в главе 32.

Какой бы подход вы ни использовали, важно помнить, что эти методы сами по себе не защищают приложение. Razor Pages обеспечивает выполнение проверки, но ничего не делает автоматически, если валидация модели потерпела неудачу. В следующем разделе мы рассмотрим, как проверить результат валидации на сервере и что делать, если проверка не удалась.

16.3.2 Валидация на сервере в целях безопасности

Валидация модели привязки происходит до выполнения обработчика страницы, но обратите внимание, что обработчик всегда выполняется, независимо от того, завершилась валидация успешно или была неудачной. Обработчик страницы должен проверить результат.

ПРИМЕЧАНИЕ Валидация модели происходит автоматически, но обработка ошибок проверки является обязанностью обработчика страницы.

Razor Pages сохраняет выходные данные попытки валидации модели в свойстве модели страницы – `ModelState`. Это свойство представляется собой объект `ModelStateDictionary`, содержащий список всех ошибок валидации, возникших после привязки модели, а также некоторые служебные свойства для работы с ним.

В качестве примера в следующем листинге показан обработчик `OnPost` для страницы `Checkout.cshtml`. Свойство `Input` помечено для привязки и использует тип `UserBindingModel`, показанный ранее в листинге 16.6. Этот обработчик страницы ничего не делает с данными, но главное здесь – шаблон проверки `ModelState` в начале метода.

Листинг 16.7 Проверка состояния модели для просмотра результата валидации

```

public class CheckoutModel : PageModel
{
    [BindProperty]
    public UserBindingModel Input { get; set; }

    public IActionResult OnPost()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }
        /* Сохранение в базу данных, обновление пользователя и возврат
           сообщения об успехе; */
        return RedirectToAction("Success");
    }
}

```

Свойство ModelState доступно в базовом классе PageModel

Свойство Input содержит данные, привязанные к модели

Модель привязки проверяется перед выполнением обработчика страницы

Проверка не удалась, поэтому повторно отобразите форму с ошибками и завершите метод досрочно

Если были ошибки проверки, IsValid будет false

Валидация пройдена, поэтому можно безопасно использовать данные, представленные в модели

Если свойство `ModelState` указывает, что произошла ошибка, тотчас же вызывается вспомогательный метод `Page`. Он возвращает объект `PageResult`, который в конечном итоге генерирует HTML-код, чтобы вернуть его пользователю, как показано в главе 15. Представление использует (недопустимые) значения, предоставленные в свойстве `Input` для повторного заполнения формы при ее отображении, как показано на рис. 16.7. Кроме того, полезные сообщения для пользователя добавляются автоматически с использованием ошибок валидации в свойстве `ModelState`.

ПРИМЕЧАНИЕ Сообщения об ошибках, отображаемые в форме, являются значениями по умолчанию для каждого атрибута валидации. Вы можете настроить сообщение, задав свойство `ErrorMessage` для любого из атрибутов. Например, можно настроить атрибут `[Required]`, используя `[Required(ErrorMessage = "Required")]`.

Если запрос успешен, обработчик страницы возвращает объект `RedirectResult` (используя вспомогательный метод `RedirectToPage()`), который перенаправляет пользователя на страницу `Success.cshtml`. Данный паттерн возврата ответа перенаправления после успешного вызова POST называется POST-REDIRECT-GET.

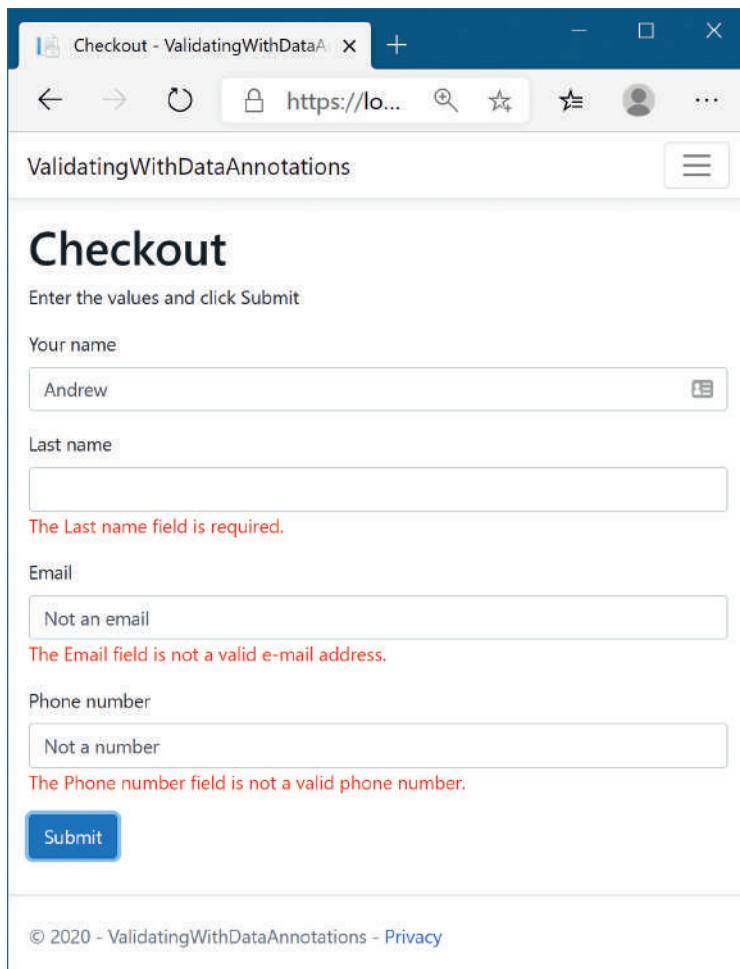


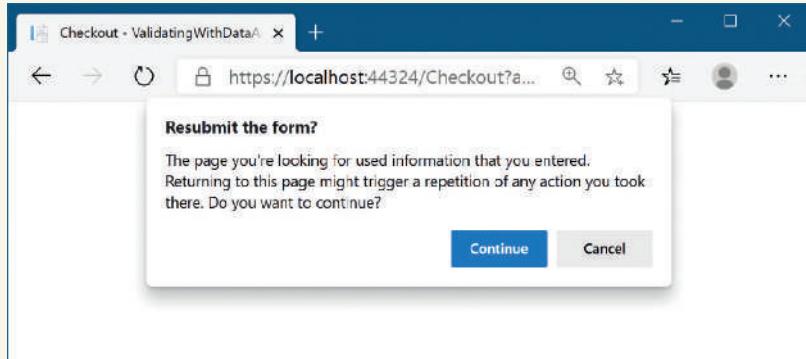
Рис. 16.7 Если валидация завершилась неудачно, можно повторно отобразить форму, чтобы показать пользователю ошибки из ModelState. Обратите внимание, что поле «Ваше имя» не имеет ассоциированных ошибок валидации, в отличие от других полей

POST-REDIRECT-GET

POST-REDIRECT-GET – это паттерн веб-разработки, который не дает пользователю случайно отправить одни и те же данные через форму несколько раз. Пользователи обычно отправляют данные через форму, используя стандартный механизм браузера POST, отправляющий данные на сервер. Например, это обычный способ для отправки данных платежа.

Если сервер использует простой подход и в ответ выдает сообщение с кодом 200 OK и некий HTML-код, пользователь по-прежнему будет использовать тот же URL-адрес. Если пользователь затем обновит окно браузера, то отправит дополнительный запрос с методом POST на сервер, потенциально совершая еще один платеж! У браузеров есть меха-

НИЗМЫ, позволяющие избежать этого, например как показано на следующем рисунке, но такая ситуация нежелательна.



При обновлении окна браузера после запроса с методом POST пользователь видит сообщение с предупреждением

Паттерн POST-REDIRECT-GET определяет, что в ответ на успешный запрос с методом POST нужно вернуть ответ REDIRECT на новый URL-адрес, после которого браузер выполнит запрос, используя метод GET для нового URL-адреса. Если пользователь обновит страницу в браузере, то будет обновлять последний вызов с методом GET для нового URL-адреса. Никаких дополнительных запросов с методом POST не производится, поэтому не должно быть никаких дополнительных платежей или побочных эффектов.

Этого легко добиться в приложениях ASP.NET Core MVC, используя паттерн, показанный в листинге 16.7. Возвращая объект `RedirectToPageResult` после успешного выполнения POST, ваше приложение будет в безопасности, если пользователь обновит страницу в своем браузере.

Вам, наверное, интересно, почему ASP.NET Core не обрабатывает недействительные запросы автоматически – если валидация не удалась, а результат у вас есть, почему обработчик страницы вообще выполняется? Нет ли риска, что вы можете забыть проверить результат?

Это правда, и в некоторых случаях лучше всего сделать автоматическую генерацию проверки и ответа. Фактически это именно тот подход, который мы будем использовать для веб-API, применяя контроллеры MVC с атрибутом `[ApiController]`, когда будем рассматривать их в главе 20.

Однако для приложений Razor Pages обычно требуется сгенерировать ответ в виде HTML-кода, даже если валидация не удалась. Это позволяет пользователю увидеть проблему и потенциально исправить ее. Сделать это автоматически намного сложнее.

Например, вы можете обнаружить, что вам нужно загрузить дополнительные данные, прежде чем вы сможете повторно отобразить страницу Razor, например загрузить список доступных валют. С помощью шаблона `ModelState.IsValid` сделать это проще и яснее. Попытка сделать это автоматически, скорее всего, закончится борьбой с пограничными случаями и обходными путями.

Кроме того, за счет явного включения проверки с помощью `IsValid` в обработчики страниц легче контролировать, что происходит в случае сбоя дополнительных валидаций. Например, если пользователь пытается обновить продукт, то при валидации с помощью атрибутов `DataAnnotation` вы не узнаете, существует ли продукт с запрошенным идентификатором, а узнаете только, есть ли у идентификатора правильный формат. Перенося проверку в метод обработчика, вы можете работать с ошибками валидации данных и бизнес-правил одинаковым образом.

СОВЕТ Вы также можете добавить в коллекцию дополнительные ошибки валидации, например ошибки валидации бизнес-правил, поступающие из другой системы. Можно добавить ошибки в `ModelState`, вызвав метод `AddModelError()`, который будет отображаться пользователям в форме вместе с ошибками атрибута `DataAnnotation`.

Надеюсь, мне удалось донести до вас, насколько важно проверять данные, вводимые пользователями в ASP.NET Core, но на всякий случай повторю: ВЫПОЛНЯЙТЕ ВАЛИДАЦИЮ! Договорились?! Однако валидация только на стороне сервера может оставить у пользователей негативные впечатления. Сколько раз вы заполняли онлайн-форму, отправляли ее, уходили перекусить и, возвращаясь, выясняли, что вы где-то сделали опечатку и вам придется все переделывать. Не лучше было бы получить отклик сразу же?

16.3.3 Валидация на стороне клиента для улучшения пользовательского опыта

Можно добавить в приложение валидацию на стороне клиента несколькими способами. В HTML5 существует несколько встроенных режимов валидации, которые будут использовать многие браузеры. Если открыть страницу с полем адреса электронной почты и использовать следующий синтаксис `<input type="email">`, то браузер автоматически не даст вам указать недопустимый формат почты, как показано на рис. 16.8.

ПРИМЕЧАНИЕ Поддержка проверки ограничений HTML5 зависит от браузера. Подробнее о доступных ограничениях см. в документации Mozilla (<http://mng.bz/daX3>) и на странице <http://mng.bz/XNo1>.

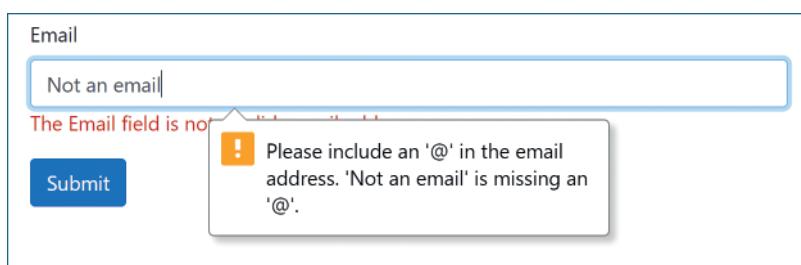


Рис. 16.8 По умолчанию современные браузеры автоматически проверяют то, что вы вводите в поле адреса электронной почты

Альтернативный подход – выполнить валидацию на стороне клиента, запустив на странице Java Script и проверяя значения, введенные пользователем перед отправкой формы. Это наиболее распространенный подход, используемый в Razor Pages.

Я подробно расскажу, как сгенерировать помощники валидации на стороне клиента, в главе 18, где вы снова увидите, как атрибуты `DataAnnotation` выходят на первый план. Декорируя модель представления этими атрибутами, вы предоставляете механизму Razor метаданные, необходимые для создания соответствующего HTML-кода.

При таком подходе пользователь сразу видит все ошибки в своей форме, еще до того, как запрос будет отправлен на сервер, как показано на рис. 16.9. Это дает гораздо более короткий цикл обратной связи, обеспечивая более качественный пользовательский опыт.

Если вы создаете одностраничное приложение, ответственность за валидацию данных на стороне клиента перед их отправкой в API лежит на клиентском фреймворке. API по-прежнему будет проверять данные, когда они поступают на сервер, но фреймворк отвечает за обеспечение плавного взаимодействия с пользователем.

Когда вы используете Razor Pages для генерации HTML-кода, то получаете большую часть этой валидации бесплатно. Он автоматически настраивает валидацию на стороне клиента для большинства встроенных атрибутов, не требуя дополнительной работы, как вы увидите в главе 18.

К сожалению, если вы используете свои собственные атрибуты `ValidationAttributes`, по умолчанию они будут выполняться только на сервере; вам нужно выполнить дополнительное подключение атрибута, чтобы он также работал и на стороне клиента. Несмотря на это, пользовательские атрибуты валидации могут быть полезны при работе с распространенными сценариями проверки в приложении, как вы увидите в главе 31.

Фреймворк привязки модели в ASP.NET Core дает множество вариантов для настройки страниц Razor: параметры обработчика страницы или свойства модели страницы; одна модель привязки или несколько; варианты того, где определять классы модели привязки. В следующем разделе я дам несколько советов относительно того, какая организация страниц Razor нравится мне.

16.4 Организация моделей привязки в Razor Pages

В этом разделе я даю несколько общих советов по поводу того, как мне нравится настраивать модели привязки в Razor Pages. Если вы будете следовать шаблонам из этого раздела, ваши страницы Razor будут придерживаться согласованного макета, чтобы другим было легче понять, как работает каждая страница в вашем приложении.

ПРИМЕЧАНИЕ Это сугубо личный совет, поэтому не стесняйтесь адаптировать его, если найдете аспекты, с которыми вы не согласны. Важно понять, почему я вношу каждое предложение, и принимать его во внимание. В случае необходимости я тоже отклоняюсь от этих рекомендаций!

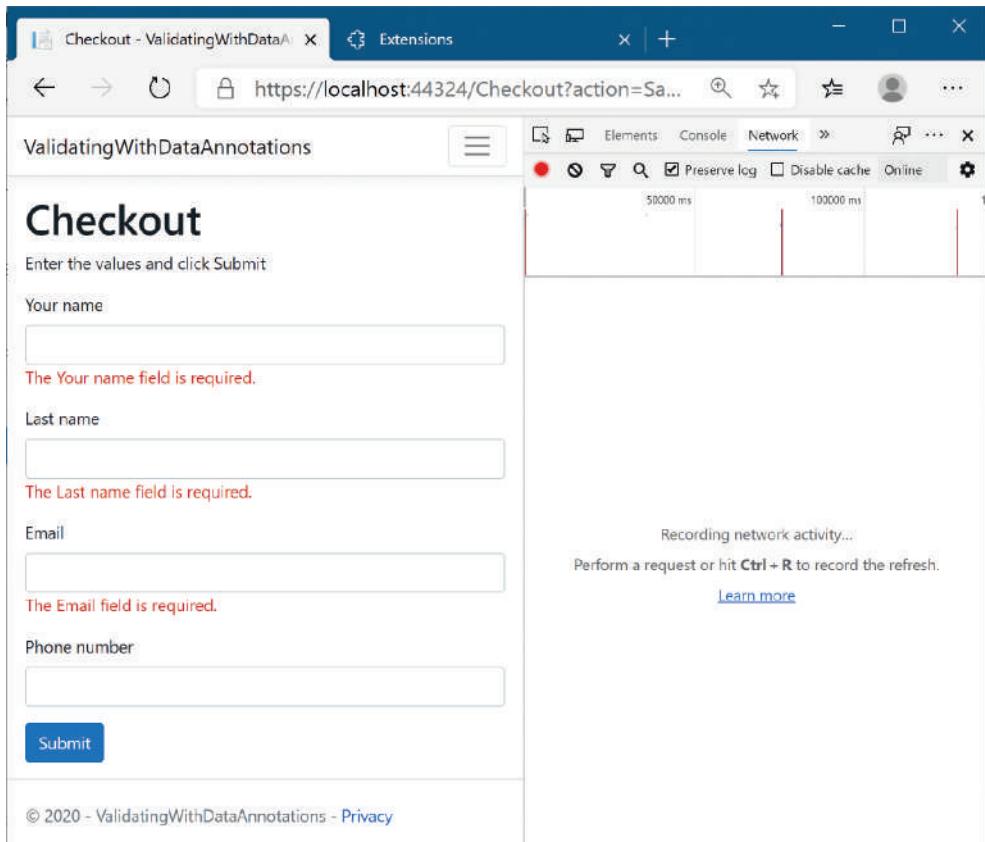


Рис. 16.9 При валидации на стороне клиента нажатие кнопки «Отправить» инициирует проверку, которая будет отображаться в браузере, до того, как запрос отправится на сервер. Как показано на правой панели, запрос не отправляется

Привязка модели в ASP.NET Core имеет множество эквивалентных подходов, поэтому «правильного» способа не существует. В следующем листинге показан пример того, как бы я спроектировал простую страницу Razor. В нем отображена форма продукта с заданным идентификатором. Страница позволяет редактировать детали с помощью запроса методом POST. Это гораздо более длинный пример, по сравнению с тем, что мы рассматривали до сих пор, но ниже я выделю важные моменты.

Листинг 16.8 Проектирование страницы Razor для редактирования информации о продукте

```
public class EditProductModel : PageModel
{
    private readonly ProductService _productService;
    public EditProductModel(ProductService productService)
    {
        _productService = productService;
    }
}
```

ProductService внедряется с помощью внедрения зависимостей и обеспечивает доступ к моделям приложения

```

[BindProperty]
public InputModel Input { get; set; }

public IActionResult OnGet(int id)
{
    var product = _productService.GetProduct(id);

    Input = new InputModel
    {
        Name = product.ProductName,
        Price = product.SellPrice,
    };
    return Page();
}

public IActionResult OnPost(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _productService.UpdateProduct(id, Input.Name, Input.Price);
    return RedirectToPage("Index");
}

public class InputModel
{
    [Required]
    public string Name { get; set; }

    [Range(0, int.MaxValue)]
    public decimal Price { get; set; }
}

```

Загружает сведения о продукте из модели приложения

Обновляет продукт в модели приложения с помощью ProductService

Одно свойство помечается BindProperty

Параметр id привязан к модели из шаблона маршрута для обработчиков OnGet и OnPost

Создает экземпляр InputModel для редактирования в форме на основе сведений о существующем продукте

Параметр id привязан к модели из шаблона маршрута для обработчиков OnGet и OnPost

Если запрос невалидный, повторно отображает форму без сохранения

Перенаправляет на новую страницу, используя подход POST-перенаправление-GET

Определяет InputModel как вложенный класс на странице Razor

На этой странице показана модель PageModel типичной «формы редактирования». Она очень часто встречается во многих бизнес-приложениях, и это тот сценарий, для которого прекрасно подходит Razor Pages. Вы увидите, как создавать HTML-сторону форм, в главе 18.

ПРИМЕЧАНИЕ Цель данного примера – только описать подход к привязке модели. С точки зрения логики код излишне упрощен. Например, здесь не проверяется, существует ли продукт с указанным идентификатором, и нет обработки ошибок.

Эта форма показывает несколько паттернов, связанных с привязкой модели, которых я стараюсь придерживаться при создании страниц Razor:

- выполняйте привязку только одного свойства с помощью атрибута [BindProperty] – я предпоючила, чтобы у меня было лишь одно свой-

ство, декорированное `[BindProperty]` для привязки модели в целом. Когда нужно привязать несколько значений, я создаю отдельный класс `InputModel`, где будут храниться значения. Я декорирую это единственное свойство атрибутом `[BindProperty]`. При декорировании одного такого свойства сложнее забыть добавить атрибут, а это означает, что все ваши страницы Razor используют один и тот же шаблон;

- определите свою модель привязки как вложенный класс. Я определяю `InputModel` как вложенный класс внутри страницы Razor. Модель привязки обычно очень специфична для этой единственной страницы, поэтому все, над чем вы работаете, будет собрано воедино. Кроме того, я обычно использую именно `InputModel` в качестве имени класса для всех своих страниц, что опять же добавляет им единообразия;
- не используйте атрибут `[BindProperties]`. Помимо `[BindProperty]`, существует атрибут `[BindProperties]` (обратите внимание на разницу в написании), который можно применять к модели страницы Razor напрямую. Это приведет к тому, что все свойства в вашей модели будут привязаны. Это может сделать вас уязвимым для атак с помощью оверпостинга, если вы не будете проявлять осторожность. Я предлагаю вам не использовать атрибут `[BindProperties]` и вместо этого придерживаться привязки *одного* свойства с помощью атрибута `[BindProperty]`;
- принимайте параметры маршрута в обработчике страницы. В случае с простыми параметрами маршрута, такими как `id`, передаваемого в обработчики `OnGet` и `OnPost` в листинге 16.8, я добавляю параметры к самому методу обработчика страницы. Это позволяет избежать неуклюжего синтаксиса типа `Supports Get=true` для запросов методом GET;
- всегда выполняйте валидацию модели перед использованием данных. Я уже говорил это прежде, поэтому повторю снова. Проверяйте корректность данных, вводимых пользователем.

На этом мы завершаем обзор привязки модели в Razor Pages. Вы увидели, как фреймворк ASP.NET Core использует привязку модели, чтобы упростить процесс извлечения значений из запроса и преобразования их в обычные .NET-объекты, с которыми можно быстро работать. Наиболее важным аспектом этой главы является валидация модели – это распространенная проблема всех веб-приложений, а использование атрибутов `DataAnnotations` может облегчить процесс добавления валидации к вашим моделям.

В следующей главе мы продолжим наше путешествие по Razor Pages и рассмотрим, как создавать представления. В частности, вы узнаете, как сгенерировать HTML-код в ответ на запрос, используя механизм шаблонов Razor.

Резюме

- Razor Pages использует три различные модели, каждая из которых отвечает за свой аспект запроса. Модель привязки инкапсулирует данные, отправляемые как часть запроса. Модель приложения пред-

ставляет состояние приложения. Модель страницы – это дополнительный класс для страницы Razor, который предоставляет данные, используемые представлением Razor, чтобы сгенерировать ответ;

- привязка модели извлекает значения из запроса и использует их для создания объектов .NET, которые обработчик страницы может использовать при выполнении. Любые свойства модели страницы, отмеченные атрибутом `[BindProperty]`, и параметры методов обработчиков страниц будут принимать участие в привязке модели;
- по умолчанию существует три источника привязки: значения формы, переданные с помощью метода POST, значения маршрута и строка запроса. Связыватель будет опрашивать их по порядку при попытке привязать модель привязки;
- при привязке значений к моделям имена параметров и свойств не чувствительны к регистру;
- вы можете выполнять привязку к простым типам или к свойствам сложных типов. Простые типы должны иметь возможность конвертироваться из строк для автоматической привязки, например числа, даты, логические значения и пользовательские типы с помощью метода `TryParse`;
- чтобы привязать сложные типы, они должны иметь конструктор по умолчанию и открытые доступные для записи свойства. Связыватель моделей Razor Pages привязывает каждое свойство сложного типа, используя значения из источников привязки;
- коллекции и словари можно привязать с помощью синтаксиса `[index]=value` и `[key]=value`;
- вы можете настроить источник привязки для модели привязки с помощью атрибутов `[From*]`, примененных к методу, например `[FromHeader]` и `[FromBody]`. Их можно использовать для привязки к источникам привязки не по умолчанию, таким как заголовки или содержимое тела в формате JSON. Атрибут `[FromBody]` всегда требуется при привязке к телу в формате JSON;
- валидация необходима для проверки на наличие угроз безопасности. Проверяйте правильность форматирования данных и убедитесь, что они соответствуют ожидаемым значениям и вашим бизнес-правилам;
- валидация в Razor Pages происходит автоматически после привязки модели, но необходимо вручную проверять результат валидации и действовать соответствующим образом в обработчике страницы, запрашивая свойство `ModelState.IsValid`;
- валидация на стороне клиента обеспечивает лучший пользовательский опыт, чем проверка на стороне сервера, но вы всегда должны использовать проверку на стороне сервера. При валидации на стороне клиента обычно используются JavaScript и атрибуты, применимые к элементам HTML для проверки значений формы.

17

Отрисовка HTML-кода с использованием представлений Razor

В этой главе:

- создание представлений Razor для отображения HTML-кода пользователю;
- использование C# и синтаксиса разметки Razor для динамической генерации HTML-кода;
- повторное применение общего кода с макетами и частичными представлениями.

Легко запутаться в терминах, используемых в Razor Pages, – модель страницы, обработчики страниц, представления Razor, – тем более что одни термины описывают конкретные функции, а другие – паттерны и концепции. Мы подробно обсудили все эти термины в предыдущих главах, но важно, чтобы вы четко понимали:

- *Razor Pages* – обычно Razor Pages обозначает парадигму веб-приложения на основе страниц, которая сочетает в себе маршрутизацию, привязку модели и генерацию HTML-кода с использованием представлений Razor;

- *страница Razor* – представляет отдельную страницу или «конечную точку». Обычно она состоит из двух файлов: файла с расширением .cshtml, содержащего представление Razor, и файла с расширением .cshtml.cs, содержащего модель страницы;
- *PageModel* – модель страницы Razor – это то место, где происходит большая часть действий. Здесь вы определяете модели привязки для страницы, которая извлекает данные из входящего запроса, а также определяете обработчики страницы;
- *обработчик страницы* – каждая страница Razor обычно обрабатывает один маршрут, но может обрабатывать несколько HTTP-методов, например GET и POST. Каждый обработчик страницы обычно обрабатывает один HTTP-метод;
- *представление Razor* – представления Razor (также называемые шаблонами Razor) используются для генерации HTML-кода. Обычно они используются на заключительном этапе страницы Razor для генерации ответа в виде HTML-кода, который будет отправлен пользователю.

В предыдущих четырех главах мы рассмотрели все разделы Razor Pages, включая паттерн проектирования MVC, модель страницы Razor, обработчики страниц, маршрутизацию и модели привязки. В этой главе разбирается последняя часть паттерна MVC – использование представления для генерации HTML-кода, который доставляется в браузер пользователя.

В ASP.NET Core представления обычно создаются с использованием синтаксиса разметки *Razor* (иногда его называют языком шаблонов), который использует смесь HTML-кода и кода C# для генерации окончательного варианта. В этой главе рассказывается о некоторых функциях Razor и о том, как использовать его для создания шаблонов представлений приложения. Пользователи будут взаимодействовать с вашим приложением двумя способами: будут читать данные, которые оно отображает, и будут в ответ отправлять ему данные или команды. Язык Razor содержит ряд конструкций, упрощающих создание приложений обоих типов.

При отображении данных можно использовать язык Razor, чтобы с легкостью сочетать статический HTML-код и значения из модели страницы. Razor может использовать C# в качестве механизма управления, поэтому добавлять условные элементы и циклы просто – чего нельзя добиться с помощью одного только HTML.

Обычный подход к отправке данных в веб-приложения – это HTML-формы. Практически каждое динамическое приложение, которое вы создаете, будет использовать формы, а некоторые приложения будут представлять собой *только* формы! ASP.NET Core и язык шаблонов Razor включают в себя *тег-хелперы*, которые упрощают создание HTML-форм.

ПРИМЕЧАНИЕ В следующем разделе вы кратко познакомитесь с тег-хелперами, а более подробно мы рассмотрим их в следующей главе.

В этой главе мы сосредоточимся в первую очередь на отображении данных и генерации HTML-кода с помощью Razor, а не на создании

форм. Вы увидите, как преобразовать значения из модели страницы в HTML-код и использовать C# для управления генерированным выводом. Наконец, вы узнаете, как извлечь распространенные элементы представлений во вложенные представления, которые называются *макетами и частичными представлениями*, а также способы их компоновки для создания окончательной HTML-страницы.

17.1 Представления: отрисовка пользовательского интерфейса

В этом разделе я даю краткое введение в отрисовку HTML-кода с использованием представлений Razor. Мы резюмируем все вышесказанное о паттерне проектирования MVC, используемом Razor Pages, и посмотрим, где лучше всего использовать представление. Затем я расскажу, как синтаксис Razor позволяет смешивать C# и HTML для создания динамических пользовательских интерфейсов.

Как вы знаете из предыдущих глав, где говорится о паттерне проектирования MVC, обязанность выбирать, что вернуть клиенту, лежит на обработчике страницы Razor. Например, если вы разрабатываете приложение со списком дел, представьте себе запрос на просмотр конкретного элемента, как показано на рис. 17.1.

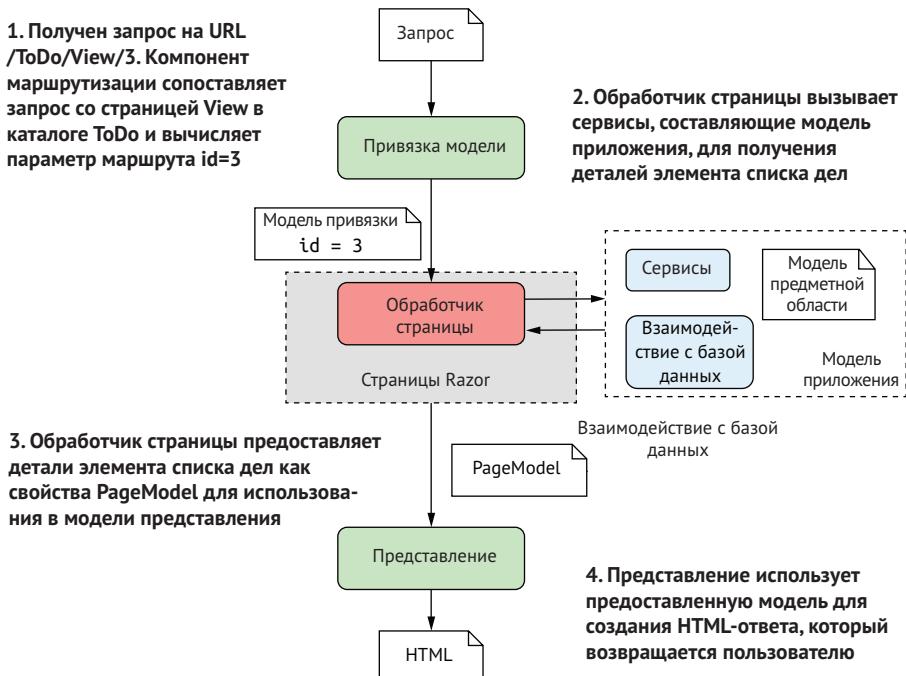


Рис. 17.1 Обработка запроса элемента списка дел с помощью Razor Pages. Обработчик страницы создает данные, необходимые для представления, и предоставляет их как свойства модели страницы. Представление генерирует HTML-код только на основе предоставленных данных; ему не нужно знать, откуда они

Типичный запрос следует этапам, показанным на рис. 17.1:

- конвейер промежуточного ПО получает запрос, а компонент маршрутизации определяет конечную точку для вызова – в данном случае страницу `View` в папке `ToDo`;
- связыватель модели (часть фреймворка Razor Pages) использует запрос для сборки моделей привязки страницы, как вы видели в предыдущей главе. Модели привязки задаются как свойства на странице Razor или передаются методу обработчика страницы в качестве аргументов при выполнении обработчика. Он проверяет, что вы передали допустимый идентификатор элемента списка дел и помечает `ModelState` как валидный, если это так;
- предполагая, что все в порядке, обработчик страницы обращается к различным сервисам, из которых состоит модель приложения. Можно загрузить подробную информацию о текущих делах из базы данных или файловой системы, возвращая их обработчику. В рамках этого процесса либо модель приложения, либо сам обработчик страницы генерирует значения для передачи их в представление и задает их в качестве свойств модели страницы Razor.

После выполнения обработчика страницы модель страницы должна содержать все данные, необходимые для отрисовки представления. В этом примере она содержит подробную информацию о самом элементе списка, но также может содержать и другие данные: сколько дел у вас осталось, есть ли у вас какие-либо дела на сегодня, ваше имя пользователя и т. д. – все, что контролирует создание конечного пользовательского интерфейса для запроса;

- шаблон представления Razor использует модель страницы для генерации окончательного ответа и возвращает его пользователю через конвейер промежуточного ПО.

Общей темой в этом обсуждении MVC является разделение задач, которое обеспечивает MVC, и представления не являются исключением. Было бы достаточно легко напрямую сгенерировать HTML-код в модели приложения или в действиях контроллера, но вместо этого мы делегируем эту ответственность одному единственному компоненту, известному как представление.

И даже более того. Мы также будем отделять *данные*, необходимые для сборки представления, от *процесса* его создания с помощью свойств модели страницы. Эти свойства должны содержать все динамические данные, необходимые представлению для генерации окончательного вывода.

СОВЕТ Представления не должны вызывать методы модели страницы – как правило, представление должно получать доступ только к данным, которые уже были собраны и представлены в виде свойств.

Обработчики страниц Razor указывают на то, что представление Razor следует визуализировать, возвращая объект `PageResult` (или `void`),

как вы видели в главе 15. Инфраструктура Razor Pages выполняет представление Razor, связанное с данной страницей Razor, для генерации окончательного ответа. Использование C# в шаблоне Razor означает, что вы можете динамически генерировать окончательный HTML-код, отправляемый в браузер. Это позволяет, например, отображать имя текущего пользователя на странице, скрывать ссылки, к которым у пользователя нет доступа, или отображать кнопку для каждого элемента в списке.

Представьте, что ваш начальник просит вас добавить в приложение страницу со списком пользователей приложения. Вы также должны иметь возможность перейти к просмотру пользователя с этой страницы или создать нового, как показано на рис. 17.2.

PageModel содержит данные, которые вы желаете отобразить на странице

Model.ExistingUsers = new[] {
 "Andrew",
 "Robbie",
 "Jimmy",
 "Bart"
 };

Разметка Razor описывает, как отображать данные, используя смесь HTML и C#

```
@foreach(var user in Model.ExistingUsers)
{
  <li>
    <span>@user</span>
    <button>View</button>
  </li>
}
```

Совмещая данные в модели представления и разметку Razor, HTML может создаваться динамически, вместо того чтобы быть созданным в момент компиляции программы

Элементы формы могут использоватьсь для отправки данных приложению

Рис. 17.2 Использование C# в Razor позволяет с легкостью генерировать динамический HTML-код, который изменяется во время выполнения. В этом примере использование цикла foreach в представлении Razor значительно сокращает дублирование в HTML-коде. В противном случае нам пришлось бы все это писать самим

С помощью шаблонов Razor создать такого рода динамическое содержимое очень просто. Например, в листинге 17.1 показан шаблон, который можно использовать для создания интерфейса, представленного на рис. 17.2. Он сочетает в себе стандартный HTML-код с инструкциями C# и использует тег-хелперы для генерации элементов форм.

Листинг 17.1. Шаблон Razor для вывода списка пользователей и формы для добавления нового пользователя

```
@page
@model IndexViewModel
```

```

<div class="row">
    <div class="col-md-6">
        <form method="post">
            <div class="form-group">
                <label asp-for="NewUser"></label>
                <input class="form-control" asp-for="NewUser" />
                <span asp-validation-for="NewUser"></span>
            </div>
            <div class="form-group">
                <button type="submit"
                    class="btn btn-success">Add</button>
            </div>
        </form>
    </div>
</div>

<h4>Number of users: @Model.ExistingUsers.Count</h4>
<div class="row">
    <div class="col-md-6">
        <ul class="list-group">
            @foreach (var user in Model.ExistingUsers)
            {
                <li class="list-group-item d-flex justify-content-between">
                    <span>@user</span>
                    <a class="btn btn-info"
                        asp-page="ViewUser"
                        asp-route-userName="@user">View</a>
                </li>
            }
        </ul>
    </div>
</div>

```

Обычный HTML отправляется в браузер без изменений

Тег-хелперы прикрепляются к элементам HTML для создания форм

Значения можно записывать из объектов C# в HTML

Конструкции C#, такие как циклы for, можно использовать в Razor

Тег-хелперы также можно использовать вне форм для помощи в создании другого HTML-кода

В этом примере демонстрируются различные функции Razor. Это смесь HTML-кода, написанного без изменений для вывода ответа, и различных конструкций C#, используемых для динамической генерации HTML-кода. Кроме того, можно увидеть несколько тег-хелперов. Они выглядят как обычные HTML-атрибуты, начинающиеся с буквосочетания `asp-`, и являются частью языка Razor. Они могут настраивать HTML-элемент, к которому прикреплены, изменяя способ его отображения, и делают создание HTML-форм намного проще. Не волнуйтесь, если данный шаблон кажется несколько громоздким; мы разберем все это по частям по мере прохождения данной и последующей глав.

Страницы Razor компилируются при сборке приложения. За кулисами они становятся просто еще одним классом C# в приложении. Также можно активировать компиляцию страниц Razor во время выполнения. Это позволяет изменять страницы Razor во время работы приложения без необходимости явно останавливаться и выполнять повторную сборку, что может быть удобно при локальной разработке, но лучше избегать этого при развертывании в промышленном окружении. На странице <http://mng.bz/jP2P> можно прочитать, как это активировать.

ПРИМЕЧАНИЕ Как и большинство вещей в ASP.NET Core, движок шаблонов Razor можно заменить собственным движком для отрисовки на стороне сервера, но нельзя заменить Razor на такие фреймворки, как Angular или React. Если вы хотите применить данный подход, то используйте минимальные API или контроллеры веб-API и отдельный фреймворк для разработки веб-приложений.

В следующем разделе мы подробнее рассмотрим, как представления Razor вписываются в фреймворк Razor Pages и как передавать данные из обработчиков страниц Razor в представление Razor, чтобы помочь сгенерировать ответ в виде HTML-кода.

17.2 Создание представлений Razor

В этом разделе мы рассмотрим, как представления Razor вписываются в фреймворк Razor Pages. Вы узнаете, как передавать данные из обработчиков страниц в представления Razor и как использовать эти данные для создания динамического HTML-кода.

В ASP.NET Core всякий раз, когда вам нужно отобразить пользователю ответ в виде HTML-кода, вы должны использовать представление для его генерации. Хотя можно напрямую сгенерировать строку из обработчиков страниц, которая будет отображаться в браузере в виде HTML-кода, такой подход не соответствует разделению ответственности MVC и быстро приведет к тому, что вы начнете рвать на себе волосы.

ПРИМЕЧАНИЕ Некоторые компоненты промежуточного ПО, например WelcomePageMiddleware, показанный в главе 3, могут генерировать ответы в виде HTML-кода без использования представления, что может иметь смысл в некоторых ситуациях. Но страница Razor и контроллеры MVC всегда должны генерировать HTML-код с использованием представлений.

Полагаясь на представления Razor для генерации ответа, вы получаете доступ к широкому спектру функций, а также к инструментам редактора, которые могут вам помочь. Данный раздел служит кратким введением в представления Razor. В нем рассказывается, что можно с ними делать, и обсуждаются различные способы, с помощью которых можно передавать им данные.

17.2.1 Представления Razor и сопутствующий код

Как уже было показано в этой книге, обычно страницы Razor состоят из двух файлов:

- файла с расширением .cshtml, который обычно называют *представлением Razor*;
- файла с расширением .cshtml.cs, который обычно называют *сопутствующим кодом*. Он содержит модель страницы.

Представление Razor содержит директиву @page, которая делает его страницей Razor, как было показано в главе 4. Без нее фреймворк

Razor Pages не будет маршрутизировать запросы на страницу, и для большинства целей файл будет проигнорирован.

ОПРЕДЕЛЕНИЕ *Директива* – это инструкция в файле Razor, изменяющая способ парсинга или компиляции шаблона. Еще одна распространенная директива – `@using newNamespace`, которая делает доступными объекты в пространстве имен `newNamespace`.

Файл `.cshtml.cs` с сопутствующим кодом содержит модель связанный страницы Razor. В нем находятся обработчики страниц, которые отвечают на запросы, и именно здесь страница Razor обычно взаимодействует с другими частями приложения.

Несмотря на то что файлы `.cshtml` и `.cshtml.cs` имеют одно и то же имя, например `ToDoItem.cshtml` и `ToDoItem.cshtml.cs`, не оно их связывает. Но если это не имя файла, как же фреймворк Razor Pages узнает, какая модель страницы ассоциируется с данным файлом представления страницы Razor?

В верхней части каждой страницы Razor сразу после директивы `@page` идет директива `@model` с типом, указывающим, какая модель страницы ассоциируется с представлением Razor. Например, следующие директивы указывают на то, что `ToDoItemModel` – это модель страницы, связанная с этой страницей Razor:

```
@page  
@model ToDoItemModel
```

Как только запрос маршрутизируется на страницу Razor, как мы рассмотрели в главе 5, фреймворк ищет директиву `@model`, чтобы решить, какую модель страницы использовать. Основываясь на выбранной модели страницы, затем он выполняет привязку к любым свойствам модели, отмеченным атрибутом `[BindProperty]` (о чем шла речь в главе 16), и выполняет соответствующий обработчик страницы (на основе HTTP-метода запроса, как описано в главе 15).

ПРИМЕЧАНИЕ Формально говоря, модель страницы и директива `@model` не являются обязательными. Если вы не укажете модель страницы, фреймворк выполнит обработчик страницы по умолчанию, как показано в главе 5. Также можно объединить файлы `.cshtml` и `.cshtml.cs` в один файл с расширением `.cshtml`. Подробнее об этом подходе можно прочитать в книге Майка Бринда «*Razor Pages in Action*» (Manning, 2022).

Помимо директив `@page` и `@model`, файл представления Razor содержит шаблон Razor, который выполняется для генерации ответа в виде HTML-кода.

17.2.2 Знакомство с шаблонами Razor

Шаблоны представлений Razor содержат смесь кода HTML и C#, перемежающихся друг с другом. Разметка HTML позволяет с легкостью

описать, что именно следует отправлять в браузер, тогда как код C# можно использовать для динамического изменения того, что визуализируется. Например, в следующем листинге показан пример того, как Razor визуализирует список строк, обозначающих задачи.

Листинг 17.2. Шаблон Razor для отображения списка строк

```
@page
{
    var tasks = new List<string>
    { "Buy milk", "Buy eggs", "Buy bread" };
}

<h1>Tasks to complete</h1> ←
<ul>
@for(var i = 0; i < tasks.Count; i++)
{
    var task = tasks[i];
    <li>@i - @task</li>
}
</ul>
```

Произвольный код C# может быть выполнен в шаблоне. Переменные остаются в области видимости на всей странице

Стандартная HTML-разметка будет отображаться на выходе без изменений

Смешивание C# и HTML позволяет динамически создавать HTML во время выполнения

Разделы чистого HTML в этом шаблоне заключены в угловые скобки. Движок Razor копирует этот код прямо в вывод без изменений, как если бы вы писали обычный HTML-файл.

ПРИМЕЧАНИЕ Возможность синтаксиса Razor знать, когда вы переключаетесь между HTML и C#, может быть поразительной и раздражающей одновременно. Подробности того, как управлять этим переходом, обсуждаются в разделе 17.3.

Помимо HTML-кода, здесь также можно увидеть ряд инструкций C#. Преимущество иметь возможность, например, использовать цикл `for`, вместо того чтобы явно выписывать каждый элемент ``, должно быть очевидным. Мы подробнее рассмотрим функции C# в Razor в следующем разделе. При отрисовке шаблон из листинга 17.2 даст следующий HTML-код.

Листинг 17.3. Вывод HTML, полученный при отрисовке шаблона Razor

```
<h1>Tasks to complete</h1>
<ul>
    <li>0 - Buy milk</li>
    <li>1 - Buy eggs</li>
    <li>2 - Buy bread</li>
</ul> ←
```

HTML из шаблона Razor записывается непосредственно в вывод

Элементы генерируются динамически циклом for на основе предоставленных данных

HTML из шаблона Razor записывается непосредственно в вывод

Как видите, окончательный вывод шаблона Razor после отрисовки – это простой HTML-код. Не осталось ничего сложного, просто разметка HTML, которую можно отправить в браузер и визуализировать. На рис. 17.3 показано, как это будет отображать браузер.

Данные для отображения определены в C#

```
var tasks = new List<string>
{
    "Buy milk",
    "Buy eggs",
    "Buy bread"
}
```

Разметка Razor описывает, как отображать эти данные, используя смесь HTML и C#

```
<h1>Tasks to complete</h1>
<ul>
@for(var i=0; i<tasks.Count; i++)
{
    var task = tasks[i];
    <li>@i - @task</li>
}
</ul>
```

Совмещая данные объектов C# и разметку Razor, HTML может создаваться динамически, вместо того чтобы быть созданным в момент компиляции программы

Рис. 17.3 Шаблоны Razor можно использовать для динамической генерации HTML-кода во время выполнения из объектов C#. В данном случае цикл for используется для создания повторяющихся элементов ``

В данном примере значения списка вшиты в код для простоты – динамических данных не было. Это часто бывает с простыми страницами Razor, например с тем, что у вас может быть на домашней странице, – вам нужно отображать почти статичную страницу. Что касается остальной части приложения, то здесь гораздо чаще будут присутствовать некие данные, которые нужно отобразить, обычно предоставляемые как свойства модели страницы.

17.2.3 Передача данных в представления

В ASP.NET Core есть несколько способов передачи данных из обработчика страницы на странице Razor в ее представление. Какой подход лучше, будет зависеть от данных, которые вы пытаетесь передать, но в целом нужно использовать механизмы в следующем порядке:

- *свойства модели страницы* – обычно вы должны предоставлять любые данные, которые должны отображаться как свойства модели страницы. Таким образом должны быть представлены все данные, относящиеся к связанному представлению Razor. Как вы вскоре увидите, объект модели страницы доступен в представлении во время визуализации;
- *ViewData* – это словарь объектов со строковыми ключами, которые можно использовать для передачи произвольных данных из обработчика страницы в представление. Кроме того, он позволяет передавать данные в файлы макета, как вы увидите в разделе 17.4. Это основная причина использовать *ViewData*, вместо того чтобы задавать свойства модели страницы;
- *TempData* – это словарь объектов со строковыми ключами, аналогичный *ViewData*, который хранится до тех пор, пока не будет прочитан в другом запросе. Обычно он используется для временного сохранения данных при использовании паттерна POST-

REDIRECT-GET. По умолчанию TempData хранит данные в зашифрованном файле cookie, но доступны и другие варианты хранения, как описано в документации: <http://mng.bz/Wzx1>;

- `HttpContext` – технически объект `HttpContext` доступен как в обработчике страницы, так и в представлении Razor, поэтому его можно было бы использовать для передачи данных между ними. Но не нужно – в этом нет необходимости, поскольку в нашем распоряжении имеются другие доступные методы;
- `сервисы @inject` – вы можете использовать внедрение зависимостей, чтобы сделать сервисы доступными в представлениях, хотя обычно это следует использовать очень экономно. Внедрение зависимостей и директивы `@inject` описаны в главе 10.

Безусловно, лучший подход для передачи данных из обработчика страницы в представление – использовать свойства модели страницы. В самих свойствах нет ничего особенного; вы можете хранить там все, что вам нужно.

ПРИМЕЧАНИЕ Многие фреймворки имеют концепцию контекста данных для привязки компонентов пользовательского интерфейса. Модель страницы представляет собой аналогичную концепцию, поскольку содержит значения для отображения в пользовательском интерфейсе, но привязка является только односторонней; модель страницы предоставляет значения пользовательскому интерфейсу, и как только интерфейс будет создан и отправлен в качестве ответа, модель страницы уничтожается.

Как написано в разделе 17.2.1, директива `@model` в верхней части представления Razor описывает, какой тип модели страницы ассоциируется с данной страницей Razor. Модель страницы, связанная со страницей Razor, содержит один или несколько обработчиков страниц и предоставляет данные в качестве свойств для использования в представлении Razor, как показано в следующем листинге.

Листинг 17.4. Представление данных в качестве свойств в PageModel

```
public class ToDoItemModel : PageModel
{
    public List<string> Tasks { get; set; }
    public string Title { get; set; }

    public void OnGet(int id)
    {
        Title = "Tasks for today";
        Tasks = new List<string>
        {
            "Get fuel",
            "Check oil",
            "Check tyre pressure"
        };
    }
}
```

Доступ к общедоступным свойствам можно получить из представления Razor

Модель страницы передается в представление Razor при его выполнении

Создание необходимых данных: обычно здесь вызывается сервис или база данных для загрузки данных

Можно получить доступ к самому экземпляру модели страницы из представления Razor, используя свойство `Model`. Например, чтобы отобразить свойство `Title` класса `ToDoItemModel` в представлении Razor, нужно использовать `<h1>@Model.Title</h1>`. Так вы визуализируете строку, указанную в свойстве `ToDoItemModel.Title`, создавая код: `<h1>Tasks for today</h1>`.

СОВЕТ Обратите внимание, что директива `@model` должна находиться в верхней части представления сразу после директивы `@page` и что в ней строчная буква `m`. К свойству `Model` можно получить доступ в любом месте представления, и оно имеет прописную букву `M`.

В подавляющем большинстве случаев использование публичных свойств модели страницы – верный путь; это стандартный механизм передачи данных между обработчиком страницы и представлением. Но в некоторых случаях свойства модели страницы могут быть не самым лучшим вариантом. Это часто бывает, когда вы хотите передавать данные между макетами представления. Вы увидите, как это работает, в разделе 17.4.

Типичный пример – заголовок страницы. Вам необходимо указать заголовок для каждой страницы в приложении, поэтому вы могли бы создать базовый класс со свойством `Title` и сделать так, чтобы каждая модель страницы наследовала от него. Но это будет выглядеть очень громоздко, поэтому обычный подход в этой ситуации заключается в использовании коллекции `ViewData` для передачи данных.

Фактически стандартные шаблоны страниц Razor используют такой подход по умолчанию, задавая значения в словаре `ViewData` из самого представления:

```
@{
    ViewData["Title"] = "Home Page";
}
<h2>@ViewData["Title"].</h2>
```

Этот шаблон задает для ключа `"Title"` в словаре `ViewData` значение `"Home Page"`, а затем извлекает значение по ключу для визуализации его в шаблоне. Все это может показаться излишним, но поскольку словарь `ViewData` используется во всем запросе, он делает заголовок страницы доступным в макетах, как вы увидите позже. После визуализации предыдущий шаблон даст следующий вывод:

```
<h2>Home Page.</h2>
```

Также можно задать значения в словаре `ViewData` из обработчиков страниц двумя разными способами, как показано в следующем листинге.

Листинг 17.5 Задаем значения ViewData с помощью атрибута

```
public class IndexModel: PageModel
{
    [ViewData]
```



Свойства, отмеченные атрибутом `[ViewData]`, задаются в `ViewData`

```

public string Title { get; set; }

public void OnGet()
{
    Title = "Home Page"; ← Значение ViewData["Title"]
    ViewData["Subtitle"] = "Welcome"; ← будет задано как "Home Page"
}

```

Можно задать ключи напрямую в словаре ViewData

Вы можете отобразить значения в шаблоне так же, как и раньше:

```
<h1>@ViewData["Title"]</h3>
<h2>@ViewData["Subtitle"]</h3>
```

СОВЕТ Я не считаю атрибут [ViewData] особенно полезным, но это еще одна особенность, на которую стоит обратить внимание. Я создаю набор глобальных статических констант для всех ключей ViewData и ссылаюсь на них, вместо того чтобы постоянно набирать "Title". У вас будет IntelliSense, когда вы будете работать со значениями. Они будут безопасны для рефакторинга, и вы избежите труднозаметных опечаток.

Как я упоминал ранее, помимо свойств модели страницы и ViewData, существуют и другие механизмы, которые можно использовать для передачи данных, но только эти два я использую сам, поскольку с ними можно делать все, что вам нужно. Напоминаю: по возможности всегда используйте свойства модели страницы, так вы получаете выгоду от строгой типизации и IntelliSense. Возвращайтесь к ViewData, только если речь идет о значениях, которые должны быть доступны *за пределами* представления Razor.

Вы немного познакомились с возможностями, доступными вам в шаблонах Razor, но в следующем разделе мы подробнее рассмотрим некоторые доступные возможности C#.

17.3 Создание динамических веб-страниц с помощью Razor

Вероятно, вам будет приятно узнать, что практически все, что вы можете делать в C#, возможно в синтаксисе Razor. За кулисами файлы с расширением .cshtml компилируются в обычный код C# (используя тип `string` для блоков чистого HTML), поэтому какое бы странное и чудесное поведение вам ни понадобилось, его можно создать!

Говоря это, следует учитывать: то, что вы *можете* что-то делать, не означает, что вы *должны* это делать. Вам будет намного легче работать с файлами и сопровождать их, если они будут как можно проще. Это касается почти всего в программировании, но я считаю, что особенно это касается шаблонов Razor.

В этом разделе рассматриваются некоторые наиболее распространенные конструкции C#, которые вы можете использовать. Если вы

обнаружите, что вам нужно нечто более экзотическое, обратитесь к документации по синтаксису Razor: <http://mng.bz/8rMw>.

17.3.1 Использование кода C# в шаблонах Razor

Одним из наиболее распространенных требований при работе с шаблонами Razor является отрисовка значения, вычисленного в C#, в HTML. Например, у вас может возникнуть желание вывести текущий год, который будет отображаться рядом со знаком авторского права в HTML-коде:

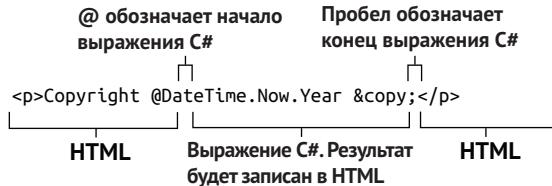
```
<p>© Copyright 2022</p>
```

Или вывести результат сложения:

```
<p>The sum of 1 and 2 is <i>3</i></p>
```

Можно сделать это двумя способами, в зависимости от того, какой именно код C# нужно выполнить. Если код представляет собой однократную инструкцию, то можно использовать символ @, чтобы указать, что вы хотите записать результат в вывод HTML, как показано на рис. 17.4. Вы уже видели, как это используется для записи значений из модели страницы или ViewData.

Рис. 17.4 Запись результата выражения C# в HTML. Символ @ указывает, где начинается код C#, а выражение заканчивается в конце инструкции, в данном случае на пробеле



Если код C#, который вы хотите выполнить, требует пробела, то нужно использовать круглые скобки для разделения, как показано на рис. 17.5.

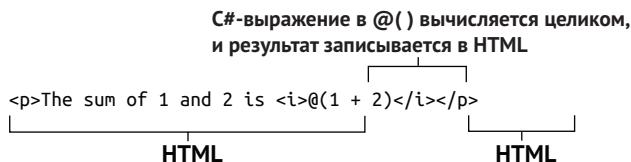


Рис. 17.5 Если выражение C# содержит пробелы, нужно заключить его в круглые скобки, используя @(), чтобы движок Razor знал, где заканчивается C# и начинается HTML

Эти два подхода, в которых код C# вычисляется и записывается непосредственно в вывод HTML, называются *выражениями Razor*.

СОВЕТ Если вы хотите написать литерал @, а не выражение C#, используйте второй символ @: @@.

Иногда нужно выполнить некий код C#, но не нужно выводить значения. Мы использовали эту технику, когда задавали значения в ViewData:

```
@{  
    ViewData["Title"] = "Home Page";  
}
```

В этом примере демонстрируется *блок кода Razor*, который представляет собой обычный код C#, идентифицируемый конструкцией @{}. Здесь ничего не записывается в вывод HTML; все компилируется, как если бы вы написали это в любом другом файле C#.

СОВЕТ Когда вы выполняете код в блоках кода, он должен быть допустимым, поэтому нужно добавлять точки с запятой. И наоборот, когда вы записываете значения непосредственно в ответ с помощью выражений Razor, они вам не нужны. Если в вашем HTML-коде на выходе неожиданно что-то пошло не так, следите за отсутствующими или лишними точками с запятой.

Выражения Razor – один из наиболее распространенных способов записи данных из модели страницы в вывод HTML. В следующей главе вы увидите другой подход, при котором используются тег-хелперы. Однако возможности Razor простираются гораздо дальше, в чем вы убедитесь, прочитав следующий раздел, в котором узнаете, как включать традиционные структуры C# в свои шаблоны.

17.3.2 Добавление циклов и условий в шаблоны Razor

Одно из самых больших преимуществ использования шаблонов Razor по сравнению со статическим HTML – возможность динамически генерировать выходные данные. Возможность записывать значения из модели страницы в HTML с помощью выражений Razor является ключевой частью этого процесса, но есть еще один распространенный вариант использования – циклы и условные выражения. С их помощью можно, например, скрыть разделы пользовательского интерфейса или создать HTML-код для каждого элемента в списке.

Циклы и условные выражения включают такие конструкции, как циклы if и for. Их использование в шаблонах Razor почти идентично C#, но необходимо добавить к ним символ @. Если вы еще не освоили Razor, то в случае сомнений добавляйте еще один символ @!

Одним из весомых преимуществ Razor в контексте ASP.NET Core является тот факт, что он использует языки, с которыми вы уже знакомы: C# и HTML. Нет необходимости изучать новый набор примитивов для какого-либо другого языка шаблонов: это те же самые конструкции if, foreach и while, с которыми вы уже знакомы. А когда они вам не нужны, вы пишете чистый HTML-код, чтобы точно видеть, что пользователь получит в своем браузере.

В листинге 17.6 я применил ряд этих техник в шаблоне для отображения списка дел. Модель страницы имеет свойство IsComplete, а также свойство List<string> с именем Tasks, которое содержит все невыполненные задачи.

Листинг 17.6. Шаблон Razor для отрисовки ToDoItemViewModel

```

@page
@model ToDoItemModel
<div>
    @if (Model.IsComplete)
    {
        <strong>Well done, you're all done!</strong>
    }
    else
    {
        <strong>The following tasks remain:</strong>
        <ul>
            @foreach (var task in Model.Tasks)
            {
                <li>@task</li>
            }
        </ul>
    }
</div>

```

Директива @model указывает тип модели страницы в Model

Управляющая конструкция if проверяет значение свойства IsComplete модели страницы во время выполнения

Конструкция foreach будет генерировать элементы один раз для каждой задачи в Model.Tasks

Выражение Razor используется для записи задачи в выходные данные HTML

Этот код определенно соответствует тому, о чем мы говорили ранее: речь идет о смешении C# и HTML. Существуют традиционные управляющие конструкции C#, такие как `if` и `foreach`, которые вы ожидаете увидеть в любой обычной программе, с вкраплениями разметки HTML, которую вы хотите отправить в браузер. Вы видите, что символ `@` используется для того, чтобы обозначить, когда вы запускаете управляющую конструкцию, но в целом вы позволяете шаблону Razor самому делать вывод, где код HTML, а где C#.

В шаблоне показано, как генерировать динамический HTML-код во время выполнения в зависимости от предоставленных точных данных. Если у модели есть невыполненные задачи, HTML сгенерирует элемент списка для каждой задачи, давая на выходе примерно такой результат, как показано на рис. 17.6.

IntelliSense и поддержка инструментов

Возможно, смесь C# и HTML трудно читать, и это обоснованная претензия. Это еще один веский аргумент в пользу того, чтобы шаблоны Razor были как можно проще.

К счастью, если вы используете такие редакторы, как Visual Studio или Visual Studio Code, они могут помочь. Как видно на этом рисунке, код C# затенен, чтобы его легче было отличить от окружающего его HTML-кода.

Visual Studio затеняет код C# и подсвечивает символы @, где C# переходит в HTML. Это упрощает чтение шаблонов Razor.

```

1  @page "{id}"
2  @model ToDoList.Pages.ViewToDoModel
3
4  <p>
5      @if (Model.ToDo.IsComplete)
6      {
7          <strong>Well done, you're all done!</strong>
8      }
9      else
10     {
11         <strong>The following tasks remain:</strong>
12         <ul>
13             @foreach (var task in Model.ToDo.Tasks)
14             {
15                 <li>@task</li>
16             }
17         </ul>
18     }
19 </p>
20

```

Хотя возможность использовать циклы и условные выражения – мощное средство, это одно из преимуществ Razor перед статическим HTML – они также усложняют ваше представление. Постарайтесь ограничить количество логики в представлениях, чтобы сделать их максимально простыми для понимания и сопровождения.

Только подходящий блок if отображается в HTML, а содержимое в цикле foreach отображается в HTML по одному разу для каждого элемента

Данные для отображения определены в свойствах PageModel

```

Model.IsComplete = false;
Model.Tasks = new List<string>
{
    "Get fuel",
    "Check oil",
    "Check Tyre pressure"
};

```

Разметка Razor может включать выражения C#, такие как условные операторы if и циклы for

```

@if (Model.IsComplete)
{
    <p>Well done, you're all done!</p>
} else {
    <p>The following tasks remain:</p>
    <ul>
        @foreach(var task in Model.Tasks)
        {
            <li>@task</li>
        }
    </ul>
}

```

Рис. 17.6 Шаблон Razor создает элемент `` для каждой оставшейся задачи в зависимости от данных, передаваемых в представление во время выполнения. Вы можете использовать блок `if` для визуализации совершенно другого HTML-кода в зависимости от значений в вашей модели

Распространенный подход членов команды ASP.NET Core состоит в том, что они пытаются гарантировать, что при создании приложения вы можете пользоваться всеми преимуществами данного фреймворка без дополнительных настроек. Это соотносится с идеей, согласно которой по умолчанию *самый простой* способ сделать что-то должен быть *правильным*. Это отличная философия, поскольку это означает, что вы не должны обжечься, например, на проблемах, связанных с безопасностью, если следуете стандартным подходам. Однако иногда вам может потребоваться выйти за безопасные рамки. Вот распространенный вариант использования – вам нужно визуализировать в выводе HTML-код, содержащийся в объекте C#. Вы увидите это в следующем разделе.

17.3.3 Отрисовка HTML-кода с помощью метода `Raw`

В предыдущем примере мы визуализировали список задач в HTML, написав строковую задачу с использованием выражения Razor, `@task`. Но что, если переменная `task` содержит HTML-код, который вы хотите отобразить, то есть вместо `"Check oil"` она содержит `"Check oil"`? Если вы используете выражение Razor для вывода, как делали это раньше, то, возможно, надеетесь получить это:

```
<li><strong>Check oil</strong></li>
```

Но нет. Сгенерированный HTML-код выглядит так:

```
<li>&lt;strong&gt;Check oil&lt;/strong&gt;</li>
```

Хм, выглядит странно, правда? Что же здесь произошло? Почему шаблон не записал вашу переменную в HTML-код, как в предыдущих примерах? Если вы посмотрите, как браузер отображает этот HTML-код на рис. 17.7, тогда, надеюсь, это будет иметь больше смысла.

Шаблоны Razor кодируют выражения C# до того, как они будут записаны в выходной поток. Это делается в первую очередь из соображений безопасности; запись произвольных строк в ваш HTML-код может позволить пользователям внедрять вредоносные данные и код JavaScript на ваш сайт. Следовательно, переменные C#, которые вы выводите в шаблоне Razor, записываются как значения в кодировке HTML.

ПРИМЕЧАНИЕ Razor также отображает символы Unicode, отличные от ASCII, например ó и è, в виде HTML-сущностей: ó и è. Такое поведение можно настроить с помощью класса `WebEncoderOptions` из файла `Program.cs`, как в этом примере: `builder.Services.Configure<WebEncoderOptions>(o => o.AllowCharacter('ó'))`.

В некоторых случаях вам может потребоваться напрямую записать HTML-код, содержащийся в строке, в ответ. Если вы оказались в такой ситуации, сначала остановитесь. Вам *действительно* нужно это делать? Если значения, которые вы пишете, были введены пользователем или созданы на основе значений, предоставленных пользователями, то вы рискуете создать брешь в системе безопасности своего сайта.

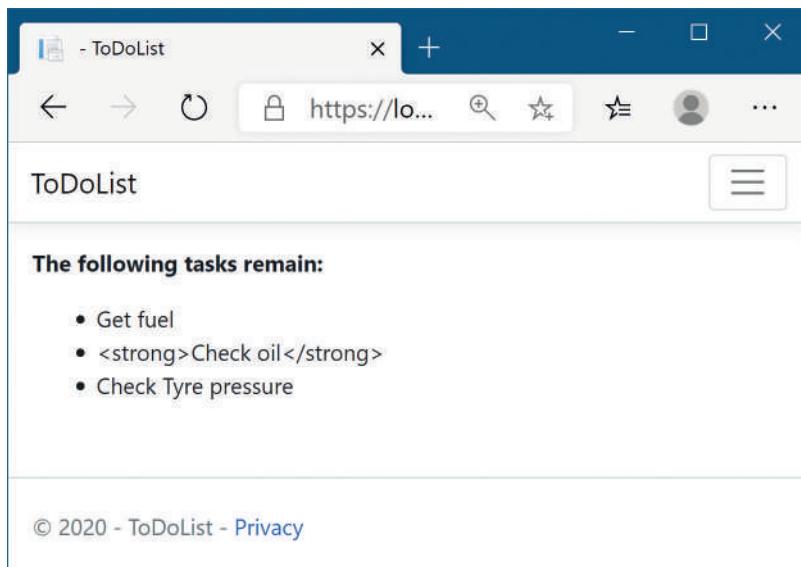


Рис. 17.7 Второй элемент, "Check oil", был в кодировке HTML, поэтому элементы видны пользователю как часть задачи. Это позволяет избежать проблем с безопасностью, поскольку пользователи не смогут внедрять вредоносные скрипты в ваш HTML-код

Если вам действительно нужно записать переменную в HTML-поток, это можно сделать, используя свойство `Html` на странице представления и вызвав метод `Raw`:

```
<li>@Html.Raw(task)</li>
```

При таком подходе строка в `task` будет напрямую записана в выходной поток, в результате чего вы получите HTML-код, который изначально был вам нужен: `Check oil`, как показано на рис. 17.8.

ВНИМАНИЕ Использование `Html.Raw` таким способом создает угрозу для безопасности, которую пользователи могут использовать для внедрения вредоносного кода на ваш сайт. По возможности избегайте этого.

Конструкции C#, показанные в данном разделе, могут быть полезны, но они могут привести к тому, что ваши шаблоны будут трудны для понимания. Как правило, легче понять назначение шаблонов Razor, которые преимущественно представляют собой HTML-разметку, а не код C#.

В предыдущей версии ASP.NET эти конструкции и, в частности, вспомогательное свойство `Html` были стандартным способом создания динамической разметки. Данный подход по-прежнему можно применять в ASP.NET Core, используя различные методы `HtmlHelper` свойства `Html`, но они в значительной степени заменены более совершенной техникой: тег-хелперами.

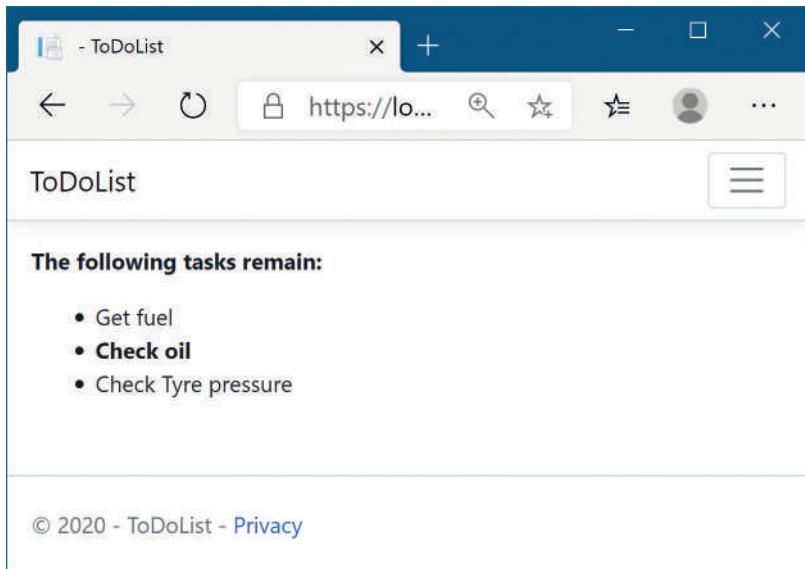


Рис. 17.8 Второй элемент, "Check oil", был выведен с использованием `Html.Raw()`, поэтому он не был закодирован в HTML. Элементы приводят к тому, что второй элемент отображается жирным шрифтом. По возможности следует избегать использования `Html.Raw()` таким способом, поскольку это угроза безопасности

ПРИМЕЧАНИЕ Я расскажу о тег-хелперах и о том, как их использовать для создания HTML-форм, в следующей главе. `HtmlHelper` по сути устарел, хотя он все еще доступен, если вы предпочитаете его использовать.

Тег-хелперы – полезная функция, впервые появившаяся в Razor, но ряд других функций были перенесены из предыдущей версии ASP.NET. В следующем разделе этой главы вы увидите, как создавать вложенные шаблоны Razor и использовать частичные представления, чтобы уменьшить число дублирований в представлениях.

17.4 Макеты, частичные представления и _ViewStart

В этом разделе вы узнаете о макетах и частичных представлениях, которые позволяют извлекать общий код для уменьшения дублирования. Эти файлы упрощают внесение изменений в HTML-код, который влияет на несколько страниц одновременно. Вы также узнаете, как выполнять общий код для каждой страницы Razor с помощью `_ViewStart` и `_ViewImports`, а также как включать дополнительные разделы в свои страницы. Каждый HTML-документ имеет определенное количество необходимых элементов: `<html>`, `<head>` и `<body>`. Кроме того, часто есть общие разделы, которые повторяются на каждой странице вашего приложения, такие как верхний и нижний колонтитулы, как показано

на рис. 17.9. На каждой странице вашего приложения также, вероятно, будут присутствовать ссылки на одни и те же файлы CSS и JavaScript.

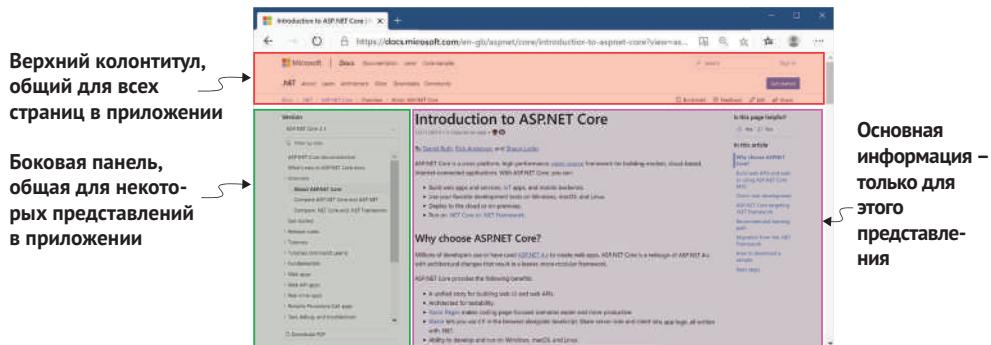


Рис. 17.9 Типичное веб-приложение имеет блочный макет, где некоторые блоки являются общими для каждой страницы приложения. Верхний колонтитул, вероятно, будет одинаковым для всего вашего приложения, но боковая панель может быть идентичной только для страниц в одном разделе. Содержимое тела будет отличаться для каждой страницы приложения

Такое количество элементов – просто кошмар, когда речь идет о сопровождении. Если бы вам пришлось вручную включать их в каждое представление, то внесение любых изменений превратилось бы в трудоемкий и подверженный ошибкам процесс, связанный с редактированием каждой страницы. Вместо этого Razor позволяет извлекать эти общие элементы в *макеты* (layouts).

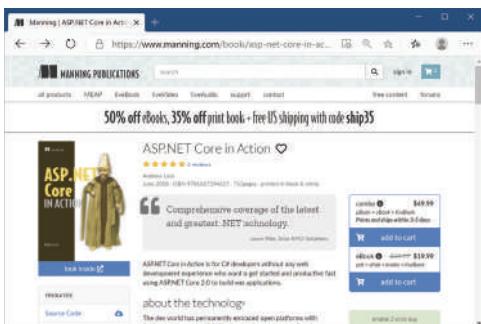
ОПРЕДЕЛЕНИЕ *Макет* в Razor – это шаблон, включающий в себя общий код. Его нельзя визуализировать напрямую, но можно делать это вместе с обычными представлениями Razor.

Располагая общую разметку в макетах, вы можете уменьшить дублирование в приложении, что упрощает внесение изменений, управление и сопровождение представлений, и вообще это хорошая практика!

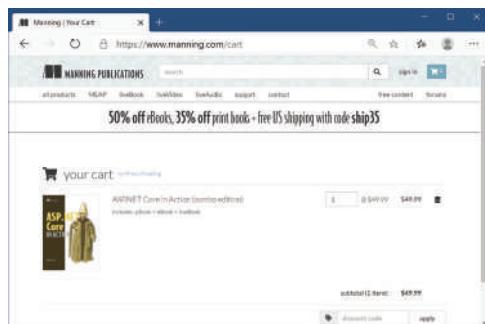
17.4.1 Использование макетов для общей разметки

Файлы макета – это по большей части обычные шаблоны Razor, содержащие разметку, общую для нескольких страниц. Приложение ASP.NET Core может иметь несколько макетов, и макеты могут ссылаться на другие макеты. Обычно это используется для макетов разных разделов вашего приложения. Например, веб-сайт для онлайн-торговли может использовать представление с тремя столбцами для большинства страниц и макет с одним столбцом, когда вы переходите на страницы оформления заказа, как показано на рис. 17.10.

Вы часто будете использовать макеты на разных страницах Razor, поэтому обычно они помещаются в папку Pages/Shared. Можете называть их как хотите, но существует общее соглашение об использовании _Layout.cshtml в качестве имени файла для базового макета в вашем приложении. Это имя по умолчанию, используемое шаблонами страниц Razor в Visual Studio и интерфейсе командной строки .NET.



Трехколоночный макет



Одноколоночный макет

Рис. 17.10 На сайте <https://manning.com> используются разные макеты для разных частей веб-приложения. На страницах товаров используется трехколоночный макет, а на странице корзины – одноколоночный

ПОДСКАЗКА Обычно к файлам макета добавляется символ подчеркивания (_), чтобы отличать их от стандартных шаблонов Razor в папке Pages. Размещение их в Pages/Shared означает, что к ним можно обращаться по короткому имени, например "_Layout", без необходимости указывать полный путь к файлу макета.

Файл макета похож на обычный шаблон Razor, за одним исключением: каждый макет должен вызывать функцию @RenderBody(). Так вы сообщаете движку шаблонов, куда вставить содержимое из дочерних представлений. Простой макет показан в следующем листинге. Обычно ссылки на все ваши файлы CSS и JavaScript будут находиться в макете, а также здесь будут содержаться все общие элементы, такие как верхний и нижний колонтитулы, но данный пример включает в себя почти минимальный HTML-код.

Листинг 17.7. Базовый файл _Layout.cshtml, вызывающий функцию RenderBody

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ ViewData["Title"]</title> ←
    <link rel="stylesheet" href="~/css/site.css" /> ←
</head>
<body>
    @RenderBody() ← | Сообщает механизму шаблонов, куда вставить содержимое дочернего представления
</body>
</html>
```

ViewData – это стандартный механизм передачи данных в макет из представления

Элементы, общие для каждой страницы, такие как CSS, обычно определяются в макете

Как видите, файл макета включает необходимые элементы, например <html> и <head>, а также элементы, которые нужны на каждой странице, такие как <title> и <link>. Этот пример также демонстрирует

преимущество хранения заголовка страницы в `ViewData`; макет может визуализировать его в элементе `<title>` так, чтобы он отображался на вкладке браузера, как показано на рис. 17.11.

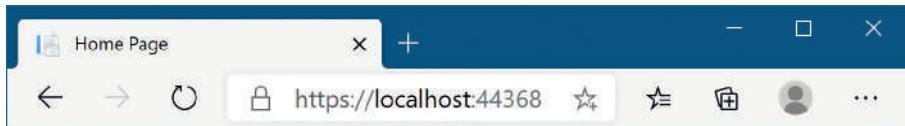


Рис. 17.11 Содержимое элемента `<title>` используется в качестве имени вкладки в браузере пользователя, в данном случае это домашняя страница

ПРИМЕЧАНИЕ Файлы макета не являются автономными страницами Razor и не участвуют в маршрутизации, поэтому они не начинаются с директивы `@page`.

Представления могут указать, какой файл макета использовать, задав свойство `Layout` внутри кодового блока Razor.

Листинг 17.8. Задаем свойство Layout из представления

```
@{  
    Layout = "_Layout"; } Задает для страницы  
    ViewData["Title"] = "Home Page"; } макет _Layout.cshtml.  
}  
<h1>@ViewData["Title"]</h1> ViewData – удобный способ  
<p>This is the home page</p> передачи данных из  
Содержимое в представлении Razor представления Razor в макет.  
для визуализации внутри макета.
```

Любое содержимое в представлении будет отображаться внутри макета, где вызывается функция `@RenderBody()`. Соедините два предыдущих листинга, и вы получите следующий HTML-код, который генерируется и отправляется пользователю.

Листинг 17.9 Визуализированный результат объединения представления и его макета

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <title>Home Page</title> ViewData, заданный в представлении,  
    <link rel="stylesheet" href="/css/site.css" /> используется для визуализации макета  
</head>  
<body>  
    <h1>Home Page</h1>  
    <p>This is the home page</p> Вызов RenderBody отображает  
</body>  
</html> содержимое представления
```

Разумное использование макетов может быть чрезвычайно полезным для уменьшения дублирования между страницами. По умолчанию макеты предоставляют только одно место, где можно отображать

содержимое из представления, при вызове функции `@RenderBody`. В тех случаях, когда это слишком ограничивает вас, можно визуализировать содержимое с помощью секций.

17.4.2 Переопределение родительских макетов с помощью секций

Когда вы начинаете использовать несколько макетов в своем приложении, то зачастую возникает необходимость визуализировать содержимое из дочерних представлений в нескольких местах вашего макета. Рассмотрим макет, в котором используются два столбца. Представлению нужен механизм, чтобы сказать: «визуализировать *это* содержимое в *левом столбце*» и «визуализировать *другое* содержимое в *правом столбце*». Для этого нужны *секции*.

ПРИМЕЧАНИЕ Помните, что все функции, описанные в этой главе, относятся только к Razor, который представляет собой движок для отрисовки на стороне сервера. Если вы используете фреймворк для создания одностраничных приложений на стороне клиента, то, вероятно, будете реализовывать эти требования другими способами.

Секции обеспечивают способ организации размещения элементов представления в макете. Они определяются в представлении с помощью определения `@section`, как показано в следующем листинге, который определяет HTML-содержимое боковой панели, отдельно от основного содержимого в секции `Sidebar`. `@section` можно разместить в любом месте файла, вверху или внизу, где удобно.

Листинг 17.10. Определение секции в шаблоне представления

```
@{  
    Layout = "_TwoColumn";  
}  
@section Sidebar {  
    <p>This is the sidebar content</p>  
}  
-> <p>This is the main content </p>
```

Все содержимое внутри фигурных скобок является частью раздела боковой панели, а не основного содержимого

Любой контент, не входящий в `@section`, будет отображаться с помощью вызова `@RenderBody`

Секция визуализируется в родительском макете с помощью вызова функции `@RenderSection()`. Так вы визуализируете содержимое, содержащееся в дочернем разделе, в макет. Секции могут быть обязательными либо нет. Если они обязательны, представление должно объявить данное определение `@section`; если они необязательны, их можно опустить, и макет пропустит их. Пропущенные секции не появятся в визуализированном HTML-коде. В следующем листинге показан макет, в котором есть обязательная секция `Sidebar` и необязательная секция `Scripts`.

Листинг 17.11 Визуализация секции в файле макета, _TwoColumn.cshtml

```

@{
    Layout = "_Layout"; ←
}
<div class="main-content">
    @RenderBody() ←
</div>
<div class="side-bar">
    @RenderSection("Sidebar", required: true) ←
    @RenderSection("Scripts", required: false) ←

```

Этот макет сам по себе вложен
в родительский макет

Отображает весь контент из представления,
которое не является частью раздела

Отрисовывает раздел «Скрипты»; если этот раздел
не определен в представлении, он игнорируется

Отрисовывает раздел боковой панели; если
этот раздел не опре-
делен в представле-
нии, выдает ошибку

СОВЕТ Обычно на страницах макета есть необязательная секция Scripts. Ее можно использовать для визуализации дополнительного кода JavaScript, который требуется для некоторых представлений, но не для каждого. Типичный пример – скрипты jQuery Unobtrusive Validation для валидации на стороне клиента. Если представлению нужны скрипты, оно добавляет соответствующий @section Scripts в разметку Razor.

Вы, наверное, заметили, что в предыдущем листинге определено свойство Layout, даже притом, что это макет, а не представление. Это вполне приемлемо и позволяет создавать вложенные иерархии макетов, как показано на рис. 17.12.

СОВЕТ Большинство веб-сайтов в наши дни должны быть «отзывчивыми», чтобы работать на самых разных устройствах. Обычно для этого не следует использовать макеты. Не нужно использовать разные макеты для одной страницы в зависимости от устройства, отправляющего запрос. Вместо этого используйте один и тот же HTML-код для всех устройств и CSS на стороне клиента, чтобы адаптировать отображение вашей веб-страницы по мере необходимости.

Помимо простого флага необязательный/обязательный для секций на страницах Razor есть несколько других сообщений, которые можно использовать для управления потоком на страницах макета:

- IsSectionDefined(string section) – возвращает true, если страница Razor определила данную секцию;
- IgnoreSection(string section) – игнорирует невизуализированную секцию. Если секция определена на странице, но не отображается, страница Razor генерирует исключение, если раздел не игнорируется;
- IgnoreBody() – игнорирует невизуализированное тело страницы Razor. Макеты должны вызывать либо функцию RenderBody(), либо функцию IgnoreBody(); в противном случае они выбросят исключение InvalidOperationException.

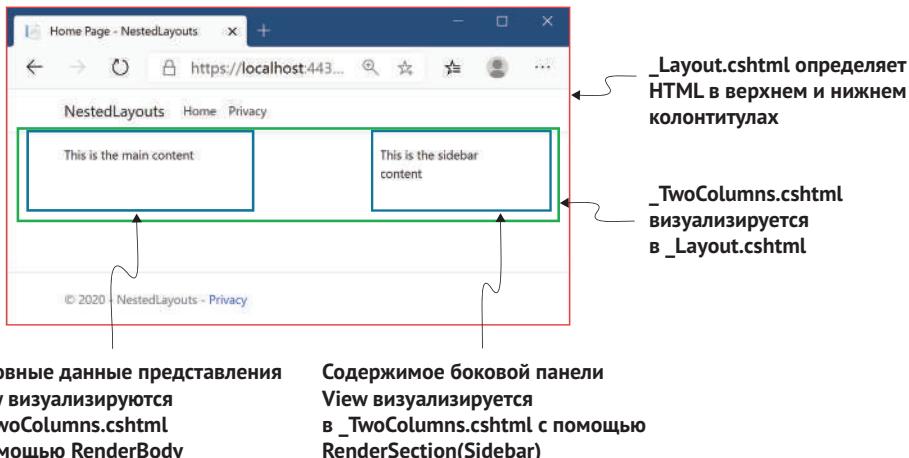


Рис. 17.12 Несколько макетов могут быть вложены для создания сложных иерархий. Это позволяет сохранить элементы, общие для всех представлений, в вашем базовом макете, и извлекать макет, общий для нескольких представлений, во вложенные макеты

Файлы макета и секции обеспечивают большую гибкость для создания сложных пользовательских интерфейсов, но одно из наиболее важных их применений – уменьшение дублирования кода в приложении. Они идеально подходят для предотвращения дублирования содержимого, которое вам пришлось бы писать для каждого представления. А как насчет тех случаев, когда вы обнаружите, что хотите повторно использовать часть представления в другом месте? Для этого есть частичные представления.

17.4.3 Использование частичных представлений для инкапсуляции разметки

Частичные представления, как и следует из их названия, являются частью представления. Они позволяют разбить крупное представление на более мелкие, переиспользуемые фрагменты. Это может быть полезно как для уменьшения сложности большого представления, разделяя его на несколько частичных представлений, так и для того, чтобы дать возможность повторно использовать часть представления внутри другого представления.

Большинство веб-фреймворков, использующих отрисовку на стороне сервера, обладают такой возможностью: в Ruby on Rails есть частичные представления, в Django – теги inclusion, а в Zend – Partial. Все они работают одинаково, извлекая общий код в небольшие повторно используемые шаблоны. Даже такие шаблонизаторы на стороне клиента, как Mustache и Handlebars, используемые такими фреймворками, как Angular и Ember, имеют аналогичные концепции «частичного представления».

Снова рассмотрим приложение со списком дел. Вы можете обнаружить, что у вас есть страница Razor, ViewToDo.cshtml, которая отображает один элемент списка с заданным идентификатором. Позже вы создаете новую страницу Razor, RecentTodos.cshtml, которая отобра-

жает пять последних дел. Вместо того чтобы копировать и вставлять код с одной страницы на другую, можно создать частичное представление _ToDo.cshtml, как показано в следующем листинге.

Листинг 17.12 Частичное представление _ToDo.cshtml для отображения ToDoItemViewModel

```
@model ToDoItemViewModel
<h2>@Model.Title</h2>
<ul>
    @foreach (var task in Model.Tasks)
    {
        <li>@task</li>
    }
</ul>
```

Частичные представления могут привязываться к данным в свойстве Model, как обычная страница Razor использует модель страницы

Содержимое частичного представления, которое ранее находилось в файле ViewToDo.cshtml

Частичные представления немного похожи на страницы Razor, только в них нет модели страницы и обработчиков. Они предназначены исключительно для визуализации небольших фрагментов HTML-кода, а не для обработки запросов, привязки и валидации модели, и вызова модели приложения. Они отлично подходят для инкапсуляции небольших полезных фрагментов HTML-кода, которые необходимо генерировать на нескольких страницах Razor.

И ViewToDo.cshtml, и RecentTodos.cshtml могут отображать частичное представление _ToDo.cshtml, которое имеет дело с генерацией HTML-кода для одного класса. Частичные представления отображаются с помощью тег-хелпера `<partial />`, предоставляющего имя представления для визуализации и данные (модель). Например, это можно сделать с помощью представления RecentTodos.cshtml, как показано в следующем листинге.

Листинг 17.13 Визуализация частичного представления со страницы Razor

```
@page
▷ @model RecentToDoListModel
    @foreach(var todo in Model.RecentItems)
    {
        <partial name="_ToDo" model="todo" />
    }
```

Модель страницы содержит список последних элементов для визуализации

Это страница Razor, поэтому используется директива @page. Частичные представления не используют ее

Перебираем недавние элементы. todo – это ToDoItemViewModel, как того требует частичное представление

Используем тег-хелпер partial для визуализации частичного представления _ToDo, передавая модель для визуализации

Когда вы визуализируете частичное представление без указания абсолютного пути или расширения файла, как, например, _ToDo в листинге 17.13, фреймворк пытается найти представление, выполняя поиск в папке Pages, начиная с страницы Razor, которая его вызвала. Например,

если ваша страница расположена в Pages/Agenda/ToDos/RecentTodos.chstml, фреймворк будет искать файл с именем _ToDo.chstml в следующих местах:

- Pages/Agenda/ToDos/ (текущая папка Razor Page);
- Pages/Agenda/;
- Pages/;
- Pages/Shared/;
- Views/Shared/.

Будет выбрано первое расположение, содержащее файл _ToDo.cshtml. Если вы включите расширение файла .cshtml, когда ссылается на частичное представление, фреймворк будет искать *только* в папке текущей страницы Razor. Кроме того, если вы укажете абсолютный путь к частичному представлению, например /Pages/Agenda/ToDo.cshtml, то это будет единственным местом, где фреймворк будет искать.

ПРИМЕЧАНИЕ Как и в большинстве страниц Razor, места поиска представляют собой соглашения, которые можно настроить. Если вы обнаружите необходимость, то можете настроить пути, как показано здесь: <http://mng.bz/nM9e>.

Код Razor, содержащийся в частичном представлении, почти идентичен стандартному представлению. Основное отличие состоит в том, что частичные представления вызываются только из других представлений. Другое отличие состоит в том, что частичные представления не используют _ViewStart.cshtml при выполнении. Скоро вы это увидите.

ПРИМЕЧАНИЕ Как и у макетов, в именах частичных представлений в начале обычно идет подчеркивание.

Дочерние действия в ASP.NET Core

В предыдущей версии ASP.NET MVC существовала концепция дочернего действия. Это был метод действия, который можно было вызвать из представления. Это был основной механизм визуализации отдельных секций сложного макета, не имевший ничего общего с основным методом действия. Например, дочерний метод действия мог визуализировать корзину на сайте онлайн-магазина.

Такой подход означал, что вам не нужно было загрязнять каждую модель представления страницы элементами, необходимыми для визуализации корзины, но он в корне нарушил паттерн проектирования MVC, ссылаясь на контроллеры из представления.

В ASP.NET Core дочерних действий больше нет. Их заменили компоненты представления. Концептуально они очень похожи тем, что допускают выполнение произвольного кода и визуализацию HTML, но не вызывают действия контроллера напрямую. Можно рассматривать их как более мощное частичное представление, которое следует использовать везде, где частичное представление должно содержать значимый код или бизнес-логику. Вы увидите, как создать небольшой компонент представления, в главе 32.

Частичные представления – не единственный способ уменьшить дублирование в шаблонах представлений. Razor также позволяет переносить общие элементы, например объявления пространств имен и конфигурацию макета, в централизованные файлы. В следующем разделе вы увидите, как использовать эти файлы для очистки шаблонов.

17.4.4 Выполнение кода в каждом представлении с помощью _ViewStart и _ViewImports

Из-за природы представлений вы неизбежно обнаружите, что вы снова и снова пишете определенные вещи. Если все ваши представления используют один и тот же макет, то при добавлении следующего кода в верхнюю часть каждой страницы вы почувствуете, что это лишнее:

```
@{
    Layout = "_Layout";
}
```

Точно так же, если вы обнаружите, что вам нужно ссылаться на объекты из другого пространства имен в представлениях Razor, тогда добавление `@using WebApplication1.Models` в верхнюю часть каждой страницы может стать рутиной. К счастью, у ASP.NET Core есть два механизма, чтобы справиться с этими распространенными задачами: `_ViewImports.cshtml` и `_ViewStart.cshtml`.

Импорт общих директив с помощью _ViewImports

Файл `_ViewImports.cshtml` содержит директивы, которые будут вставляться в верхнюю часть каждого представления. Сюда входят такие директивы, как `@using` и `@model`, которые вы уже видели, – по сути, это любая директива Razor. Чтобы не добавлять конструкцию в каждое представление, можно включить ее в `_ViewImports.cshtml`, а не в страницы Razor.

Листинг 17.14 Типичный файл `_ViewImports.cshtml`, импортирующий дополнительные пространства имен

```
@using WebApplication1          | Пространство имен по умолчанию для
@using WebApplication1.Pages    | вашего приложения и папка Pages
@using WebApplication1.Models   |
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers           | Добавляет эту
                                                                | директиву, чтобы
                                                                | избежать ее раз-
                                                                |мещения в каждом
                                                                | представлении
                                                                |
                                                                | Делает доступными тег-хелперы в ваших пред-
                                                                |ставлениях, добавляется по умолчанию
```

Файл `_ViewImports.cshtml` можно поместить в любую папку, и он будет применяться ко всем представлениям и вложенным папкам. Как правило, он помещается в корневую папку `Pages` и применяется ко всем страницам Razor и частичному представлению в приложении.

Важно отметить, что в `_ViewImports.cshtml` нужно помещать только директивы Razor – нельзя помещать туда обычный код C#. Как видно из предыдущего листинга, вы ограничены такими конструкциями, как `@using` или директива `@addTagHelper`, о которых вы узнаете в следующей

главе. Если вы хотите выполнять произвольный C# код в начале каждого представления приложения, например чтобы задать свойство Layout, то должны использовать файл _ViewStart.cshtml.

Выполнение кода для каждого представления с помощью _ViewStart

Можно легко выполнить общий код в начале каждой страницы Razor, добавив файл _ViewStart.cshtml в папку Pages своего приложения. Этот файл может содержать любой код Razor, но обычно он используется, чтобы задать свойство Layout для всех страниц вашего приложения, как показано в следующем листинге. Затем можно опустить инструкцию Layout на всех страницах, использующих макет по умолчанию. Если для представления необходимо использовать макет, отличный от макета по умолчанию, можно переопределить его, задав значение на самой странице Razor.

Листинг 17.15 Типичный файл _ViewStart.cshtml, задающий макет по умолчанию

```
@{  
    Layout = "_Layout";  
}
```

Любой код в файле _ViewStart.cshtml выполняется до выполнения представления. Обратите внимание, что _ViewStart.cshtml запускается только для представлений Razor Page – он не запускается для макетов или частичных представлений. Также обратите внимание, что имена для этих специальных файлов Razor являются обязательными. Это не соглашения, которые можно изменить.

ВНИМАНИЕ! Вы должны использовать имена _ViewStart.cshtml и _ViewImports.cshtml, чтобы движок Razor мог их найти и правильно выполнить. Чтобы применить их ко всем страницам приложения, добавьте их в корень папки Pages, а не во вложенную папку Shared.

Можно указать дополнительные файлы _ViewStart.cshtml или _ViewImports.cshtml, чтобы они выполнялись для подмножества представлений, включив их во вложенную папку внутри Pages. Файлы во вложенных папках будут выполняться после файлов в корневой папке Pages.

Частичные представления, макеты и AJAX

В этой главе описывается, как использовать Razor для отрисовки полных HTML-страниц на стороне сервера, которые затем отправляются в браузер пользователя в традиционных веб-приложениях. Распространенным альтернативным подходом при создании веб-приложений является использование фреймворка JavaScript для создания одностраничного приложения (SPA) на стороне клиента, которое визуализирует HTML-код на стороне клиента в браузере.

Одной из технологий, которые обычно используют одностраничные приложения, является AJAX (асинхронный JavaScript и XML), когда браузер отправляет запросы в приложение ASP.NET Core, не перезагружая целиком новую страницу. Также можно использовать запросы AJAX с приложениями, применяющими отрисовку на стороне сервера. Для этого вы должны применять JavaScript, чтобы запрашивать обновление части страницы.

Если вы хотите использовать AJAX с приложением, использующим Razor, то следует рассмотреть возможность широкого применения частичных представлений. Тогда вы можете предоставлять к ним доступ с помощью дополнительных обработчиков страниц Razor, как показано в этой статье: <http://mng.bz/vzB1>. Использование AJAX может уменьшить общий объем данных, которые необходимо пересыпать между браузером и приложением, благодаря чему приложение будет работать более плавно и станет отзывчивее, поскольку не нужно будет загружать полностью столько страниц. Но использование AJAX с Razor может добавить сложности, особенно для больших приложений. Если вы уверены, что будете широко применять AJAX для создания отзывчивых веб-приложений, то можно рассмотреть возможность использования минимальных API или контроллеров веб-API и фреймворка для разработки приложений на стороне клиента или подумать об использовании Blazor.

На этом мы завершаем наш первый обзор отрисовки HTML-кода с использованием механизма шаблонов Razor. В следующей главе вы узнаете о тег-хелперах и о том, как их использовать для создания HTML-форм – основы современных веб-приложений. Тег-хелперы – это одно из самых больших улучшений Razor в ASP.NET Core по сравнению с устаревшим ASP.NET, поэтому знакомство с ними сделает редактирование представлений в целом более приятным занятием!

Резюме

- Razor – это язык шаблонов, позволяющий генерировать динамический HTML-код, используя смесь HTML и C#. Это представляет возможность использовать всю силу C# без необходимости вручную создавать ответ в виде HTML с использованием строк;
- Razor Pages может передавать строго типизированные данные в представление Razor, задавая публичные свойства в модели страницы. Чтобы получить доступ к свойствам модели представления, представление должно объявить тип модели с помощью директивы `@model`;
- обработчики страниц могут передавать пары «ключ–значение» в представление с помощью словаря `ViewData`. Это полезно для неявной передачи общих данных в макеты и частичные представления;
- выражения Razor визуализируют значения C# в выводе HTML с помощью символов `@` или `@()`. При использовании выражений Razor не нужно ставить точку с запятой после инструкции;

- блоки кода Razor, определенные с помощью @{}, выполняют код C# без вывода HTML. Блоки кода C# в Razor должны содержать полноценные инструкции, разделенные точкой с запятой;
- циклы и условные выражения можно использовать для простой генерации динамического HTML-кода в шаблонах, но рекомендуется, в частности, ограничить количество операторов if, чтобы представления можно было легко читать;
- если вам нужно визуализировать строку как чистый HTML-код, можно использовать Html.Raw, но делайте это осторожно – отображение чистого пользовательского ввода может привести к возникновению бреши в системе безопасности приложения;
- тег-хелперы позволяют привязать модель данных к элементам HTML, благодаря чему становится проще генерировать динамический HTML-код, сохраняя при этом удобство редактирования;
- в макете можно разместить общий HTML-код для нескольких представлений. Макет будет отображать любое содержимое из дочернего представления в месте, где вызывается функция @RenderBody;
- инкапсулируйте часто используемые фрагменты кода Razor в частичном представлении. Частичное представление можно визуализировать с помощью тега <partial>;
- файл _ViewImports.cshtml можно использовать для включения общих директив, таких как инструкции @using, во всех представлениях;
- файл _ViewStart.cshtml вызывается перед выполнением каждой страницы Razor и может использоваться для выполнения кода, общего для всех страниц Razor, например для установки страницы макета по умолчанию. Он не выполняется для макетов или частичных представлений;
- _ViewImports.cshtml и _ViewStart.cshtml являются иерархически – файлы в корневой папке выполняются первыми, а за ними следуют файлы в папках представления для конкретного контроллера.

18

Создание форм с помощью тег-хелперов

В этой главе:

- простое создание форм с помощью тег-хелперов;
- генерация URL-адресов с помощью Anchor Tag Helper;
- использование тег-хелперов для добавления функций в Razor.

В предыдущей главе вы узнали о шаблонах Razor и о том, как использовать их для создания представлений для своего приложения. Смешивая HTML и C#, можно создавать динамические приложения, которые могут отображать разные данные в зависимости от запроса, выполнившего вход пользователя, или любых других данных, к которым вы можете получить доступ.

Отображение динамических данных – важный аспект многих веб-приложений, но обычно это только половина дела. Помимо отображения данных пользователю, часто требуется, чтобы пользователь мог отправлять данные обратно в приложение. Вы можете использовать данные для настройки представления или чтобы обновить модель приложения, например сохранив ее в базе данных. В случае с традиционными веб-приложениями эти данные обычно отправляются с использованием HTML-формы.

В главе 16 вы узнали о привязке модели, то есть о том, как *принимать* данные, отправленные пользователем в запросе, и конверти-

ровать их в объекты C#, которые можно использовать на страницах Razor. Вы также узнали, что такое валидация и как важно проверять данные, отправленные в запросе. Мы использовали атрибуты `DataAnnotations` для определения правил, связанных с моделями, а также другие метаданные, например отображаемое имя свойства.

Последний аспект, который мы еще не рассмотрели, – это создание HTML-форм, которые пользователи используют для отправки этих данных в запросе. Формы – один из основных способов взаимодействия пользователей с приложением в браузере, поэтому важно, чтобы они были правильно определены и удобны для пользователя. Для этой цели ASP.NET Core предоставляет функцию под названием тег-хелперы.

Тег-хелперы – новинка от ASP.NET Core. Это дополнения к синтаксису Razor, которые можно использовать для настройки HTML-кода, генерируемого в шаблонах. Тег-хелперы можно добавить к стандартному элементу HTML, например `<input>`, для настройки его атрибутов на основе модели C#, избавляя себя от необходимости писать шаблонный код. Они также могут быть автономными элементами и использоваться для генерации полностью настраиваемого HTML-кода.

ПРИМЕЧАНИЕ Помните, что Razor, а следовательно, и тег-хелперы предназначены для отрисовки HTML-кода на стороне сервера. Тег-хелперы нельзя использовать напрямую в таких фреймворках, как Angular или React.

Если вы использовали предыдущую версию ASP.NET, то тег-хелперы могут напомнить вам HTML-хелперы, которые также можно применять для генерации HTML-кода на основе классов C#. Тег-хелперы являются логическим преемником HTML-хелперов, поскольку обеспечивают более упрощенный синтаксис по сравнению с предыдущими вариантами, ориентированными на C#. HTML-хелперы по-прежнему доступны в ASP.NET Core, поэтому если вы конвертируете некоторые старые шаблоны в ASP.NET Core, то можете использовать их в своих шаблонах, но я не буду рассматривать их в этой книге.

В данной главе вы узнаете, как использовать тег-хелперы при создании форм. Они упрощают процесс создания правильных имен и идентификаторов элементов, поэтому привязка модели может происходить без проблем при отправке формы обратно в приложение. Чтобы рассматривать их в контексте, мы продолжим создание приложения конвертера валют, которое мы видели в предыдущих главах. Мы добавим возможность отправлять ему запросы на обмен валюты, проверять данные и повторно отображать ошибки в форме с помощью тег-хелперов, которые сделают всю работу за нас, как показано на рис. 18.1.

По мере разработки приложения вы познакомитесь с наиболее распространенными тег-хелперами, с которыми встретитесь при работе с формами. Вы также увидите, как с их помощью упростить другие распространенные задачи, среди которых – создание ссылок, условное отображение данных в приложении, и как сделать так, чтобы пользователи видели последнюю версию файла изображения, когда обновляют страницу в браузере.

Для начала я немножко расскажу, для чего нужны тег-хелперы, если Razor уже может генерировать любой HTML-код, который вы пожелаете, путем объединения C# и HTML в один файл.

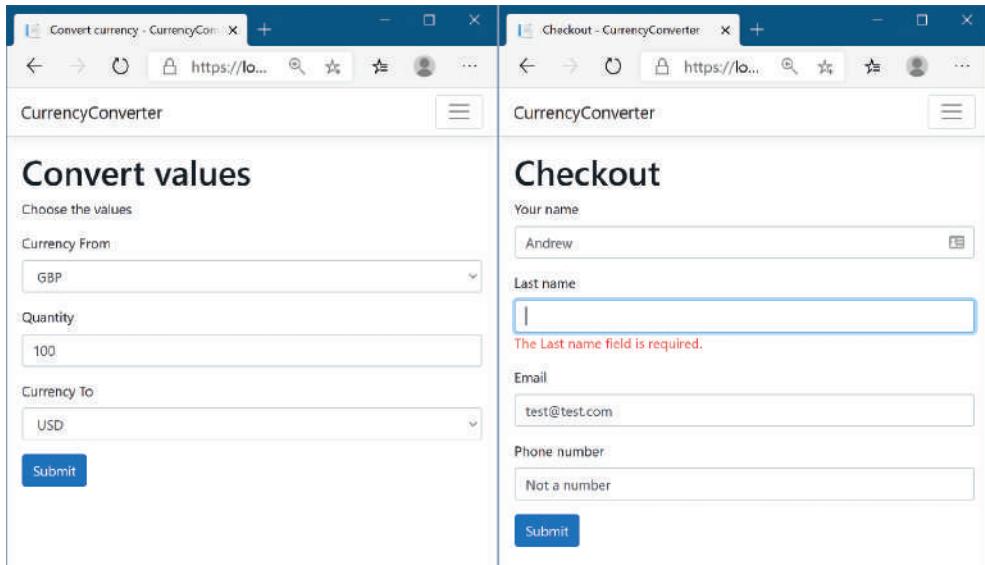


Рис. 18.1 Формы приложения конвертера валют, созданные с помощью тег-хелперов. Ярлыки, раскрывающиеся списки, элементы ввода и сообщения о валидации генерируются с помощью тег-хелперов

18.1 Редакторы кода и тег-хелперы

Одна из распространенных жалоб на сочетание C# и HTML в шаблонах Razor заключается в том, что с ними нелегко использовать стандартные инструменты редактирования HTML-кода; все символы @ и {} в коде C#, как правило, сбивают редакторов кода с толку. Чтение шаблонов также может представлять трудность и для людей; переключение парадигм между C# и HTML иногда может немного раздражать.

Возможно, это не было такой проблемой, когда Visual Studio был единственным поддерживаемым способом создания веб-сайтов на ASP.NET, поскольку, очевидно, он мог без проблем понимать шаблоны и услужливо использовать выделение цветом в редакторе. Но так как ASP.NET Core становится кросс-платформенным, вернулось желание экспериментировать с другими редакторами.

Это было одним из главных мотивов создания тег-хелперов. Они легко интегрируются в стандартный синтаксис HTML, добавляя элементы, которые выглядят как атрибуты, обычно начинающиеся с `asp-*`. Чаще всего они используются для создания HTML-форм, как представлено в следующем листинге. В нем показано представление из первой итерации приложения конвертера валют, в котором вы выбираете валюту и сумму для конвертации.

Листинг 18.1 Форма регистрации пользователя с использованием тег-хелперов

```

@page
@model ConvertModel
<form method="post">
    <div class="form-group">
        <label asp-for="CurrencyFrom"></label>
        <input class="form-control" asp-for="CurrencyFrom" />
        <span asp-validation-for="CurrencyFrom"></span>
    </div>
    <div class="form-group">
        <label asp-for="Quantity"></label>
        <input class="form-control" asp-for="Quantity" />
        <span asp-validation-for="Quantity"></span>
    </div>
    <div class="form-group">
        <label asp-for="CurrencyTo"></label>
        <input class="form-control" asp-for="CurrencyTo" />
        <span asp-validation-for="CurrencyTo"></span>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

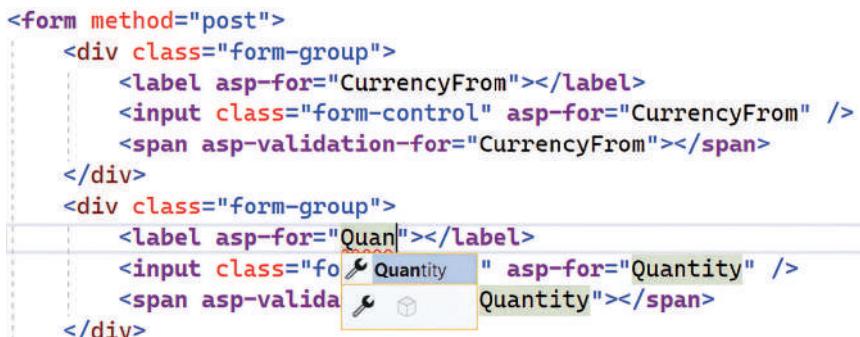
```

Сообщения проверки записываются в диапазон с помощью вспомогательных функций тегов

Это представление для Razor Page Convert.cshtml.
Тип модели – ConvertModel

asp-for на полях ввода генерирует правильный тип, значение, имя и атрибуты проверки для модели

На первый взгляд, вы можете даже не заметить тег-хелперы, настолько хорошо они сочетаются с HTML-кодом! Это упрощает редактирование файлов с помощью любого стандартного текстового редактора HTML. Но не беспокойтесь о том, что вы пожертвовали удобочитаемостью, предоставляемой Visual Studio, – как видно на рис. 18.2, элементы с тег-хелперами четко отличаются от стандартного элемента `<div>` и стандартного атрибута `class` элемента `<input>`. Свойства C# модели представления, на которую ссылаются (в данном случае `Curren-`
`cyFrom`), также отображаются иначе, чем «обычные» атрибуты HTML. И конечно же, как и следовало ожидать, вы получаете IntelliSense. Большинство других интегрированных сред разработки (IDE) также включают подсветку синтаксиса и поддержку IntelliSense.



```

<form method="post">
    <div class="form-group">
        <label asp-for="CurrencyFrom"></label>
        <input class="form-control" asp-for="CurrencyFrom" />
        <span asp-validation-for="CurrencyFrom"></span>
    </div>
    <div class="form-group">
        <label asp-for="Quantity"></label>
        <input class="form-control" asp-for="Quantity" />
        <span asp-validation-for="Quantity"></span>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

```

Рис. 18.2 В Visual Studio тег-хелперы отличаются от обычных элементов и атрибутов жирным шрифтом и иным цветом

Тег-хелперы – это дополнительные атрибуты стандартных элементов HTML (или полностью новые элементы), которые работают путем изменения HTML-элемента, к которому они прикреплены. Они позволяют легко интегрировать ваши значения на стороне сервера, как те, что представлены в `PageModel`, со сгенерированным HTML-кодом.

Обратите внимание, что в листинге 18.1 не указаны заголовки, которые будут отображаться в ярлыках. Вместо этого мы декларативно использовали `asp-for="CurrencyFrom"`, чтобы сказать: «для этого элемента `<label>` используйте свойство `CurrencyFrom`, чтобы определить, какой заголовок применять». Аналогичным образом для элементов `<input>` тег-хелперы используются, чтобы:

- автоматически заполнять значение из свойства `PageModel`;
- выбрать правильный идентификатор и имя, чтобы при отправке формы обратно на страницу Razor свойство было правильно привязано к модели;
- выбрать правильный тип ввода для отображения (например, числовой ввод для свойства `Quantity`);
- отобразить все ошибки валидации, как показано на рис. 18.3.

Рис. 18.3 Тег-хелперы подключаются к метаданным, предоставляемым атрибутами `DataAnnotations`, а также к самим типам свойств. Тег-хелпер валидации может даже заполнять сообщения об ошибках на основе `ModelState`, как было показано в предыдущей главе

Тег-хелперы могут выполнять множество функций, изменяя элементы HTML, к которым они применяются. В этой главе рассказывается о нескольких распространенных тег-хелперах и о том, как их использовать, но это не исчерпывающий список. Я не буду рассказывать обо всех тег-хелперах, которые входят в стандартную комплектацию ASP.NET Core (с каждым выпуском их будет все больше!), а вы легко можете создать собственные тег-хелперы, как будет показано в главе 32. В качестве альтернативы можно использовать те, что опубликованы другими на NuGet или GitHub.

Вспоминая WebForms

У тех, кто использовал ASP.NET еще во времена WebForms, до появления паттерна MVC для веб-разработки, тег-хелперы могут вызывать плохие воспоминания. Хотя префикс `asp-` чем-то напоминает определения элементов управления веб-сервера ASP.NET, не бойтесь – это два разных существа.

Элементы управления веб-сервера были напрямую добавлены в сопутствующий класс страницы C# и имели обширную область действия, которая могла изменять, казалось бы, несвязанные части страницы. В сочетании с этим у них был сложный жизненный цикл, который было трудно понять и отладить, когда что-то не работало. Опасность попытки работать с таким уровнем сложности не забыта, но тег-хелперы не то же самое.

У тег-хелперов нет жизненного цикла – они участвуют в отрисовке элемента, к которому прикреплены, и все. Они могут изменять HTML-элемент, к которому прикреплены, но не могут изменять что-либо еще на странице, что значительно упрощает их концептуально. Дополнительная возможность, которую они привносят, – это возможность иметь несколько тег-хелперов, действующих на один элемент, – что несложно было сделать с помощью элементов управления веб-сервером.

В целом, когда вы будете писать шаблоны Razor, то получите гораздо больше удовольствия, если будете использовать тег-хелперы как неотъемлемую часть синтаксиса. Они приносят много пользы без очевидных недостатков, и ваши друзья, кросс-платформенные редакторы, будут вам благодарны!

18.2 Создание форм с помощью тег-хелперов

В этом разделе вы узнаете, как использовать одни из наиболее полезных тег-хелперов: тег-хелперы, которые работают с формами. Вы увидите, как использовать их для генерации разметки HTML на основе свойств `PageModel`, создавая правильные атрибуты `id` и `name` и задавая `value` элемента значению свойства модели (среди прочего). Такая возможность значительно сокращает объем разметки, которую необходимо писать вручную.

Представьте, что вы создаете страницу оформления заказа для приложения конвертера валют и вам нужно получить данные пользователя на странице оформления заказа. В главе 16 мы создали модель `UserBindingModel` (показанную в листинге 18.2), добавили атрибуты `DataAnnotations` для валидации и увидели, как привязать ее к модели в POST-запросе к странице Razor. В этой главе вы увидите, как создать для нее представление, предоставив `UserBindingModel` в качестве свойства `PageModel`.

ПРЕДУПРЕЖДЕНИЕ Используя Razor Pages, вы часто предоставляете в представлении тот же объект, который применяете для привязки модели. Делая это, нужно быть осторожными, чтобы не включить в модель привязки конфиденциальные значения (которые не следует редактировать), дабы избежать атак с помощью оверпостинга. Подробнее об этих атаках можно прочитать в моем блоге на странице <http://mng.bz/RXw0>.

Листинг 18.2 UserBindingModel для создания пользователя на странице оформления заказа

```
public class UserBindingModel
{
    [Required]
    [StringLength(100, ErrorMessage = "Maximum length is {1}")]
    [Display(Name = "Your name")]
    public string FirstName { get; set; }

    [Required]
    [StringLength(100, ErrorMessage = "Maximum length is {1}")]
    [Display(Name = "Last name")]
    public string LastName { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Phone(ErrorMessage = "Not a valid phone number.")]
    [Display(Name = "Phone number")]
    public string PhoneNumber { get; set; }
}
```

UserBindingModel декорирована рядом атрибутов DataAnnotations. В главе 16 мы видели, что эти атрибуты используются во время валидации модели, когда модель привязывается к запросу, до того, как будет выполнен обработчик страницы. Они также используются языком шаблонизатора Razor для предоставления метаданных, необходимых для генерации правильного HTML-кода при использовании тег-хеллеров.

Можно использовать паттерн, описанный в главе 16, предоставив UserBindindModel в качестве свойства Input нашей PageModel, чтобы использовать модель и для привязки модели, и в представлении Razor:

```
public class CheckoutModel: PageModel
{
    [BindProperty]
    public UserBindingModel Input { get; set; }
}
```

С помощью свойства UserBindingModel, тег-хеллеров и небольшого количества HTML-кода можно создать представление Razor, позволяющее пользователю вводить сведения о себе, как показано на рис. 18.4.

Шаблон Razor для создания этой страницы показан в листинге 18.3. В этом коде используются различные тег-хелперы, в том числе:

- тег-хелпер Form для элемента <form>;
- тег-хелперы Label для элементов <label>;
- тег-хелперы Input для элементов <input>;
- тег-хелперы Validation Message для элементов валидации у каждого свойства в UserBindingModel.

The screenshot shows a web browser window with the title "Checkout - CurrencyConverter". The URL in the address bar is https://lo... . The main content is a "Checkout" form. It includes fields for "Your name" (containing "Andrew"), "Last name" (empty), "Email" (containing "test@example.com"), and "Phone number" (empty). Below the "Last name" field, there is a red validation message: "The Last name field is required.". At the bottom is a blue "Submit" button.

Рис. 18.4 Страница для ввода сведений о пользователе. HTML-код генерируется на основе UserBindingModel, используя тег-хелперы для визуализации необходимых значений элемента, типов вводимых данных и сообщений валидации

Листинг 18.3 Шаблон Razor для привязки к UserBindingModel

```
@page
@model CheckoutModel
```

CheckoutModel – это PageModel, который предоставляет UserBindingModel для свойства Input

```
@{
    ViewData["Title"] = "Checkout";
}
<h1>@ViewData["Title"]</h1>
<form asp-page="Checkout">
    <div class="form-group">
        <label asp-for="Input.FirstName"></label>
        <input class="form-control" asp-for="Input.FirstName" />
        <span asp-validation-for="Input.FirstName"></span>
    </div>
    <div class="form-group">
        <label asp-for="Input.LastName"></label>
```

Тег-хелперы Label используют DataAnnotations для свойства, чтобы определить отображаемую подпись

```

<input class="form-control" asp-for="Input.LastName" />
<span asp-validation-for="Input.LastName"></span>
</div>                                     Тег-хелперы Input используют
                                                DataAnnotations, чтобы определить
                                                тип генерируемых входных данных
<div class="form-group">
    <label asp-for="Input.Email"></label>
    <input class="form-control" asp-for="Input.Email" />
    <span asp-validation-for="Input.Email"></span>
</div>                                     Тег-хелперы валидации отобража-
                                                ют сообщения об ошибках, связанных
                                                с данным свойством
<div class="form-group">
    <label asp-for="Input.PhoneNumber"></label>
    <input class="form-control" asp-for="Input.PhoneNumber" />
    <span asp-validation-for="Input.PhoneNumber"></span>
</div>
<button type="submit" class="btn btn-primary">Submit</button>
</form>

```

Вы видите HTML-разметку, создаваемую этим шаблоном, в листинге 18.4, которая отображается в браузере, как показано на рис. 18.4. Видно, что каждый из HTML-элементов с тег-хеллером был расширен при выводе: у элемента `<form>` есть атрибут `action`, у элементов `<input>` – идентификатор и имя на основе имени ссылаемого свойства, а у элементов `<input>` и `` – атрибуты `data-*` для валидации.

Листинг 18.4 HTML-код, сгенерированный шаблоном Razor

```

<form action="/Checkout" method="post">
    <div class="form-group">
        <label for="Input_FirstName">Your name</label>
        <input class="form-control" type="text"
            data-val="true" data-val-length="Maximum length is 100"
            id="Input_FirstName" data-val-length-max="100"
            data-val-required="The Your name field is required."
            Maxlength="100" name="Input.FirstName" value="" />
        <span data-valmsg-for="Input.FirstName"
            class="field-validation-valid" data-valmsg-replace="true"></span>
    </div>
    <div class="form-group">
        <label for="Input_LastName">Your name</label>
        <input class="form-control" type="text"
            data-val="true" data-val-length="Maximum length is 100"
            id="Input_LastName" data-val-length-max="100"
            data-val-required="The Your name field is required."
            Maxlength="100" name="Input.LastName" value="" />
        <span data-valmsg-for="Input.LastName"
            class="field-validation-valid" data-valmsg-replace="true"></span>
    </div>
    <div class="form-group">
        <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." />
    </div>

```

```
    Data-val-required="The Email field is required."
    Id="Input_Email" name="Input.Email" value="" />
    <span class="text-danger field-validation-valid"
        data-valmsg-for="Input.Email" data-valmsg-replace="true"></span>
    </div>
<div class="form-group">
    <label for="Input_PhoneNumber">Phone number</label>
    <input class="form-control" type="tel" data-val="true"
        data-val-phone="Not a valid phone number." Id="Input_PhoneNumber"
        name="Input.PhoneNumber" value="" />
    <span data-valmsg-for="Input.PhoneNumber"
        class="text-danger field-validation-valid"
        data-valmsg-replace="true"></span>
</div>
<button type="submit" class="btn btn-primary">Submit</button>
<input name="__RequestVerificationToken" type="hidden"
    value="CfDJ8PkYhAINFx1JmYUVIDWbpPyy_TRUNCATED" />
</form>
```

Ух, как много разметки! Если вы новичок в HTML, то все это может показаться немножко сложным, но важно отметить, что вам не нужно было писать большую часть этого кода! Тег-хелперы сделали большую часть работы за вас. Они упрощают сложный механизм создания HTML-форм, позволяя сосредоточиться на общем дизайне приложения, вместо того чтобы писать шаблонный код разметки.

ПРИМЕЧАНИЕ Если вы используете Razor для создания представлений, тег-хелперы облегчат вам жизнь, но они совершенно необязательны. Можно писать низкоуровневый HTML-код без них или использовать устаревшие HTML-хелперы.

Тег-хелперы упрощают и абстрагируют процесс генерации HTML-кода и обычно стараются делать это, не мешая вам. Если вам нужен окончательный генерированный HTML-код с определенным атрибутом, можете добавить его в свою разметку. Это видно в предыдущих листингах, где атрибуты `class` определены в элементах `<input>`, например `<input class="form-control" asp-for="Input.FirstName" />`. Они переходят нетронутыми из Razor в вывод HTML.

СОВЕТ Это отличается от того, как работали HTML-хелперы в предыдущей версии ASP.NET; часто требовалось чуть ли не с бубном плясать, чтобы задать атрибуты в генерированной разметке.

Мало того, вы также можете задать атрибуты, которые обычно генерируются тег-хелпером, как, например, атрибут `type` элемента `<input>`.

Например, если свойство `FavoriteColor` вашей `PageModel` было бы строкой, тогда по умолчанию тег-хелперы генерировали бы элемент `<input>` с помощью этого: `type="text"`. А если вы хотите обновить свою разметку для использования типа выбора цвета HTML5, то сделать это банально просто – явно задайте атрибут `type` в представлении Razor:

```
<input type="color" asp-for="FavoriteColor" />
```

СОВЕТ HTML5 добавляет огромное количество функций, включая множество элементов формы, которые вы, возможно, не встречали раньше, например элементы <input> с типами range и color.

Я не буду описывать их в этой книге, но вы можете прочитать о них на сайте Mozilla Developer Network: <http://mng.bz/qOc1>.

В этом разделе вы создадите шаблоны Razor для калькулятора валют с нуля, добавляя тег-хелперы, когда вам понадобится. Вы, вероятно, обнаружите, что используете большую часть тег-хелперов для создания привычных форм в каждом приложении, даже если это просто страница авторизации.

18.2.1 Тег-хелпер формы

Неудивительно, что первое, что вам нужно для создания HTML-формы, – это элемент <form>. В предыдущем примере он был дополнен атрибутом `asp-page`:

```
<form asp-page="Checkout">
```

В результате в окончательный HTML-код добавляются `action` и `method`, указывающие, на какой URL-адрес должно быть отправлено содержимое формы и какой HTTP-метод использовать:

```
<form action="/Checkout" method="post">
```

Установка атрибута `asp-page` позволяет указать в приложении другую страницу Razor, на которую будет отправляться содержимое формы. Если убрать его, содержимое уйдет обратно на тот же URL-адрес, с которого было отправлено, что очень распространено в Razor Pages. Обычно результат отправки содержимого формы обрабатывается на той же странице Razor, которая используется для ее отображения.

ВНИМАНИЕ! Если вы уберете атрибут `asp-page`, то должны вручную добавить атрибут `method="post"`. Это важно сделать, чтобы содержимое формы отправлялось с использованием метода POST, а не GET по умолчанию. Использование метода GET может представлять угрозу для безопасности.

Атрибут `asp-page` добавляется с помощью `FormTagHelper`. Этот тег-хелпер использует значение, предоставленное для создания URL-адреса для атрибута `action` с применением функций маршрутизации, которые описаны в главах 5 и 14.

ПРИМЕЧАНИЕ Тег-хелперы могут сделать доступными для элемента несколько атрибутов. Рассматривайте их как свойства объекта конфигурации тег-хелпера. Добавляя единственный атрибут `asp-`, вы активируете тег-хелпер для элемента. Добавление дополнительных атрибутов позволяет переопределять другие значения по умолчанию для его реализации.

Тег-хелпер формы делает несколько других атрибутов доступными в элементе `<form>`, которые можно использовать для настройки сгенерированного URL-адреса. Надеюсь, вы помните, что можно задать значения маршрута при генерации URL-адресов. Например, если у вас есть страница Razor, `Product.cshtml`, использующая директиву

```
@page "{id}"
```

то полный шаблон маршрута для страницы будет выглядеть так: "Product/{id}". Чтобы правильно сгенерировать URL-адрес этой страницы, необходимо указать значение `{id}`. Как задать это значение с помощью тег-хелпера формы?

Тег-хелпер формы определяет подстановочный атрибут `asp-route-*`, который можно использовать, чтобы задать произвольные параметры маршрута. Задайте * в атрибуте для имени параметра маршрута. Например, чтобы задать параметр маршрута `id`, нужно задать значение `asp-route-id`. Если свойство `ProductId` вашей `PageModel` содержит необходимое значение идентификатора, то можно использовать следующий код:

```
<form asp-page="Product" asp-route-id="@Model.ProductId">
```

На основе шаблона маршрута страницы `Product.cshtml` (и предполагая, что `ProductId = 5` в этом примере) будет создана следующая разметка:

```
<form action="/Product/5" method="post">
```

Основная задача тег-хелпера формы – создать атрибут `action`, но он выполняет одну важную дополнительную функцию: создает скрытое поле `<input>` для предотвращения *межсайтовой подделки запросов*.

ОПРЕДЕЛЕНИЕ Атаки с *межсайтовой подделкой запросов* (CSRF) – это эксплойт, который может разрешить выполнение действий на вашем веб-сайте сторонним вредоносным сайтом. Подробнее о них вы узнаете в главе 29.

В листинге 18.4 вы видите сгенерированный скрытый тег `<input>` в нижней части формы. Он называется `_RequestVerificationToken` и содержит на первый взгляд случайную строку символов. Само по себе это поле не защитит вас, но в главе 29 я расскажу, как использовать его для защиты своего сайта. Тег-хелпер формы генерирует его по умолчанию, поэтому обычно не нужно беспокоиться об этом, но если вам нужно отключить его, это можно сделать, добавив `asp-antiforgery="false"` в элемент `<form>`.

Тег-хелпер формы, очевидно, полезен для генерации URL-адреса атрибута `action`, но пора переходить к более интересным элементам – тем, что вы видите в своем браузере!

18.2.2 Тег-хелпер метки

Каждое поле `<input>` в приложении конвертера валют должно иметь ассоциированную метку, чтобы пользователь знал, для чего нужен тег `<input>`. Можно было бы легко создать их самостоятельно, вручную введя

имя поля и задав соответствующий атрибут `for`, но, к счастью, есть тег-хелпер, который сделает это за вас.

Тег-хелпер метки используется для создания заголовка (видимого текста) и атрибута `for` элемента `<label>` на основе свойств в `PageModel`, путем указания имени свойства в атрибуте `asp-for`:

```
<label asp-for="FirstName"></label>
```

Тег-хелпер метки использует атрибут `[Display]` `DataAnnotations`, который вы видели в главе 16, чтобы определить соответствующее значение для отображения. Если у свойства, для которого вы создаете метку, нет атрибута `[Display]`, тег-хелпер будет использовать имя свойства. Рассмотрим модель, в которой свойство `FirstName` имеет атрибут `[Display]`, а у свойства `Email` его нет:

```
public class UserModel
{
    [Display(Name = "Your name")]
    public string FirstName { get; set; }
    public string Email { get; set; }
}
```

Используя теги

```
<label asp-for="FirstName"></label>
<label asp-for="Email"></label>
```

мы генерируем следующий HTML-код:

```
<label for="FirstName">Your name</label>
<label for="Email">Email</label>
```

Текст внутри элемента `<label>` использует значение, заданное в атрибуте `[Display]`, или имя свойства в случае свойства `Email`. Также обратите внимание, что атрибут `for` был создан с именем свойства. Это ключевой бонус использования тег-хелпера – он подключается к идентификаторам элементов, созданным другими тег-хелперами. Вскоре вы это увидите.

ПРИМЕЧАНИЕ Атрибут `for` важен для доступности. Он определяет идентификатор элемента, к которому относится метка. Это важно для пользователей, которые, например, используют программу для чтения с экрана, поскольку они могут определить, к какому свойству относится поле формы.

Помимо свойств в `PageModel`, вы также можете ссылаться на вложенные свойства дочерних объектов. Например, как я писал в главе 16, обычно на странице Razor создают вложенный класс, предоставляют его как свойство и декорируют его атрибутом `[BindProperty]`:

```
public class CheckoutModel: PageModel
{
    [BindProperty]
    public UserBindingModel Input { get; set; }
}
```

Можно ссылаться на свойство `FirstName UserBindingModel`, «расставив точки», как это делается в любом другом коде C#. В листинге 18.3 показаны дополнительные примеры.

```
<label asp-for="Input.FirstName"></label>
<label asp-for="Input.Email"></label>
```

Как это обычно бывает с тег-хеллерами, тег-хеллер метки не переопределяет значения, которые вы задали сами. Если, например, вы не хотите использовать заголовок, созданный хеллером, то можете вставить собственный заголовок вручную. Этот элемент `<label>`

```
<label asp-for="Email">Please enter your Email</label>
```

сгенерирует следующий HTML-код:

```
<label for="Email">Please enter your Email</label>
```

Как и всегда, вам будет легче сопровождать код, если вы будете придерживаться стандартных соглашений и не переопределять такие значения, но выбор есть. На очереди у нас важные персоны: тег-хелперы ввода и области текста.

18.2.3 Тег-хелперы текстового ввода

Теперь перейдем к сути нашей формы – элементам `<input>`, обрабатывающим ввод данных, предоставляемых пользователем. Учитывая, что существует очень широкий спектр возможных типов ввода, есть множество различных способов их отображения в браузере. Например, логические значения обычно представлены элементом типа `checkbox`, тогда как целочисленные значения будут использовать тип `number`, а дата будет использовать тип `date`, как показано на рис. 18.5.

Чтобы справиться с таким разнообразием, есть тег-хелпер ввода – один из самых мощных тег-хелперов. Он использует информацию, основанную на типе свойства (`bool`, `string`, `int` и т. д.), и любые примененные к нему атрибуты `DataAnnotations` (`[EmailAddress]` и `[Phone]` среди прочего), чтобы определить тип создаваемого элемента `<input>`. `DataAnnotations` также используются для добавления атрибутов валидации на стороне клиента `data-val-*` в сгенерированном HTML-коде.

Рассмотрим свойство `Email` из листинга 18.2, декорированное атрибутом `[EmailAddress]`. Добавить элемент `<input>` так же просто, как использовать атрибут `asp-for`:

```
<input asp-for="Input.Email" />
```

Свойство представляет собой строку, поэтому обычно тег-хелпер ввода генерирует тег `<input>` с типом "text". Но добавление атрибута `[EmailAddress]` предоставляет дополнительные метаданные об объекте.

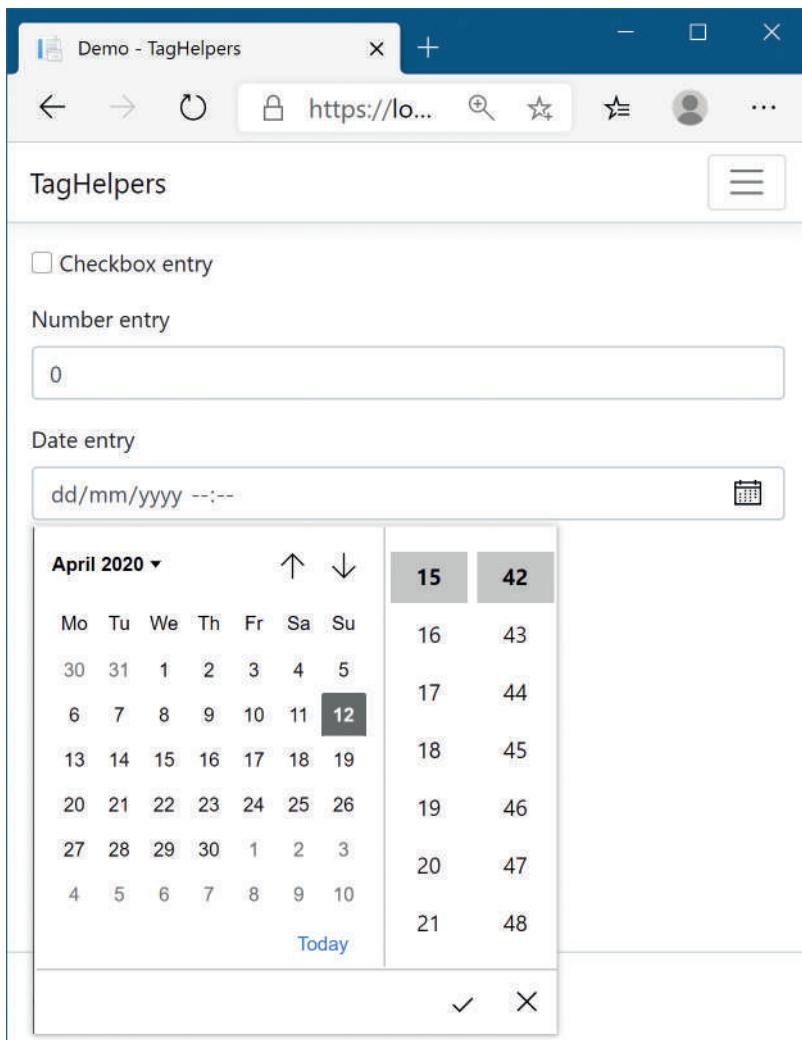


Рис. 18.5 Различные типы элемента `<input>`. Конкретный способ отображения зависит от браузера

Следовательно, тег-хелпер генерирует тег `<input>` с типом "email":

```
<input type="email" id="Input_Email" name="Input.Email"
       value="test@example.com" data-val="true"
       data-val-email="The Email Address field is not a valid e-mail address."
       data-val-required="The Email Address field is required."
       />
```

Из этого примера можно извлечь множество вещей. Во-первых, атрибуты HTML-элемента `id` и `name` были сгенерированы из имени свойства.

Значение атрибута `id` совпадает со значением, сгенерированным тег-хеллером метки в атрибуте `for`, `Input_Email`. Значение атрибута `name` сохра-

няет «точечную» нотацию `Input_Email`, чтобы привязка модели работала правильно, когда содержимое поля отправляется на страницу Razor.

Кроме того, начальное значение поля было задано равным значению, которое в настоящее время хранится в свойстве (в данном случае `"test@example.com"`). Тип элемента также был установлен как тип HTML5 `email` вместо типа `text` по умолчанию.

Возможно, наиболее ярким дополнением является набор атрибутов `data-val-*`. Они могут использоваться клиентскими библиотеками JavaScript, такими как jQuery, для обеспечения валидации ограничений `DataAnnotations` на стороне клиента. Проверка на стороне клиента обеспечивает пользователям мгновенную обратную связь, если значения, которые они вводят, невалидны, что обеспечивает более удобный пользовательский интерфейс, по сравнению с тем, который мог бы у вас быть, если бы вы использовали только проверку на стороне сервера, о чём я писал в главе 16.

Валидация на стороне клиента

Чтобы активировать валидацию на стороне клиента в своем приложении, необходимо добавить на страницу некоторые библиотеки jQuery. В частности, нужно включить библиотеки jQuery, jQuery-validation и jQuery-validation-unobtrusive.

Сделать это можно несколькими способами, но самый простой – подключить файлы сценариев в нижней части представления:

```
<script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
<script src="~/lib/jquery-validation-unobtrusive/
  jquery.validate.unobtrusive.min.js"></script>
```

В шаблонах по умолчанию эти сценарии уже есть. Это удобный частичный шаблон, который вы можете добавить на свою страницу в секции `Scripts`.

Если вы используете макет по умолчанию и вам необходимо добавить валидацию на стороне клиента в ваше представление, добавьте где-нибудь в представлении следующую секцию:

```
@section Scripts{
    @Html.Partial("_ValidationScriptsPartial")
}
```

Это частичное представление ссылается на файлы в папке `wwwroot`. Шаблон макета по умолчанию включает в себя саму библиотеку jQuery. Если вам не нужно использовать jQuery в приложении, то можете рассмотреть возможность использования небольшой альтернативной библиотеки валидации – `aspnet-client-validation`. В этом посте я описываю, почему вы можете рассмотреть эту библиотеку и как ее использовать: <http://mng.bz/V1pX>.

Вы также можете загрузить эти файлы из сети доставки содержимого (CDN). Если вы хотите использовать данный подход, вам следует рассмотреть сценарии, в которых CDN недоступна или скомпрометирована. Я обсуждаю это в своем посте: <http://mng.bz/2e6d>.

Тег-хелпер ввода пытается выбрать наиболее подходящий шаблон для данного свойства на основе атрибутов `DataAnnotations` или типа свойства. Сгенерирует ли он точно тот тип элемента `<input>`, который вам нужен, может в некоторой степени зависеть от вашего приложения.

Как и всегда, вы можете переопределить сгенерированный тип, добавив собственный атрибут `type` к элементу в вашем шаблоне Razor.

В табл. 18.1 показано, как некоторые привычные типы данных со-поставляются с типами `<input>` и как можно указать сами типы данных.

Таблица 18.1 Распространенные типы данных, их указание и сопоставление с типом элемента `input`

Тип данных	Как он указан	Тип элемента <code><input></code>
<code>byte, int, short, long, uint</code>	Тип свойства	<code>number</code>
<code>decimal, double, float</code>	Тип свойства	<code>text</code>
<code>bool</code>	Тип свойства	<code>checkbox</code>
<code>string</code>	Тип свойства, атрибут <code>[DataType(DataType.Text)]</code>	<code>text</code>
<code>HiddenInput</code>	Атрибут <code>[HiddenInput]</code>	<code>hidden</code>
<code>Password</code>	Атрибут <code>[Password]</code>	<code>password</code>
<code>Phone</code>	Атрибут <code>[Phone]</code>	<code>tel</code>
<code>EmailAddress</code>	Атрибут <code>[EmailAddress]</code>	<code>email</code>
<code>Url</code>	Атрибут <code>[Url]</code>	<code>url</code>
<code>Date</code>	Тип свойства <code>DateTime</code> атрибут <code>[DataType(DataType.Date)]</code>	<code>datetime-local</code>

У тег-хелпера ввода есть один дополнительный атрибут, который можно использовать для настройки способа отображения данных: `asp-format`.

Формы HTML полностью основаны на строках, поэтому когда атрибут `value` элемента `<input>` задан, тег-хелпер должен взять значение, хранящееся в свойстве, и преобразовать его в строку. За кулисами он выполняет метод `string.Format()` для значения свойства, передавая строку формата.

Тег-хелпер ввода использует строку форматирования по умолчанию для каждого типа данных, но в случае с атрибутом `asp-format` можно задать конкретную строку форматирования, которую нужно использовать.

Например, вы можете убедиться, что свойство `Dec` отформатировано до трех десятичных знаков, с помощью следующего кода:

```
<input asp-for="Dec" asp-format="{0:0.000}" />
```

Если бы свойство `Dec` имело значение `1,2`, был бы сгенерирован HTML-код, аналогичный этому коду:

```
<input type="text" id="Dec" name="Dec" value="1.200">
```

В качестве альтернативы можно определить используемый формат, добавив атрибут `[DisplayFormat]` к свойству модели:

```
[DisplayFormat("{0:0.000}")]
public decimal Dec { get; set; }
```

ПРИМЕЧАНИЕ Возможно, вы удивитесь, что типы `decimal` и `double` отображаются как текстовые поля, а не как числовые. Это связано с рядом технических причин, в основном с тем, как некоторые культуры визуализируют числа с запятыми и пробелами. Визуализация в виде текста позволяет избежать ошибок, которые могут появиться только в определенных сочетаниях браузера и культуры.

Помимо тег-хелпера `input`, ASP.NET Core предоставляет тег-хелпер `textarea`. Он работает аналогичным образом, используя атрибут `asp-for`, но привязан к элементу `<textarea>`:

```
<textarea asp-for="BigtextValue"></textarea>
```

Так вы сгенерируете HTML-код, подобный тому, что приведен далее. Обратите внимание, что значение свойства визуализируется внутри тега, а элементы валидации `data-val-*` привязаны как обычно:

```
<textarea data-val="true" id="BigtextValue" name="BigtextValue"
    data-val-length="Maximum length 200." data-val-length-max="200"
    data-val-required="The Multiline field is required.">This is some text,
I'm going to display it
in a text area</textarea>
```

Надеюсь, пройдя этот раздел, вы убедились, насколько тег-хелперы могут помочь вам набирать меньше кода, особенно если использовать их вместе с `DataAnnotations` для генерации атрибутов проверки. Но это больше, чем сокращение количества требуемых нажатий клавиш. Тег-хелперы обеспечивают *корректность* созданной разметки и правильное имя, идентификатор и формат для автоматической привязки моделей при их отправке на сервер.

Имея в своем распоряжении элементы `<form>`, `<label>` и `<input>`, можно создать большую часть формы конвертера валют. Прежде чем мы рассмотрим визуализацию сообщений валидации, есть еще один элемент, на который стоит обратить внимание: `<select>`, или раскрывающийся список.

18.2.4 Тег-хелпер раскрывающегося списка

Помимо полей `<input>`, в веб-формах часто встречается элемент `<select>`, или раскрывающийся список. Наше приложение конвертера валют, например, может использовать его, чтобы можно было выбрать валюту для конвертации из списка.

По умолчанию этот тег показывает список элементов и позволяет выбрать один, но можно выбрать и несколько, как показано на рис. 18.6. Помимо обычного раскрывающегося списка, можно отображать поле со списком, с возможностью множественного выбора или отображением элементов списка группами.

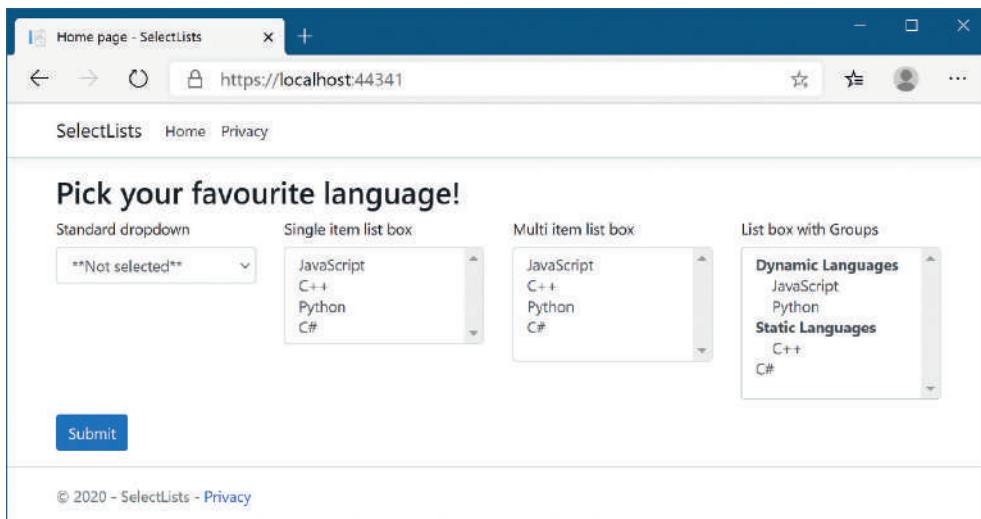


Рис. 18.6 Некоторые из множества способов отображения элементов `<select>` с помощью тег-хелпера раскрывающегося списка

Чтобы использовать элементы `<select>` в коде Razor, необходимо включить в `PageModel` два свойства: одно для списка отображаемых параметров и другое для хранения выбранного значения (или значений). Например, в листинге 18.5 показаны свойства `PageModel`, используемые для создания трех крайних левых списков выбора, показанных на рис. 18.6. Для отображения групп требуется несколько иная настройка, как вы вскоре увидите.

Листинг 18.5 Модель представления для отображения раскрывающегося списка и списка множественного выбора

```
public class SelectListModel: PageModel
{
    [BindProperty]
    public class InputModel { get; set; }

    public IEnumerable<SelectListItem> Items { get; set; }
        = new List<SelectListItem>
    {
        new SelectListItem{Value = "csharp", Text="C#"},
        new SelectListItem{Value = "python", Text= "Python"},
        new SelectListItem{Value = "cpp", Text="C++"},
        new SelectListItem{Value = "java", Text="Java"},
        new SelectListItem{Value = "js", Text="JavaScript"},
        new SelectListItem{Value = "ruby", Text="Ruby"},
    };

    public class InputModel
    {
        public string SelectedValue1 { get; set; }
        public string SelectedValue2 { get; set; }
    }
}
```

InputModel для привязки выбора пользователя к полям выбора

Список элементов для отображения в полях выбора

Эти свойства будут содержать значения, выбранные в полях выбора с одним выбором

```
    public IEnumerable<string> MultiValues { get; set; } <--  
}  
}  
} Чтобы создать список с множественным  
выбором, используйте IEnumerable<>
```

Данный листинг демонстрирует ряд аспектов работы со списками `<select>`:

- `SelectedValue1/SelectedValue2` используется для хранения значения, выбранного пользователем. Они привязаны к значению, выбранному из раскрывающегося списка или списка множественного выбора, и используются для предварительного выбора правильного элемента при визуализации формы;
- `MultiValues` используется для хранения выбранных значений для списка множественного выбора. Это объект `IEnumerable`, поэтому он может содержать несколько вариантов выбора для каждого элемента `<select>`;
- `Items` предоставляет список значений для отображения в элементах `<select>`. Обратите внимание, что тип элемента должен быть `SelectListItem`, который предоставляет свойства `Value` и `Text` для работы с тег-хеллером. Он не является частью `InputModel`, поскольку нам не нужно привязывать эти элементы к запросу – обычно они загружаются непосредственно из модели приложения или вшиты в код. Порядок значений свойства `Items` определяет порядок элементов в списке `<select>`.

ПРИМЕЧАНИЕ Тег-хелпер раскрывающегося списка работает только с элементами `SelectListItem`. Это означает, что обычно нужно выполнять преобразование из набора элементов списка для конкретного приложения (например, `List<string>` или `List<MyClass>`) в ориентированный на пользовательский интерфейс `List<SelectListItem>`.

Тег-хелпер раскрывающегося списка предоставляет атрибуты `asp-for` и `asp-items`, которые можно добавить в элементы `<select>`. Что касается тег-хелпера ввода, атрибут `asp-for` указывает свойство в `PageModel`, к которому нужно выполнить привязку.

Чтобы отобразить доступные элементы `<option>`, предоставляется атрибут `asp-items` для `IEnumerable<SelectListItem>`.

СОВЕТ Обычно требуется отобразить список параметров `enum` в списке `<select>`. Это настолько распространено, что ASP.NET Core поставляется с помощником генерации `SelectListItem` для любого перечисления. Если у вас есть перечисление вида `TEnum`, то можно сгенерировать доступные параметры в представлении: `asp-items="Html.GetEnumSelectList<TEnum>()"`.

В следующем листинге показано, как отобразить раскрывающийся список, список с одним вариантом выбора и список с несколькими вариантами. В нем используется `PageModel` из предыдущего листинга.

Каждый список `<select>` привязан к другому свойству, но для всех них повторно используется один и тот же список `Items`.

Листинг 18.6 Шаблон Razor для отображения элемента `<select>` тремя разными способами

```
@page
@model SelectListsModel
<select asp-for="Input.SelectedValue1"
       asp-items="Model.Items"></select>
<select asp-for="Input.SelectedValue2"
       asp-items="Model.Items" size="4"></select>
<select asp-for="Input.MultiValues"
       asp-items="Model.Items"></select>
```

Создает стандартный раскрывающийся список путем привязки к стандартному свойству в `asp-for`

Создает список с единственным вариантом выбора высотой 4, предоставляя стандартный HTML-атрибут `size`

Создает список с множественным выбором путем привязки к свойству `IEnumerable` в `asp-for`

Надеюсь, вы видите, что шаблон генерации раскрывающегося списка `<select>` почти идентичен шаблону для создания списка множественного выбора. Тег-хелпер заботится о добавлении HTML-атрибута `multiple` к сгенерированному выводу, если свойство, к которому оно привязывается, – это `IEnumerable`.

ВНИМАНИЕ! Атрибут `asp-for` не должен включать в себя префикс `Model`. С другой стороны, он должен быть у атрибута `asp-items` при ссылке на свойство в `PageModel`. Атрибут `asp-items` может также ссылаться на другие элементы C#, такие как объекты, хранящиеся в `ViewData`, но использование свойства `PageModel` – лучший подход.

Вы видели, как связать три разных типа списка выбора, но есть еще один, который мы не рассматривали. Он изображен на рис. 18.6. Речь идет о том, как отображать группы в списках множественного выбора с помощью элементов `<optgroup>`.

К счастью, в коде Razor ничего менять не нужно; просто нужно обновить способ определения объектов `SelectListItem`.

Объект `SelectListItem` определяет свойство `Group`, указывающее группу `SelectListGroup`, к которой принадлежит элемент. В следующем листинге показано, как создать две группы и назначить каждый элемент списка либо «динамической», либо «статической» группе, используя `PageModel`, аналогичную той, что показана в листинге 18.5. Последний элемент списка, C#, не назначается группе, поэтому он будет отображаться как обычно, без тега `<optgroup>`.

Листинг 18.7 Добавление групп в объекты `SelectListItem` для создания элементов `optgroup`

```
public class SelectListsModel: PageModel
{
    [BindProperty]
    public IEnumerable<string> SelectedValues { get; set; }
    public IEnumerable<SelectListItem> Items { get; set; }
```

Содержит выбранные значения, где разрешен множественный выбор

```

public SelectListsModel() ← Инициализирует элементы
{
    var dynamic = new SelectListGroup { Name = "Dynamic" };
    var @static = new SelectListGroup { Name = "Static" };
    Items = new List<SelectListItem> ← Создает один экземпляр
    {
        new SelectListItem { ← каждого группы для передачи
            Value= "js",
            Text="Javascript",
            Group = dynamic
        },
        new SelectListItem { ← в SelectListItems
            Value= "cpp",
            Text="C++",
            Group = @static
        },
        new SelectListItem { ←
            Value= "python",
            Text="Python",
            Group = dynamic
        },
        new SelectListItem { ←
            Value= "csharp",
            Text="C#",
            Group = dynamic
        }
    };
}
}

```

Устанавливает соответствующую группу для каждого SelectListItem

Если у SelectListItem нет группы, он не будет добавлен в <optgroup>

После этого тег-хелпер будет генерировать элементы <optgroup> по мере необходимости при отрисовке шаблона в HTML-код. Этот шаблон Razor:

```

@page
@model SelectListsModel
<select asp-for="SelectedValues" asp-items="Model.Items"></select>

```

будет отображаться в HTML следующим образом:

```

<select id="SelectedValues" name="SelectedValues" multiple="multiple">
    <optgroup label="Dynamic">
        <option value="js">JavaScript</option>
        <option value="python">Python</option>
    </optgroup>
    <optgroup label="Static">
        <option value="cpp">C++</option>
    </optgroup>
    <option value="csharp">C#</option>
</select>

```

Еще одно распространенное требование при работе с элементами <select> – включить в списке параметр, который указывает, что значение не было выбрано, как показано на рис. 18.7. Без этой дополнительной опции раскрывающийся список всегда будет иметь выбранное значение, и по умолчанию оно будет первым элементом в списке.

Этого можно добиться одним из двух способов: либо добавить опцию «не выбран» к доступным объектам `SelectListItem`, либо вручную добавить опцию к Razor, например:

```
<select asp-for="SelectedValue" asp-items="Model.Items">
    <option Value = "">**Не выбран**</option>
</select>
```

Так мы добавляем дополнительный элемент `<option>` с пустым атрибутом `Value` в начало элемента `<select>`, позволяя предоставить пользователю вариант «не выбран».

СОВЕТ Добавление опции «не выбран» к элементу `<select>` настолько распространено, что вы, возможно, захотите создать частичное представление для инкапсуляции этой логики.

Имея в своем арсенале тег-хелперы ввода и раскрывающегося списка, вы сможете создавать большинство форм, которые вам понадобятся. Теперь у вас есть все необходимое для создания приложения конвертера валют, за одним исключением.

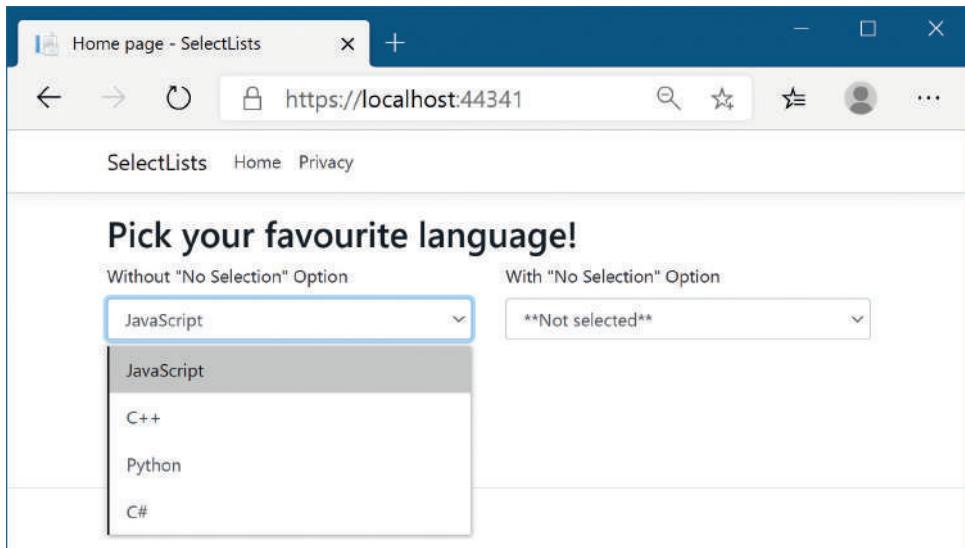


Рис. 18.7 Без опции «не выбран» элемент `<select>` всегда будет иметь какое-то значение. Возможно, это не то поведение, которое вам нужно, в случае если вы хотите, чтобы ни один элемент `<option>` не был выбран по умолчанию

Помните, что всякий раз, когда вы принимаете данные, вводимые пользователем, всегда нужно проверять их. Тег-хелперы валидации позволяют отображать пользователю ошибки проверки модели в форме без необходимости писать большое количество шаблонной разметки.

18.2.5 Тег-хелперы сообщений валидации и сводок валидации

В разделе 18.2.3 было показано, как тег-хелпер `input` генерирует необходимые атрибуты валидации `data-val-*` на самих элементах `<input>`.

Но нам нужно где-то отображать сообщения проверки. Это можно сделать для каждого свойства в модели представления с помощью тег-хелпера сообщений валидации, примененного к тегу ``, используя атрибут `asp-validation-for`:

```
<span asp-validation-for="Email"></span>
```

Когда ошибка возникает во время валидации на стороне клиента, соответствующее сообщение об ошибке для указанного свойства будет отображаться в тегах ``, как показано на рис. 18.8. Этот элемент `` также будет использоваться для отображения соответствующих сообщений валидации, если валидация на стороне сервера окончится неудачей, и форма будет отображена повторно.

Рис. 18.8 Сообщения валидации могут быть показаны в ассоциированном элементе `` с помощью соответствующего тег-хелпера



Любые ошибки, ассоциированные со свойством `Email`, хранящимся в `ModelState`, будут отображены в теле элемента, и у элемента будут соответствующие атрибуты для подключения проверки с помощью jQuery:

```
<span class="field-validation-valid" data-valmsg-for="Email"
      data-valmsg-replace="true">The Email Address field is required.</span>
```

Ошибка валидации, отображаемая в элементе, будет заменена, когда пользователь обновит поле `Email` `<input>`, и валидация будет выполнена на стороне клиента.

ПРИМЕЧАНИЕ Для получения дополнительных сведений о `ModelState` и валидации модели на стороне сервера см. главу 16.

Помимо отображения сообщений валидации для отдельных свойств, также можно отобразить сводку всех сообщений в элементе `<div>` с помощью тег-хелпера сводки валидации, показанного на рис. 18.9. Он отображает теги ``, содержащие список ошибок `ModelState`.

Тег-хелпер сводки валидации применяется к элементу `<div>` с помощью атрибута `asp-validation-summary` и предоставления значения перечисления `ValidationSummary`, например

```
<div asp-validation-summary="All"></div>
```

Перечисление `ValidationSummary` управляет отображаемыми значениями и имеет три возможных значения:

- `None` – не отображать сводку (не знаю, зачем это использовать);
- `ModelOnly` – отображать только ошибки, не ассоциированные со свойством;
- `All` – отображать ошибки, ассоциированные со свойством либо с моделью.

Тег-хелпер сообщения валидации

Тег-хелпер сводки валидации

Рис. 18.9 Форма, показывающая ошибки валидации. Тег-хелпер сообщения валидации применяется к элементу `` рядом с ассоциированным элементом `<input>`, а тег-хелпер сводки валидации – к элементу `<div>`, как правило, в верхней или нижней части формы

Тег-хелпер сводки валидации особенно полезен, если у вас есть ошибки, связанные с вашей страницей, которые не относятся только к одному свойству. Их можно добавить к состоянию модели, используя пустой ключ, как показано в листинге 18.8. В этом примере валидация свойства прошла успешно, но мы предоставляем дополнительную проверку на уровне модели, чтобы убедиться, что мы не пытаемся конвертировать валюту в саму себя.

Листинг 18.8 Добавление ошибок валидации на уровне модели в ModelState

```
public class ConvertModel : PageModel
{
    [BindProperty]
    public InputModel Input { get; set; }

    [HttpPost]
    public IActionResult OnPost()
    {
        if(Input.CurrencyFrom == Input.CurrencyTo) ←
        {
            ModelState.AddModelError(
                string.Empty,
                "Cannot convert currency to itself");
        }
        if (!ModelState.IsValid)
        {
            return Page(); ←
        }
    }
}
```

Конвертация валюты из самой в себя невозможна

Добавляем ошибку уровня всей модели (не привязанную к конкретному свойству) при помощи пустого ключа

Отображаем ошибки уровня модели или уровня свойства, при наличии

```
// Сохраняем где-нибудь допустимые значения и т. д.;
return RedirectToAction("Checkout");
}
}
```

Без этого тег-хелпера ошибки на уровне модели все равно будут добавлены, если пользователь дважды использовал одну и ту же валюту, и форма будет отображена повторно. К сожалению, у пользователя не было визуальной подсказки, указывающей на то, почему данные не были отправлены, – очевидно, это проблема! При добавлении тег-хелпера сводки сообщений валидации ошибки на уровне модели отображаются пользователю, чтобы он мог исправить проблему, как показано на рис. 18.10.

ПРИМЕЧАНИЕ Для простоты я добавил валидацию в обработчик страницы. Более подходящий способ может заключаться в создании специального атрибута валидации или использования интерфейса `IValidatableObject` (как описано в главе 7). Таким образом, ваш обработчик сохранит гибкость, соответствующую принципу единственной ответственности. Вы увидите, как создать собственный атрибут валидации, в главе 32.

В этом разделе описано большинство распространенных тег-хелперов, доступных для работы с формами, включая все части, необходимые для создания форм конвертера валют. Они должны дать вам все необходимое, чтобы вы могли приступить к созданию форм в собственных приложениях. Но формы – не единственная область, где полезны тег-хелперы; обычно они применяются всякий раз, когда вам нужно смешать логику на стороне сервера с генерацией HTML-кода.

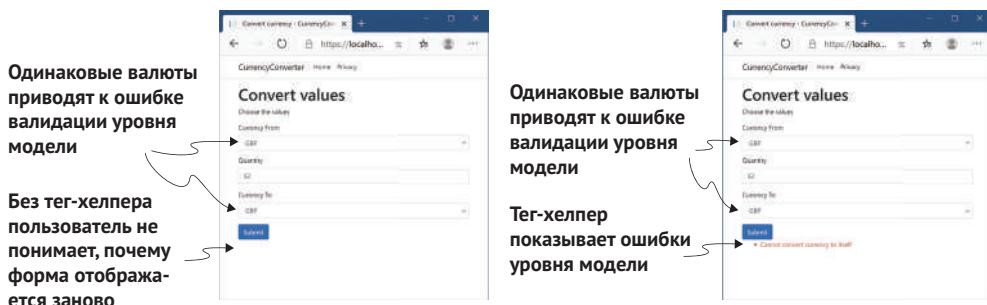


Рис. 18.10 Ошибки на уровне модели отображаются только с помощью тег-хелпера сводки сообщений валидации. Без него у пользователей не будет никаких указаний на то, что в форме есть ошибки, поэтому они не смогут исправить их

Один из таких примеров – создание ссылок на другие страницы приложения с помощью генерации URL-адресов на основе маршрутизации. Если учесть, что маршрутизация спроектирована так, чтобы быть гибкой при рефакторинге приложения, то отслеживание точных URL-адресов, на которые должны указывать ссылки, превратится в кошмар при сопровождении, если вам придется делать это вручную. Как и следовало ожидать, для этого есть тег-хелпер: тег-хелпер привязки ссылки.

18.3 Создание ссылок с помощью тег-хелпера привязки ссылки

В главах 6 и 15 я показал, как генерировать URL-адреса для ссылок на другие страницы приложения с помощью `Link Generator` и `UrlHelper`. Представления – еще одно распространенное место, где нужно ссылаться на другие страницы, обычно с помощью элемента `<a>` с атрибутом `href`, указывающим на соответствующий URL-адрес.

В этом разделе я покажу, как использовать тег-хелпер привязки ссылки, чтобы создать URL-адрес для данной страницы Razor с использованием маршрутизации. Концептуально это почти идентично тому, как тег-хелпер формы генерирует URL-адрес атрибута `action`, как было показано в разделе 18.2.1. По большей части использование тег-хелпера привязки ссылки идентично; вы предоставляете атрибуты `asp-page` и `asp-page-handler` наряду с атрибутами `asp-route-*`, если необходимо. Шаблоны Razor Pages по умолчанию используют тег-хелпер привязки ссылки для создания ссылок в панели навигации с помощью кода из следующего листинга.

Листинг 18.9 Использование тег-хелпера привязки ссылки для генерации URL-адресов в _Layout.cshtml

```
<ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-area="" asp-page="/Index">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark"
            asp-area="" asp-page="/Privacy">Privacy</a>
    </li>
</ul>
```

Как видите, у каждого элемента `<a>` есть атрибут `asp-page`. Этот тег-хелпер использует систему маршрутизации для генерации соответствующего URL-адреса для элемента `<a>`, что приводит к появлению следующей разметки:

```
<ul class="nav navbar-nav">
    <li class="nav-item">
        <a class="nav-link text-dark" href="/">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" href="/Privacy">Privacy</a>
    </li>
</ul>
```

URL-адреса используют значения по умолчанию там, где это возможно, поэтому страница Razor `Index` генерирует просто "/" вместо `"/Index"`.

Если вам нужно больше контроля над созданным URL-адресом, тег-хелпер привязки ссылки предоставляет несколько дополнительных

свойств, которые можно задать и которые будут использоваться при генерации URL-адреса.

Вот атрибуты, наиболее часто используемые в Razor Pages:

- `asp-page` – указывает на страницу Razor, которая будет обрабатывать запрос;
- `asp-page-handler` – указывает на обработчик страницы Razor, которая будет применяться для обработки запроса;
- `asp-area` – устанавливает параметр маршрута области, который будет использоваться. Области могут быть применены для представления дополнительного уровня организации приложения¹;
- `asp-host` – если он задан, то ссылка будет указывать на предоставленный хост и генерировать абсолютный URL-адрес вместо относительного;
- `asp-protocol` – устанавливает, какую следует генерировать: ссылку с `http` или `https`. Если он задан, то будет генерировать абсолютный URL-адрес вместо относительного;
- `asp-route-*` – задает параметры маршрута для использования во время генерации. Его можно добавлять несколько раз для разных параметров маршрута.

Используя тег-хелпер привязки ссылки и его атрибуты, мы генерируем URL-адреса с помощью системы маршрутизации, как описано в главах 5 и 14. Это уменьшает дублирование в коде, удаляя вшитые в код URL-адреса, которые в противном случае нужно было бы встраивать во все представления.

Если вы обнаружите, что пишете повторяющийся код в своей разметке, скорее всего, кто-то уже написал тег-хелпер, который может помочь вам. Тег-хелпер `добавления версии`, о котором идет речь в следующем разделе, – отличный пример использования тег-хелперов для уменьшения количества необходимого кода.

18.4 Сброс кеша с помощью тег-хелпера добавления версии

Распространенная проблема, возникающая и при разработке, и при развертывании приложения в промышленном окружении, – это гарантия того, что все браузеры используют самые последние файлы. Из соображений производительности браузеры часто кешируют файлы локально и повторно используют их для последующих запросов, вместо того чтобы вызывать приложение каждый раз, когда запрашивается файл.

Обычно это замечательно – большинство статических ресурсов на сайте редко меняются, поэтому их кеширование значительно снижает нагрузку на сервер. Возьмем изображение логотипа вашей компании – как часто оно меняется? Если на каждой странице есть ваш логотип, то кеширование изображения в браузере имеет большой смысл.

¹ Я не буду подробно останавливаться на этих вопросах в данной книге. Это неизбежный аспект MVC, который часто используется только в крупных проектах. Вы можете прочитать о них здесь: <http://mng.bz/3X64>.

Но что будет, если его *поменять*? Вы хотите, чтобы пользователи получали обновленные ресурсы, как только они станут доступны. Более важным требованием может быть, если меняются файлы JavaScript, ассоциированные с вашим сайтом. Если пользователи в конечном итоге будут использовать кешированные версии JavaScript, они могут увидеть странные ошибки, или им может показаться, что приложение неисправно.

Такая головоломка часто встречается в веб-разработке, и один из самых распространенных способов справиться с ней – использовать строку запроса со сбросом кеша.

ОПРЕДЕЛЕНИЕ Страна запроса со сбросом кеша добавляет параметр запроса к URL-адресу, например ?v=1. Браузеры будут кешировать ответ и использовать его для последующих запросов к URL-адресу. Когда ресурс изменяется, строка запроса также изменяется, например ?v=2. Браузеры будут рассматривать это как запрос нового ресурса и будут делать новый запрос.

Самая большая проблема с этим подходом заключается в том, что он требует, чтобы вы обновляли URL-адрес каждый раз, когда меняется изображение, файл CSS или JavaScript.

Это шаг, совершающийся вручную. Он требует обновлять все места, где упоминается ресурс, поэтому ошибки неизбежны. Но тег-хелперы спешат на помощь! Когда вы добавляете элемент `<script>`, `` или `<link>` в свое приложение, то можете использовать тег-хелперы для автоматической генерации строки запроса со сбросом кеша:

```
<script src "~/js/site.js" asp-append-version="true"></script>
```

Атрибут `asp-append-version` загрузит файл, на который указывает ссылка, и сгенерирует уникальный хеш на основе его содержимого. Затем он добавляется в виде уникальной строки запроса к URL-адресу ресурса:

```
<script src "/js/site.js?v=EWaMeWsJBYWmL2g_KkgXZQ5nPe"></script>
```

Поскольку это значение является хешем содержимого файла, оно останется неизменным до тех пор, пока файл не изменится, поэтому файл будет кешироваться в браузерах пользователей. Но если файл изменен, изменится хеш содержимого, а следовательно, и строка запроса.

Это гарантирует, что браузеры всегда получают самые свежие файлы для приложения и вам не нужно беспокоиться о ручном обновлении каждого URL-адреса при каждом изменении файла.

До сих пор в этой главе вы видели, как использовать тег-хелперы для форм, создания ссылок и сброса кеша. Вы также можете использовать их для условной визуализации разной разметки в зависимости от текущего окружения. Здесь используется техника, которую вы еще не видели. И тег-хелпер объявляется как совершенно отдельный элемент.

18.5 Использование условной разметки с помощью тег-хелпера окружения

Во многих случаях вам нужно визуализировать разный HTML-код в своих шаблонах Razor в зависимости от того, работает ваш сайт в окружении разработки или в промышленном окружении. Например, в окружении разработки обычно необходимо, чтобы ресурсы JavaScript и CSS были подробными и удобными для чтения, но в промышленном окружении мы обрабатываем эти файлы, чтобы сделать их как можно меньше. Еще один пример – желание применить баннер к приложению при запуске в тестовом окружении, который удаляется при переходе в промышленное окружение, как показано на рис. 18.11.

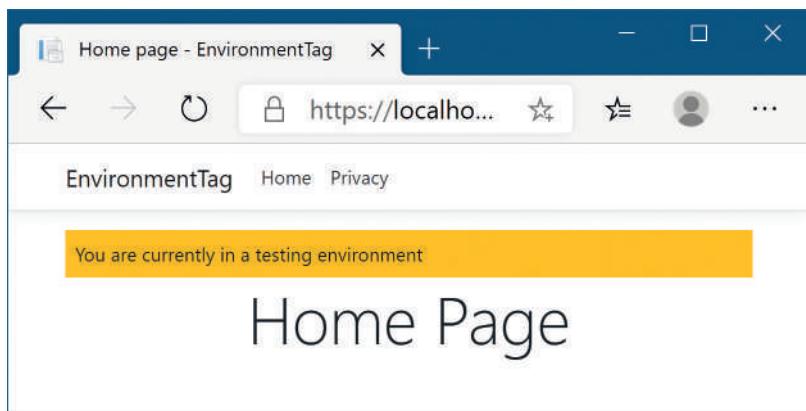


Рис. 18.11 Предупреждающий баннер будет отображаться всякий раз, когда вы работаете в тестовом окружении, чтобы его легко было отличить от промышленного окружения

Вы уже видели, как использовать C# для добавления операторов `if` в разметку, поэтому вполне возможно использовать эту технику для добавления дополнительного элемента `div` в разметку, когда текущее окружение имеет определенное значение. Если предположить, что переменная `env` содержит текущее окружение, то можно использовать что-то вроде этого:

```
@if(env == "Testing" || env == "Staging")
{
    <div class="warning">You are currently on a testing environment</div>
}
```

В этом нет ничего плохого, но лучше использовать тег-хелперы, чтобы ваша разметка была чистой и удобочитаемой. К счастью, ASP.NET Core поставляется с `EnvironmentTagHelper`, который можно использовать для достижения того же результата немного яснее:

```
<environment include="Testing,Staging">
    <div class="warning">You are currently on a testing environment</div>
</environment>
```

Этот тег-хелпер немного отличается от тех, что вы видели раньше. Вместо того чтобы дополнять существующий HTML-элемент с помощью атрибута `asp-`, *весь элемент* является тег-хеллером. Он полностью отвечает за создание разметки и использует атрибут для ее настройки.

Функционально этот тег-хелпер идентичен разметке C# (где переменная `env` содержит окружение размещения, как описано в главе 10), но он более декларативен по своей функции, нежели альтернатива C#. Очевидно, что вы можете использовать любой из подходов, но лично мне нравится HTML-подобная природа тег-хеллеров.

Мы подошли к концу главы, посвященной тег-хеллерам, а вместе с ней и нашему первому знакомству с созданием традиционных веб-приложений, отображающих HTML-код для пользователей. В последней части книги мы вернемся к шаблонам Razor, и вы узнаете, как создавать собственные компоненты, среди которых специальные тег-хеллеры и компоненты представления. На данный момент у вас есть все необходимое для создания сложных макетов Razor – специальные компоненты могут помочь привести ваш код в порядок.

Третья часть книги представляет собой краткий обзор того, как создавать приложения Razor Page с помощью ASP.NET Core. Теперь у вас есть основа, чтобы приступить к созданию простых приложений ASP.NET Core. Прежде чем мы перейдем к обсуждению безопасности в четвертой части, я уделю пару глав обсуждению создания приложений с помощью контроллеров MVC.

Я много говорил о контроллерах MVC, но в главе 19 вы узнаете, почему я рекомендую Razor Pages вместо контроллеров MVC для приложений с отрисовкой на стороне сервера. Тем не менее есть некоторые ситуации, в которых контроллеры MVC имеют смысл.

Резюме

- С помощью тег-хеллеров можно привязать свою модель данных к HTML-элементам, что упрощает создание динамического HTML-кода, который удобен для редактора;
- как и в случае с Razor в целом, тег-хеллеры предназначены только для отрисовки HTML-кода на стороне сервера. Нельзя использовать их непосредственно в таких фреймворках, как Angular или React;
- тег-хеллеры могут быть автономными элементами или могут быть прикреплены к существующему HTML-коду с помощью атрибутов. Это позволяет настраивать HTML-элементы и добавлять совершенно новые элементы;
- тег-хеллеры могут настраивать элементы, к которым они прикреплены, добавлять дополнительные атрибуты и настраивать способ своего отображения в HTML. Это может значительно уменьшить объем разметки, которую вам нужно писать;
- тег-хеллеры могут отображать несколько атрибутов в одном элементе. Это упрощает настройку тег-хеллера, поскольку можно установить несколько отдельных значений;

- можно добавить атрибуты `asp-page` и `asp-page-handler` к элементу `<form>`, чтобы задать URL-адрес `action` с помощью функции генерации URL-адресов;
- значения маршрута для использования во время маршрутизации указываются с помощью тег-хелпера формы, используя атрибуты `asp-route-*`. Эти значения применяются для сборки конечного URL-адреса или передаются как данные запроса;
- тег-хелпер формы также создает скрытое поле, которое можно использовать для предотвращения CSRF-атак. Оно добавляется автоматически и является важной мерой для обеспечения безопасности;
- можно прикрепить тег-хелпер метки к тегу `<label>` с помощью атрибута `asp-for`. Он генерирует соответствующий атрибут `for` и заголовок на основе атрибута `[Display]` `Data-Annotations` и имени свойства `PageModel`;
- тег-хелпер ввода задает для атрибута `type` элемента `<input>` соответствующее значение на основе `Type` привязанного свойства и всех примененных к нему `DataAnnotations`. Он также генерирует атрибуты `data-val-*`, необходимые для валидации на стороне клиента, что значительно сокращает объем HTML-кода, который вам нужно писать;
- чтобы активировать валидацию на стороне клиента, нужно добавить в представление необходимые файлы JavaScript для проверки с использованием `jQuery`;
- тег-хелпер раскрывающегося списка может создавать раскрывающиеся элементы `<select>`, а также списки прокрутки, используя атрибуты `asp-for` и `asp-items`. Чтобы создать элемент `<select>` множественного выбора, привяжите элемент к свойству `IEnumerable` модели представления. Можно использовать эти подходы для создания нескольких разных стилей поля выбора;
- элементы, поставляемые в атрибуте `asp-for`, должны быть `IEnumerable<SelectListItem>`. При попытке привязки другого типа вы получите ошибку компиляции в представлении Razor. Можно создать `IEnumerable<SelectListItem>` для enum `TEnum`, используя вспомогательный метод `Html.GetEnumSelectList<TEnum>()`. Это избавит вас от необходимости самостоятельного написания кода отображения;
- тег-хелпер раскрывающегося списка будет генерировать элементы `<optgroup>`, если у элементов, предоставленных в атрибуте `asp-for`, есть ассоциированный `SelectListGroup` в свойстве `Group`. Группы можно использовать для разделения элементов в списках выбора;
- любые дополнительные элементы `<option>`, добавленные в разметку Razor, будут переданы в окончательный HTML-код. Можно использовать эти дополнительные элементы для легкого добавления опции «без выбора» к элементу `<select>`;
- тег-хелпер сообщений валидации используется для визуализации сообщений об ошибках валидации для данного свойства на стороне клиента и на стороне сервера. Это обеспечивает важную

связь с вашими пользователями, когда в элементах есть ошибки. Используйте атрибут `asp-validation-for`, чтобы прикрепить этот тег-хелпер к элементу ``;

- тег-хелпер сводки сообщений валидации используется для отображения ошибок валидации для модели, а также для отдельных свойств. Можно использовать свойства на уровне модели для отображения дополнительной проверки, которая не применяется только к одному свойству. Используйте атрибут `asp-validation-summary`, чтобы присоединить этот тег-хелпер к элементу `<div>`;
- можно генерировать URL-адрес `<a>` с помощью тег-хелпера привязки ссылки. Он использует маршрутизацию для генерации URL-адреса `href` с использованием атрибутов `asp-page`, `asp-page-handler` и `asp-route-*`, предоставляя вам все возможности маршрутизации;
- вы можете добавить атрибут `asp-append-version` к элементам `<link>`, `<script>` и ``, чтобы обеспечить возможность сброса кеша в зависимости от содержимого файла. Это гарантирует, что пользователи смогут кешировать файлы по соображениям производительности, но при этом всегда получать их последнюю версию;
- можно использовать тег-хелпер окружения для условной визуализации разного HTML-кода на основе текущего окружения, в котором выполняется приложение. Его можно применять, чтобы при желании отображать совершенно разный HTML-код в разных окружениях.

19

Создание сайта с использованием контроллеров MVC

В этой главе:

- создание MVC-приложения;
- выбор между Razor Pages и контроллерами MVC;
- возврат представлений Razor из контроллеров MVC.

В этой книге я сосредоточился на Razor Pages вместо контроллеров MVC для HTML-приложений с отрисовкой на стороне сервера, поскольку считаю, что Razor Pages является предпочтительной парадигмой в большинстве случаев. В этой главе мы подробно рассмотрим, почему я считаю именно Razor Pages правильным выбором, и кратко изучим альтернативный вариант.

В разделе 19.2 мы создадим MVC-приложение по умолчанию, используя шаблон, чтобы вы могли ознакомиться с общей структурой проекта приложения. Мы рассмотрим некоторые различия между MVC-приложением и приложением Razor Pages, а также множество схожих черт.

Далее я подробно объясню, почему считаю Razor Pages более предпочтительной моделью приложений по сравнению с контроллерами

MVC. Вы узнаете об улучшенной эргономике разработки Razor Pages по сравнению с контроллерами MVC, а также случаи, в которых контроллеры MVC тем не менее являются правильным выбором.

В разделе 19.4 вы узнаете об отрисовке представлений Razor с использованием контроллеров MVC. Вы увидите, как фреймворк MVC использует соглашения для поиска файлов представлений и как их переопределить, выбрав для отрисовки определенный шаблон представления Razor. Наконец, вы увидите полный алгоритм выбора представления во всей его красе.

19.1 Razor Pages и MVC в ASP.NET Core

В этой книге основное внимание уделено Razor Pages, но я также упомянул, что Razor Pages «за кулисами» использует фреймворк MVC и что вы можете использовать его напрямую, если хотите. Кроме того, если вы создаете API для работы с мобильными или клиентскими приложениями и не хотите (или не можете) использовать минимальный API, то вполне можете использовать фреймворк MVC напрямую, создавая контроллеры веб-API.

ПРИМЕЧАНИЕ В главе 20 я расскажу, как создавать веб-API с помощью фреймворка MVC.

Итак, в чем разница между Razor Pages и MVC и что и когда следует выбирать?

Если вы новичок в ASP.NET Core, ответ довольно прост: используйте Razor Pages для приложений с отрисовкой на стороне сервера и минимальные API (или контроллеры веб-API) для создания API. У этого совета есть нюансы, которые я обсуждаю в разделе 19.5, но на данный момент это разделение сослужит вам хорошую службу.

И снова проблемы с названиями

У Microsoft долгая история создания фреймворков и присвоения им названий в честь обобщенной концепции: MVC, Web Forms, веб-страницы, Multi-Platform App UI и т. д. Честно говоря, невероятно, что Blazor выжил! Веб-API ничем не отличается.

В устаревшей версии ASP.NET компания Microsoft создала фреймворк веб-API, который по конструкции был похож на существующий фреймворк MVC, но также не был с ним совместимым. Таким образом, у нас были контроллеры MVC, которые представляли собой классы контроллеров, используемые в MVC для создания HTML и контроллеры веб-API, которые представляли собой классы контроллеров, используемые с фреймворком веб-API для создания кода в формате JSON или XML.

В ASP.NET Core Microsoft объединила эти два параллельных стека в один фреймворк ASP.NET Core MVC. Контроллеры в ASP.NET Core могут генерировать как HTML-код, так и файлы в формате JSON или XML; нет никакого разделения. Тем не менее контроллеры обычно предназначены для

генерации HTML или JSON/XML. По этой причине термины контроллер MVC и контроллер веб-API часто используются для обозначения двух основных видов контроллера: MVC для HTML и веб-API для JSON/XML.

В этой книге когда я говорю о контроллерах веб-API, то имею в виду стандартные контроллеры ASP.NET Core, генерирующие ответы API. В другом месте это можно описать как приложение веб-API, использующее контроллеры MVC, или как приложение веб-API. Все три случая относятся к одной и той же концепции: HTTP API, созданный с использованием контроллеров ASP.NET Core.

Однако, прежде чем мы сможем перейти к сравнениям, нужно кратко взглянуть на сам фреймворк MVC. Понимание сходства и различий между MVC и Razor Pages может быть очень полезным, поскольку в какой-то момент вы, вероятно, найдете применение MVC, даже если вы большую часть времени используете Razor Pages.

19.2 Наше первое веб-приложение MVC

В этом разделе вы узнаете, как создать свое первое веб-приложение MVC, которое отображает HTML-страницы на сервере с помощью контроллеров MVC и представлений Razor. Мы используем шаблон для создания приложения и сравним сгенерированный код, чтобы увидеть, чем он отличается от приложения Razor Pages.

Мы снова воспользуемся шаблоном, чтобы быстро запустить приложение. На этот раз мы будем использовать шаблон веб-приложения ASP.NET Core («модель–представление–контроллер»). Чтобы создать приложение в Visual Studio, выполните следующие действия:

- 1 Выберите **File** (Файл) > **Create** (Создать).
- 2 В диалоговом окне **Create a new project** (Создать новый проект) выберите шаблон **ASP.NET Core Web App (Model-View-Controller)**.
- 3 В диалоговом окне **Configure your new project** (Настройка нового проекта) введите имя проекта и просмотрите поле **Additional information** (Дополнительная информация), показанное на рис. 19.1.
- 4 Выберите **Create** (Создать). Если вы используете интерфейс командной строки (CLI), то можете создать аналогичный шаблон с помощью команды `dotnet new mvc`.

Шаблон MVC настраивает проект ASP.NET Core для использования контроллеров MVC с представлениями Razor. Как всегда, мы настраиваем приложение для использования контроллеров MVC в файле `Program.cs`, как показано в листинге 19.1. Если вы сравните этот шаблон со своими проектами Razor Pages, то увидите, что проект веб-API использует метод `AddControllersWithViews()` вместо `AddRazorPages()`. Контроллеры MVC отображаются как конечные точки путем вызова метода `MapControllerRoute()`. Этот метод отображает все контроллеры в приложении и настраивает для них обычный маршрут по умолчанию. Мы обсуждали традиционную маршрутизацию в главе 14, и вскоре я вернусь к ней снова.

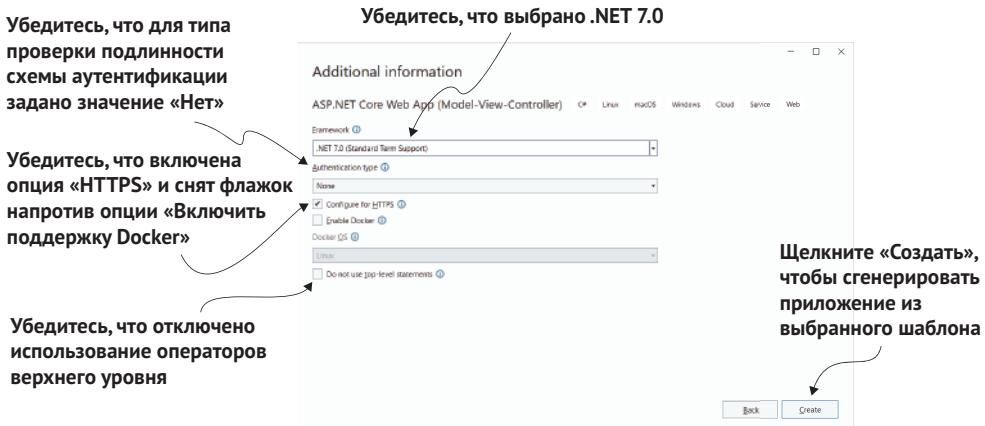


Рис. 19.1 Экран дополнительной информации для шаблона MVC. Он идет после диалогового окна «Настройка нового проекта» и позволяет настроить шаблон, который генерирует приложение

Листинг 19.1. Файл Program.cs для проекта MVC по умолчанию

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews(); <-- AddControllersWithViews добавляет сервисы для контроллеров MVC с представлениями Razor

WebApplication app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error"); <-- Путь обработчика исключений отличается от пути Razor Pages по умолчанию – /Error
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();

app.MapControllerRoute( <-- Добавляет все контроллеры MVC в приложение, используя маршрутизацию на основе соглашений
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}"); <-- Определяет шаблон маршрута на основе соглашений по умолчанию

app.Run();
```

Большая часть конфигурации приложения MVC такая же, как и для Razor Pages. Конфигурация промежуточного ПО по существу идентична, что неудивительно, учитывая, что MVC и Razor Pages представляют собой приложения одного и того же типа: приложение с отрисовкой на стороне сервера, возвращающее HTML. Основное отличие, как вы увидите в разделе 19.3, заключается в структуре проекта.

Прежде чем идти дальше, запустите приложение MVC, нажав клавишу **F5** в Visual Studio или выполнив команду `dotnet run` в папке проекта.

Приложение должно выглядеть удивительно знакомым; по сути, оно идентично версии приложения Razor Pages, которое мы создали в главе 13, как показано на рис. 19.2.

Вывод приложения идентичен приложению Razor Pages по умолчанию, но инфраструктура, используемая для создания ответа, отличается. Вместо `PageModel` и обработчика страниц MVC использует концепцию *контроллеров* и *методов действий*. В следующем листинге показан класс `HomeController` из приложения по умолчанию. Каждый неабстрактный открытый метод – это действие, которое выполняется в ответ на запрос.

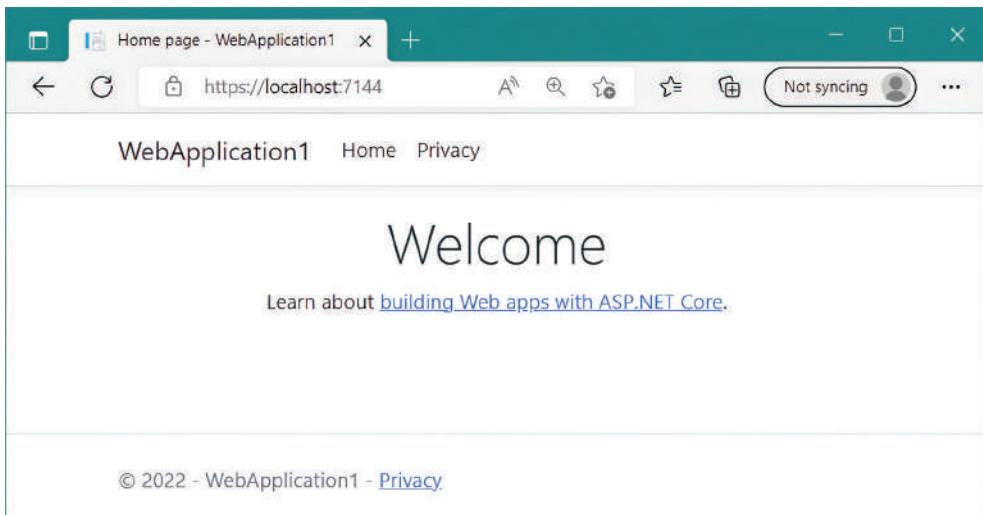


Рис. 19.2 Приложение MVC по умолчанию. Полученное приложение идентично эквиваленту Razor Pages, созданному в главе 13.

Можно гарантировать, что метод-кандидат не будет рассматриваться как метод действия, декорировав его атрибутом `[NonAction]`.

Листинг 19.2 Класс HomeController для приложения MVC по умолчанию

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }
    public IActionResult Index()
    {
        return View();
    }
}
```

Контроллеры MVC часто наследуют от базового класса `Controller`

Методы действий – это конечные точки, которые запускаются в ответ на запросы

Возврат `View()` отображает представление Razor

```

public IActionResult Privacy()
{
    return View();
}

[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
    NoStore = true)]
public IActionResult Error()
{
    return View(new ErrorViewModel
    {
        RequestId = Activity.Current?.Id
            ?? HttpContext.TraceIdentifier
    });
}

```

Вы можете применять фильтры к действиям, как вы узнаете в главах 21 и 22

Любой объект, возвращаемый с помощью View, передается в представление Razor как модель представления

ОПРЕДЕЛЕНИЕ *Действие* (или *метод действия*) – это метод, который запускается в ответ на запрос. *Контроллер MVC* – это класс, который содержит один или несколько логически сгруппированных методов действий.

Каждый из трех методов действия вызывает метод `View()` и возвращает результат. Так мы возвращаем объект `ViewResult`, который дает указание фреймворку MVC выполнить отрисовку представления Razor для действия.

Подробнее об этом процессе вы узнаете в разделе 19.4. Метод действия `Error` также устанавливает объект при вызове `View()`. Это модель представления, которая передается в представление Razor при отрисовке.

ПРИМЕЧАНИЕ. Контроллеры MVC используют явные модели представления для передачи данных в представление Razor, а не предоставляют данные как свойства самих себя (как Razor Pages делает это с моделями страниц). Это обеспечивает более четкое, чем в Razor Pages, разделение между различными «моделями», хотя в обоих случаях используется один и тот же общий паттерн проектирования MVC.

Еще одно большое различие между Razor Pages и контроллерами MVC заключается в том, что контроллеры MVC обычно используют маршрутизацию на основе соглашений, а не явную маршрутизацию, используемую Razor Pages. Я коснулся маршрутизации на основе соглашений и ее отличий от явной маршрутизации в главе 14, но вы можете увидеть ее в действии в этом приложении MVC.

Маршрутизация на основе соглашений определяет один или несколько паттернов шаблонов маршрутов, которые используются для всех контроллеров MVC в приложении. Шаблон маршрута по умолчанию, показанный в листинге 19.1, состоит из трех необязательных сегментов:

```
"{controller=Home}/{action=Index}/{id?}"
```

Маршруты на основе соглашений должны описывать, какой контроллер и действие должны выполняться для того или иного запроса, поэтому должны включать как минимум параметры контроллера и маршрута действия. При получении запроса ASP.NET Core сопоставляет шаблон маршрута и на его основе вычисляет, какой контроллер MVC и метод действия использовать. Например, маршрут по умолчанию будет соответствовать всем следующим URL-адресам:

- /Home/Privacy – выполняет действие HomeController.Privacy();
- /Home – выполняет действие HomeController.Index();
- /customer/list – выполняет действие CustomerController.List();
- /products/view/123 – выполняет действие ProductsController.View() с параметром маршрута id=123.

При маршрутизации на основе соглашений один шаблон маршрута сопоставляется с несколькими конечными точками, тогда как при явной маршрутизации один или несколько шаблонов маршрутов обычно сопоставляются с одной конечной точкой.

В обоих случаях есть свои тонкости, но в целом маршрутизация на основе соглашений более лаконична, а явная маршрутизация более выразительна.

Как я упоминал в главе 14, далее в этой книге я не буду обсуждать маршрутизацию на основе соглашений. Часто она используется только с контроллерами MVC, но даже в этом случае я обычно предпочитаю использовать явную маршрутизацию с атрибутами. Как использовать маршрутизацию на основе атрибутов, рассказывается в главе 20, когда мы будем обсуждать контроллеры веб-API.

Познакомившись с базовым приложением MVC, вы, вероятно, заметите множество сходств и различий между фреймворком MVC и Razor Pages. В следующем разделе мы рассмотрим один из аспектов этих черт: контроллеры MVC и их эквивалент в Razor Page: PageModel.

19.3 Сравнение контроллера MVC с PageModel из Razor Page

В главе 13 мы рассмотрели паттерн проектирования MVC и то, как он применяется к страницам Razor в ASP.NET Core. Возможно, неудивительно, что вы можете использовать контроллеры MVC с паттерном проектирования MVC почти таким же образом.

Как упоминалось в разделе 19.2, контроллеры и действия MVC аналогичны PageModel и обработчикам страниц в Razor Pages. Рисунок 19.3 проясняет это; это контроллер MVC, эквивалентный версии Razor Pages из главы 13.

В главе 13 я показал простую модель PageModel для отображения всех элементов дел в заданной категории в приложении со списком дел. В следующем листинге для удобства воспроизводится код Razor Pages из листинга 13.5.

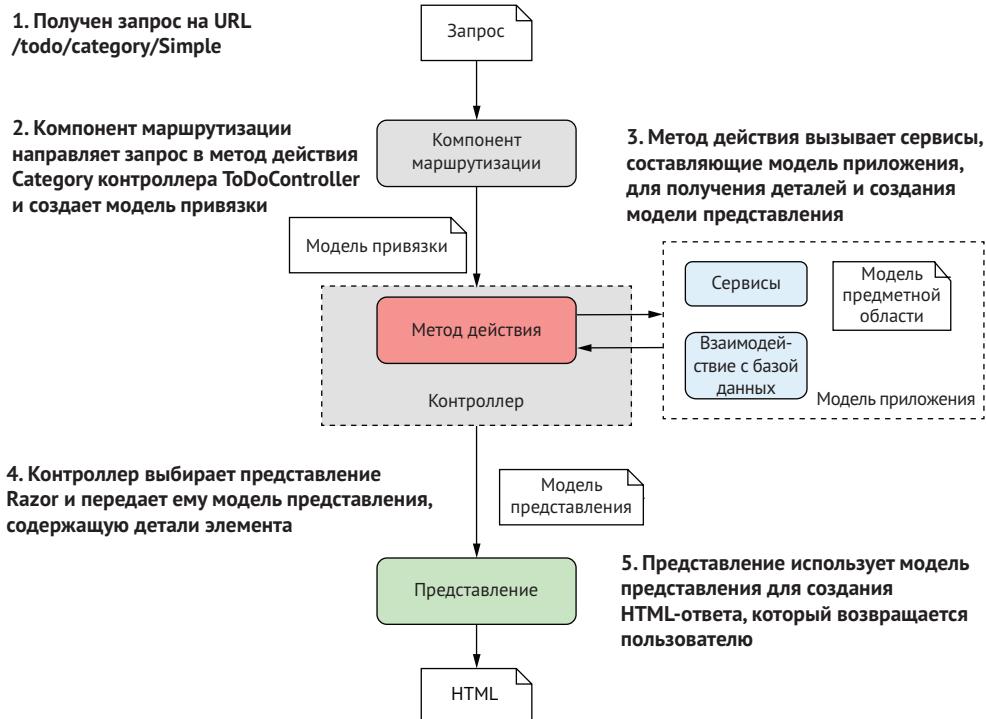


Рис. 19.3 Полный запрос контроллера MVC для категории. Паттерн контроллера MVC почти идентичен шаблону Razor Pages, показанному на рис. 13.12. Контроллер эквивалентен странице Razor, а действие эквивалентно обработчику страницы

Листинг 19.3 Страница Razor для просмотра всех дел в заданной категории

```
public class CategoryModel : PageModel
{
    private readonly ToDoService _service;
    public CategoryModel(ToDoService service)
    {
        _service = service;
    }
    public ActionResult OnGet(string category)
    {
        Items = _service.GetItemsForCategory(category);
        return Page();
    }

    public List<ToDoListModel> Items { get; set; }
}
```

MVC-эквивалент этой страницы Razor Page показан в листинге 19.4. Во фреймворке MVC контроллеры часто используются для объединения похожих действий, поэтому в данном случае контроллер называется `ToDoController`, поскольку обычно он содержит дополнительные

методы действий для работы с делами, например действия для просмотра определенного элемента или создания нового.

Листинг 19.4 Контроллер MVC для просмотра всех дел в заданной категории

```
public class ToDoController : Controller
{
    private readonly ToDoService _service;
    public ToDoController(ToDoService service)
    {
        _service = service;
    }
    public ActionResult Category(string id)
    {
        var items = _service.GetItemsForCategory(id);
        return View(items);
    }
    Возвращает ViewResult, указывающий, что представление Razor должно быть отображено, передавая модель представления.
}
public ActionResult Create(ToDoListModel model)
{
    // ...
}
```

ToDoService предоставляетя в конструкторе контроллера с использованием внедрения зависимостей

Метод действия «Категория» принимает параметр id

Метод действия вызывает ToDoService для получения данных и построения модели представления

Контроллеры MVC часто содержат несколько методов действий, которые отвечают на разные запросы

За исключением некоторых различий в именах, `ToDoController` похож на эквивалент `Razor Page` из листинга 19.3:

- оба используют внедрение зависимостей для доступа к сервисам;
- оба обработчика (обработчик страницы и метод действия) принимают созданные при помощи привязки модели параметры одним и тем же способом;
- оба одинаково взаимодействуют с моделью приложения для обработки запроса;
- оба создают модель представления для отрисовки представления `Razor`.

Одно из основных различий между `Razor Pages` и контроллерами `MVC` заключается в последнем этапе: отрисовке представления `Razor`. В следующем разделе вы увидите, как визуализировать представления `Razor` с помощью действий контроллера `MVC`, чем эти представления отличаются от представлений `Razor`, которые вы видели, и как фреймворк находит правильное представление `Razor` для отрисовки.

19.4 Выбор представления из контроллера MVC

В этом разделе:

- как контроллеры `MVC` используют объекты `ViewResult` для отрисовки представлений `Razor`;
- как создать новое представление `Razor`;
- как фреймворк находит представление `Razor` для отрисовки.

Одно из основных различий между контроллерами MVC и Razor Pages заключается в том, как обработчик страницы или метод действия выбирает представление Razor для отрисовки. В случае с Razor Pages это легко; страница отображает представление Razor, связанное со страницей. Что касается контроллеров MVC, то здесь все сложнее, поэтому важно понимать, как вы выбираете, какое представление нужно визуализировать после выполнения метода действия. На рис. 19.4 показано увеличенное изображение этого процесса сразу после того, как действие вызвало модель приложения и получило обратно некие данные.

Некоторые из этих цифр должны быть вам знакомы; это нижняя половина рис. 19.3 (с парой дополнений). На нем показано, что метод действия контроллера MVC использует объект `ViewResult`, чтобы указать, что представление Razor должно быть отображено. Этот объект содержит имя шаблона представления Razor для отрисовки и модель представления – произвольный класс РОСО, содержащий данные для отрисовки.

ПРИМЕЧАНИЕ `ViewResult` – это MVC-эквивалент `PageResult` страницы Razor. Основное отличие состоит в том, что `ViewResult` включает имя представления для визуализации и модель для передачи в шаблон представления, тогда как `PageResult` всегда отображает связанное представление страницы Razor и всегда передает `PageModel` в шаблон представления.

После возврата объекта `ViewResult` из метода действия поток управления возвращается к фреймворку MVC, который использует эвристику для поиска представления на основе предоставленного имени шаблона.

Обнаружив шаблон представления Razor, движок Razor передает модель представления из объекта `ViewResult` в представление и заполняет шаблон для создания окончательного HTML-кода. Этот последний шаг – отрисовка HTML-кода – по сути представляет собой тот же процесс, что мы наблюдали в Razor Pages.

В Visual Studio можно добавить новый шаблон представления Razor в обозревателе решений, щелкнув правой кнопкой мыши по папке, в которую вы хотите добавить представление. Выберите **Add** (Добавить) > **New Item** (Новый элемент), а затем выберите **Razor View – Empty** в диалоговом окне, как показано на рис. 19.5. Если вы не используете Visual Studio, создайте новый пустой файл в папке Views с расширением `.cshtml`.

Файлы представлений Razor практически идентичны файлам `.cshtml`, которые вы видели в главе 17. Единственная разница состоит в том, что файлы представлений Razor не должны указывать директиву `@page` в верхней части файла. В остальном они идентичны; вы можете использовать тот же синтаксис, частичные представления, макеты и модели представлений, что и в Razor Pages.



Рис. 19.4 Процесс генерации HTML-кода из контроллера MVC с использованием объекта ViewResult. Это очень похоже на процесс для страницы Razor. Основное отличие состоит в том, что в случае с Razor Pages представление является неотъемлемой частью страницы Razor; в случае с контроллерами MVC представление должно быть найдено во время выполнения

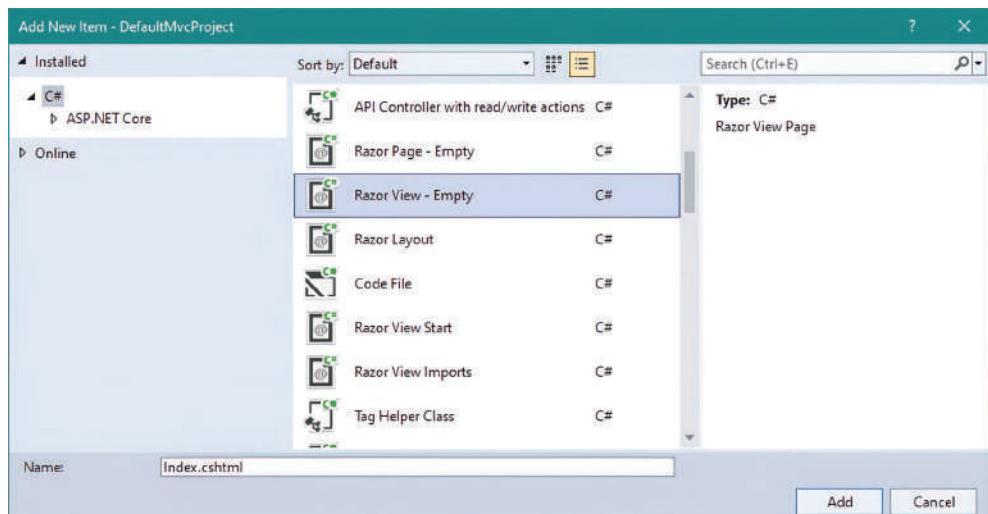


Рис. 19.5 Диалоговое окно «Добавить новый элемент». Выберите **Razor View – Empty** и добавьте новый файл шаблона представления Razor в приложение

Например, в следующем листинге показана часть представления Razor, Error.cshtml, для шаблона MVC по умолчанию. Все это можно распознать как стандартный синтаксис Razor.

Листинг 19.5 Представление Razor

```
@model ErrorViewModel
{
    ViewData["Title"] = "Error";
}

<h1 class="text-danger">Error.</h1>
<h2 class="text-danger">An error occurred while
    processing your request.</h2>

@if (Model.ShowRequestId)
{
    <p>
        <strong>Request ID:</strong> <code>@Model.RequestId</code>
    </p>
}
```

К представлению Razor можно привязывать модель

Можно выполнять произвольные выражения C# и использовать ViewData

Можно напрямую писать в вывод стандартный HTML

Можно использовать обычные конструкции управления Razor, а модель представления доступна через Model

Выражения C# должны начинаться с символа @

Создав шаблон представления, нужно его вызывать. В большинстве случаев нельзя создавать объект ViewResult непосредственно в методах действий. Вместо этого воспользуйтесь одним из вспомогательных методов View базового класса Controller. Эти методы упрощают передачу модели представления и выбор шаблона представления, но в них нет ничего волшебного: все, что они делают, – это создают объекты ViewResult.

В простейшем случае можно вызвать метод View без каких-либо аргументов, как показано в следующем листинге, взятом из приложения MVC по умолчанию. Вспомогательный метод View() возвращает объект ViewResult, который использует соглашения для поиска шаблона представления для отрисовки и не предоставляет модель представления при выполнении представления.

Листинг 19.6 Возвращаем объект ViewResult из метода действия с использованием соглашений по умолчанию

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Наследование от базового класса Controller делает доступными вспомогательные методы View

Вспомогательный метод View возвращает объект ViewResult

В этом примере вспомогательный метод View возвращает объект ViewResult без указания имени запускаемого шаблона. Имя используемого шаблона основано на имени контроллера и имени метода действия. Учитывая, что контроллер называется HomeController, а метод называется Index, по умолчанию шаблонизатор Razor ищет шаблон в расположении Views/Home/Index.cshtml, как показано на рис. 19.6.

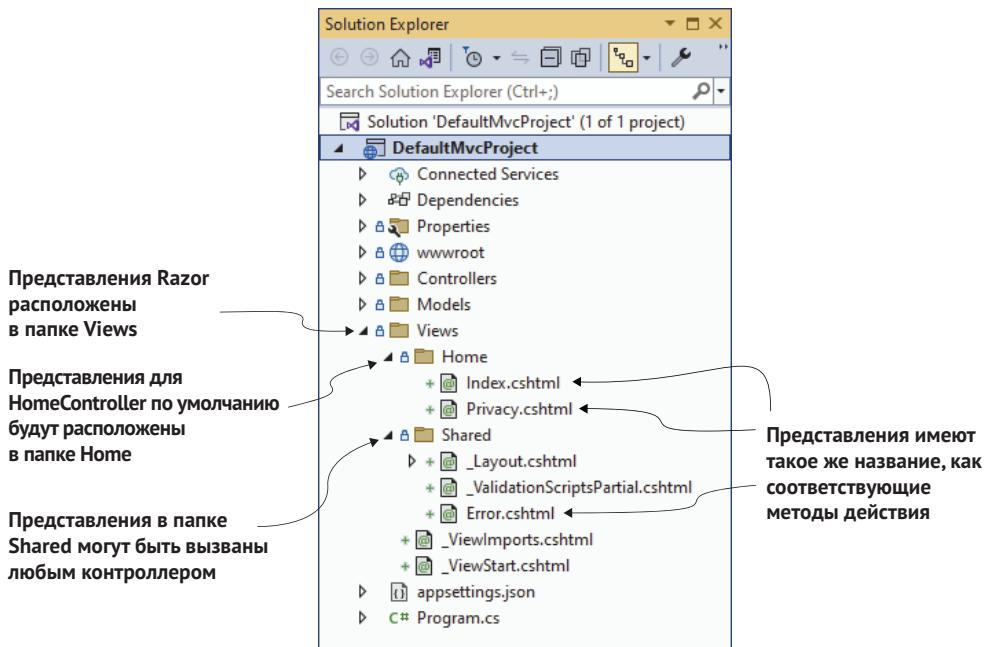


Рис. 19.6 Поиск файлов представлений во время выполнения производится в соответствии с соглашениями об именах. Файлы представления Razor находятся в папке на основе имени ассоциированного контроллера MVC и носят имя метода действия, который их запрашивал. Представления в папке Shared могут использоваться любым контроллером

Это еще один случай использования соглашений в MVC, чтобы уменьшить количество шаблонного кода, который вам нужно написать. Как всегда, соглашения необязательны. Вы также можете явно передать имя шаблона для выполнения в виде строки методу `View`. Например, если метод `Index` возвратил `View("ListView")`, шаблонизатор будет искать шаблон с именем `ListView.cshtml`. Вы даже можете указать полный путь к файлу представления относительно корневой папки своего приложения, например `View("Views/global.cshtml")`, который будет искать шаблон в расположении `Views/global.cshtml`.

ПРИМЕЧАНИЕ При указании абсолютного пути к представлению необходимо включить в путь и папку верхнего уровня `Views`, и файл с расширением `.cshtml`. Это напоминает правила поиска шаблонов частичных представлений.

Процесс поиска представления MVC Razor очень похож на процесс поиска частичного представления для отрисовки, как вы видели в главе 17. Фреймворк выполняет поиск в нескольких местах, чтобы найти запрашиваемое представление. Разница состоит в том, что в случае с Razor Pages процесс поиска выполняется только для отрисовки частичного представления, поскольку основное представление Razor, которое нужно визуализировать, уже известно – это шаблон представления страницы Razor.

На рис. 19.7 показан полный процесс, используемый фреймворком MVC для поиска правильного шаблона представления для выполнения, когда объект `ViewResult` возвращается из контроллера MVC. Может быть найдено несколько шаблонов, например если файл `Index.cshtml` присутствует в папках `Home` и `Shared`. Аналогично правилам поиска частичных представлений, движок будет использовать первый найденный шаблон.

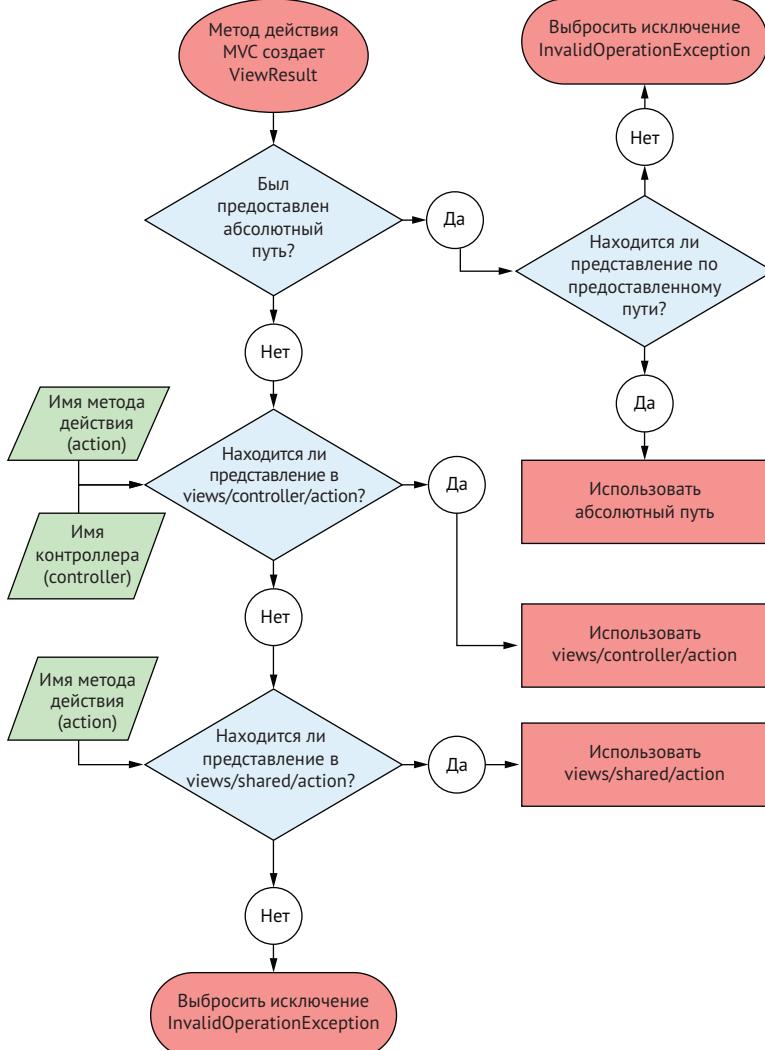


Рис. 19.7 Блок-схема, описывающая, как шаблонизатор Razor находит правильный шаблон представления для выполнения. Избегайте сложности этой диаграммы. Это одна из причин, по которой я рекомендую использовать Razor Pages везде, где это возможно!

СОВЕТ Можно изменить все эти соглашения, включая алгоритм, показанный на рис. 19.8, во время начальной настройки. Фак-

тически можно заменить весь шаблонизатор Razor, если вы и в самом деле захотите этого!

У вас может возникнуть соблазн явно указать имя файла представления, который вы хотите визуализировать в контроллере. Если это так, то я бы посоветовал побороть данное желание. Будет намного проще, если вы примете соглашения такими, какие они есть, и будете им следовать. Это распространяется на всех, кто просматривает ваш код; если вы будете придерживаться стандартных соглашений, им будет комфортно иметь дело с вашим приложением. А от этого будет только польза!

Помимо указания имени шаблона представления, вы также можете передать объект, который будет действовать как модель представления для представления Razor. Этот объект должен соответствовать типу, указанному в директиве представления `@model`, и доступ к нему осуществляется точно так же, как и для Razor Pages, – используя свойство `Model`.

СОВЕТ Все остальные способы передачи данных в представление, описанные в главе 17, доступны и в контроллерах MVC. Обычно следует отдавать предпочтение модели представления, где это возможно, но вы также можете использовать, например, сервисы `ViewData`, `TempData` или `@inject`.

В следующем листинге показаны два примера передачи модели представления в представление.

Листинг 19.7 Возвращаем объект `ViewResult` из метода действия с использованием соглашений по умолчанию

```
public class ToDoController : Controller
{
    public IActionResult Index()
    {
        var listViewModel = new ToDoListModel(); ← Создание экземпляра модели представления для передачи в представление Razor
        return View(listViewModel); ← Модель представления передается View в качестве аргумента
    }
    public IActionResult View(int id)
    {
        var viewModel = new ViewToDoModel(); ← Одновременно с моделью представления можно указать имя шаблона представления
        return View("ViewToDo", viewModel); ←
    }
}
```

После того как шаблон представления Razor будет найден, представление визуализируется с использованием синтаксиса Razor, который вы видели в этой главе. Вы можете использовать все уже знакомые вам функции, например макеты, частичные представления, `_ViewImports` и `_ViewStart`. С точки зрения представления Razor между представлением Razor Pages и представлением MVC Razor нет разницы.

Теперь после краткого обзора приложения MVC можно подробнее рассмотреть, когда следует выбирать контроллеры MVC вместо Razor Pages.

19.5 Выбор между Razor Pages и контроллерами MVC

На протяжении всей книги я говорил, что для приложений с отрисовкой на стороне сервера обычно следует выбирать Razor Pages вместо использования контроллеров MVC. В этом разделе я покажу разницу между Razor Pages и контроллерами MVC с точки зрения структуры проекта и обоснуй свои аргументы. Я также приведу случаи, когда контроллеры MVC являются хорошим выбором.

Если вы знакомы с устаревшей версией .NET Framework ASP.NET или более ранними версиями ASP.NET Core, возможно, вы уже знакомы и с контроллерами MVC. Если вы не уверены, стоит ли придерживаться того, что вы знаете, или перейти на Razor Pages, этот раздел поможет вам сделать выбор. Разработчики с таким опытом часто изначально имеют неправильное представление о Razor Pages (и я был в их числе!), неверно приравнивая их к веб-формам и упуская из виду лежащую в их основе основу фреймворка MVC. В этом разделе делается попытка внести ясность.

С архитектурной точки зрения Razor Pages и MVC, по сути, эквивалентны, поскольку оба используют паттерн проектирования MVC. Наиболее очевидные различия связаны с тем, где в проекте размещаются файлы, о чем и пойдет речь в следующем разделе.

19.5.1 Преимущества Razor Page

В предыдущем разделе я показал, что код контроллера MVC очень похож на код модели страницы `PageModel`. В таком случае какая польза от применения Razor Pages? В этом разделе мы обсудим некоторые болееевые точки контроллеров MVC и то, как Razor Pages пытается их решить.

В MVC один контроллер может иметь несколько методов действия. Каждое действие обрабатывает разные запросы и генерирует разные ответы. Группировка нескольких действий в контроллере несколько произвольна, но обычно она используется для группировки действий, связанных с определенной сущностью: в данном случае элементы списка дел. Более полная версия `ToDoController` из листинга 19.4 может включать в себя методы действий для перечисления всех элементов, например для удаления элементов и создания новых. К сожалению, часто можно обнаружить, что контроллеры становятся очень большими и раздутыми и у них большое количество зависимостей¹.

Razor Pages – это не Web Forms

Распространенный аргумент, который я слышу от разработчиков ASP.NET не в пользу Razor Pages: «О, это всего лишь Web Forms». Это мнение не соответствует действительности во многих отношениях, но оно достаточно распространено, поэтому рассмотрим его подробнее.

¹ Перед переходом на Razor Pages шаблон ASP.NET Core, включающий функциональность входа пользователя, содержал два таких контроллера, в каждом из которых было свыше 20 методов действий и более 500 строк кода!

Web Forms – это модель веб-программирования, которая была выпущена как часть .NET Framework 1.0 в 2002 году. Это попытка обеспечить высокопроизводительный опыт для разработчиков, впервые переходящих от разработки настольных приложений к веб-разработке.

Сейчас Web Forms сильно критикуют, но слабые стороны этой модели стали очевидными лишь в последнее время. Веб-формы пытались скрыть от вас сложности интернета, чтобы создать впечатление, что вы ведете разработку с помощью настольного приложения. Часто это приводило к тому, что приложения были медленными, с большим количеством взаимозависимостей, и их было трудно сопровождать.

Web Forms предоставляют модель программирования на основе страниц, поэтому Razor Pages иногда ассоциируется с ними. Однако, как вы видели, Razor Pages основан на паттерне проектирования MVC и предоставляет доступ к внутренней веб-функциональности, не пытаясь скрыть ее от вас.

Razor Pages оптимизирует определенные пути, используя соглашения, но не пытается создать модель приложения с состоянием поверх веб-приложения без состояния, как это делали Web Forms.

Если вы были поклонником модели приложений с состоянием Web Forms, вам следует обратить внимание на Blazor Server, который использует аналогичную парадигму и вместе с тем органично задействует веб-технологии, а не борется с ними. Подробнее о сходстве можно прочитать на странице <http://mng.bz/7Dy9>.

ПРИМЕЧАНИЕ Необязательно делать контроллеры такими большими. Это распространенный шаблон. Вместо этого можно, например, создать отдельный контроллер для каждого действия.

Еще один минус контроллеров MVC – их типичная организация в вашем проекте. Большинству методов действий в контроллере потребуется связанное представление Razor и модель представления для передачи данных в представление. В MVC классы традиционно группируются по типу (контроллер, представление, модель представления), тогда как в Razor Page группировка идет по функциям – все, что связано с определенной страницей, размещается в одном месте.

На рис. 19.8 сравнивается макет файла для простого проекта Razor Pages с эквивалентом MVC. Использование Razor Pages означает гораздо меньшую прокрутку вверх и вниз между контроллером, представлениями и папками моделей представлений при работе с определенной страницей. Все, что вам нужно, находится в двух файлах: представлении Razor .cshtml и (вложенном) файле PageModel.cshtml.cs.

Между MVC и Razor Pages есть дополнительные различия, которые я буду освещать на протяжении всей книги, но эта разница в макете действительно является самым большим выигрышем. Razor Pages учитывает тот факт, что вы создаете страничное приложение, и оптимизирует рабочий процесс, объединяя все, что связано с одной страницей.

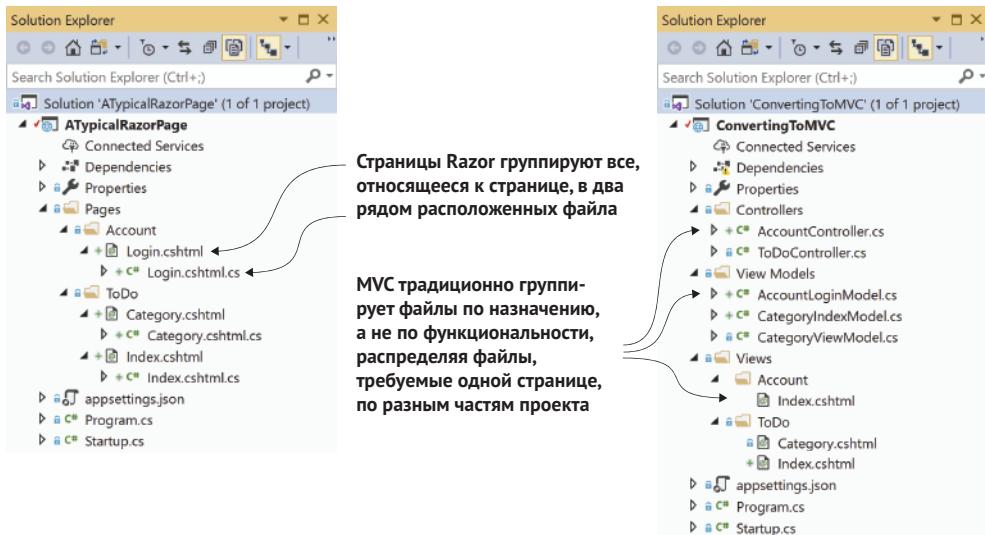


Рис. 19.8 Сравнение структуры папок для проекта MVC со структурой папок для проекта Razor Pages

СОВЕТ Можно рассматривать каждую страницу Razor как мини-контроллер, сфокусированный на одной странице. Обработчики страниц с функциональной точки зрения эквивалентны методам действия контроллера MVC.

Этот макет также имеет преимущество, которое состоит в том, что каждая страница превращается в отдельный класс. Это контрастирует с MVC, где каждая страница превращается в действие в данном контроллере. Каждая страница Razor связана с определенной функцией, например отображением списка дел. Контроллеры MVC содержат методы действий, которые обрабатывают несколько различных функций для более абстрактной концепции, например все функции, относящиеся к элементам списка дел.

ПРИМЕЧАНИЕ ASP.NET Core предельно детально настраивается, поэтому не нужно группировать файлы MVC-приложения по типам; это всего лишь настройка по умолчанию и простой путь. Фактически если вы решите использовать контроллеры MVC, я настоятельно рекомендую группировать их с помощью папок с разделением по функциональности. Неплохое введение по этой теме можно найти на странице <http://mng.bz/mVOr>.

Еще один важный момент заключается в том, что Razor Pages не теряет разделения ответственности, которое есть в MVC. Представление Razor Pages по-прежнему связано только с отрисовкой HTML, а обработчик – это координатор, который обращается к модели приложения. Единственное реальное отличие – это отсутствие явной модели представления, которая есть в MVC, но ее вполне возможно скомпилировать в Razor Pages, если вам это мешает.

Преимущества использования Razor Pages особенно заметны, когда у вас есть «содержательные» веб-сайты, маркетинговые веб-сайты, на которых вы в основном отображаете статические данные, а реальная логика отсутствует. В этом случае MVC добавляет сложности без каких-либо реальных преимуществ, поскольку в контроллерах вообще нет никакой логики. Еще один отличный вариант использования – создание форм для пользователей для отправки данных. Razor Pages специально оптимизирован для этого сценария, в чем вы убедитесь в следующих главах.

Ясно, что я поклонник Razor Pages, но нельзя сказать, что это идеальный вариант для любой ситуации. В следующем разделе мы обсудим случаи, когда можно использовать контроллеры MVC в своем приложении.

Имейте в виду, что это не вариант «либо-либо» – можно одновременно использовать и контроллеры MVC, и Razor Pages, и даже минимальные API в одном приложении, и во многих случаях это может оказаться лучшим вариантом.

19.5.2 Когда выбирать контроллеры MVC вместо Razor Pages

Razor Pages отлично подходят для создания многостраничных приложений с отрисовкой на стороне сервера. Но не все приложения соответствуют данному шаблону, и даже некоторые приложения, которые все же попадают в эту категорию, лучше всего разрабатывать с использованием контроллеров MVC вместо Razor Pages. Вот несколько таких сценариев:

- *когда вам не нужна отрисовка представлений* – Razor Pages лучше всего подходит для многостраничных приложений, где вы отрисовываете представление для пользователя. Если вы создаете HTTP API, то следует использовать минимальные API или контроллеры MVC (веб-API). Вы познакомитесь с контроллерами веб-API в главе 20;
- *когда вы конвертируете существующее приложение MVC в ASP.NET Core* – если у вас уже есть приложение ASP.NET, использующее MVC, вероятно, не стоит преобразовывать существующие контроллеры MVC в Razor Pages. Имеет смысл сохранить существующий код и, возможно, подумать о разработке *нового* приложения с помощью Razor Pages;
- *когда вы делаете много частичных обновлений страницы* – можно использовать JavaScript в приложении, чтобы избежать полной навигации по странице, обновляя только часть страницы за раз. Такой подход, находящийся на полпути между полной отрисовкой на стороне сервера и клиентским приложением, возможно, проще реализовать с помощью контроллеров MVC, чем использовать Razor Pages. С другой стороны, можно легко смешивать Razor Pages и контроллеры MVC, используя Razor Pages, где это необходимо, и контроллеры MVC для частичных представлений.

Я надеюсь, что к этому моменту вы уже увлеклись Razor Pages и их общим дизайном с использованием паттерна MVC. Тем не менее в некоторых ситуациях использование контроллеров MVC имеет смысл, поэтому стоит иметь это в виду. Еще один важный момент, который

следует помнить, – вы можете включить в одно приложение и контроллеры MVC, и Razor Pages, если они вам нужны.

Когда не следует использовать Razor Pages или контроллеры MVC

Обычно для написания большей части логики приложения вы будете использовать либо Razor Pages, либо контроллеры MVC. Вы будете применять их для определения API и страниц в вашем приложении, а также того, как они взаимодействуют с вашей бизнес-логикой. Razor Pages и MVC предоставляют обширный фреймворк со множеством функций, помогающих быстро и эффективно создавать приложения. Но они подходят не для всех приложений.

Такая обширная функциональность непременно сопряжена с определенными накладными расходами на производительность. Для типичных бизнес-приложений удобство при использовании MVC или Razor Pages сильно перевешивает любое влияние на производительность. Но если вы создаете JSON API, вам, вероятно, захочется рассмотреть минимальные API для повышения производительности. Для межсерверных API или небраузерных клиентов может подойти альтернативный протокол, например gRPC (<https://docs.microsoft.com/aspnet/core/grpc>). Вы также можете рассмотреть такие протоколы, как GraphQL, как описано в статье Валерио Де Санктиса «Создание веб-API в ASP.NET Core» (Manning, 2023).

В качестве альтернативы, если вы создаете приложение, функционирующее в режиме реального времени, вы, вероятно, захотите рассмотреть возможность использования WebSockets вместо традиционных HTTP-запросов. Для добавления в приложение подобной функциональности можно использовать ASP.NET Core SignalR. ASP.NET Core SignalR предоставляет абстракцию поверх WebSockets. SignalR также предоставляет простой механизм «отката транспорта» (переключения с WebSockets на доступную технологию, например Long polling), а еще модель приложения удаленного вызова процедур (RPC). Дополнительные сведения см. в документации: <https://docs.microsoft.com/ru-ru/aspnet/core/signalr>.

Еще один вариант, доступный в ASP.NET Core 7, – это Blazor. Данный фреймворк позволяет создавать интерактивные клиентские веб-приложения через использование стандарта WebAssembly для исполнения кода .NET непосредственно в браузере либо через модель с отслеживанием состояния и SignalR. Дополнительную информацию см. в книге Криса Сэйнти «*Blazor в действии*» (Manning, 2022).

Вы узнали о контроллерах MVC как альтернативе Razor Pages, а в первой части книги вы узнали об использовании минимальных API для создания API JSON. Контроллеры веб-API находятся где-то посередине; они используют контроллеры MVC, но генерируют данные в формате JSON и других машиночитаемых форматах, а не HTML. В главе 20 вы узнаете, почему можно использовать контроллеры веб-API вместо минимальных API и как создать приложение веб-API.

Резюме

- Действие (или метод действия) – это метод, который запускается в ответ на запрос. Контроллер MVC – это класс, который содержит один или несколько логически сгруппированных методов действий;
- чтобы использовать контроллеры MVC в приложении ASP.NET Core, вызовите метод `AddControllersWithViews()` в вашем `WebApplicationBuilder`. Так вы добавите все необходимые сервисы для контроллеров MVC и отрисовки представлений Razor в контейнер внедрения зависимостей;
- контроллеры MVC обычно используют традиционную маршрутизацию для выбора контроллера MVC и метода действия. Вместо того чтобы связывать шаблон маршрута с каждым методом действия в приложении, маршрутизация на основе соглашений задает один или несколько паттернов шаблонов маршрутов, которые сопоставляются с несколькими конечными точками. Маршруты на основе соглашений должны определять контроллер и параметр маршрута действия, чтобы определить действие, которое необходимо выполнить;
- вы можете возвращать экземпляры `IActionResult` из контроллеров MVC, и они обрабатываются так же, как и в Razor Pages. Наиболее часто возвращаемым типом является `ViewResult` с использованием вспомогательного метода `View()`, который сообщает фреймворку отображать представление Razor;
- `ViewResult` может содержать имя представления для отрисовки и, при необходимости, объект модели представления, который будет использоваться при отрисовке представления. Если имя представления не указано, представление выбирается с использованием соглашений;
- по соглашению представления MVC Razor называются так же, как и метод действия, который их вызывает. Они находятся либо в папке с тем же именем, что и контроллер метода действия, либо в папке `Shared`;
- контроллеры MVC содержат несколько методов действий, обычно сгруппированных вокруг объекта или ресурса высокого уровня. Напротив, Razor Pages группирует все обработчики страниц для одной страницы в одном месте, вокруг страницы или функции, а не объекта. Это обеспечивает улучшенную эргономику разработчика при работе над конечной точкой;
- выбрать контроллеры MVC вместо Razor Pages может иметь смысл, если вы выполняете модернизацию приложения, которое уже использует контроллеры MVC, или если приложение использует много частичных обновлений страниц.

20

Создание HTTP API с использованием контроллеров веб-API

В этой главе:

- создание контроллера веб-API для возврата клиентам ответа в формате JSON;
- использование маршрутизации на основе атрибутов для настройки URL-адресов;
- создание ответа с использованием согласования содержимого;
- применение общих соглашений с помощью атрибута [ApiController].

В главах с 13 по 19 вы проработали каждый уровень приложения ASP.NET Core с отрисовкой на стороне сервера, используя Razor Pages и контроллеры MVC для отображения HTML в браузере. В первой части книги вы видели другой тип приложения ASP.NET Core, использующий минимальные API для обслуживания JSON для одностраничных приложений на стороне клиента или мобильных приложений. В этой главе вы узнаете о контроллерах веб-API, которые находятся где-то посередине!

Большую часть того, что вы уже узнали, можно применить к контроллерам веб-API; они используют ту же систему маршрутизации, что

и минимальные API, и тот же паттерн проектирования MVC, привязку модели и проверку валидности данных, что и Razor Pages и контроллеры MVC.

В этой главе вы узнаете, как определять контроллеры веб-API и действия, а также увидите, насколько они похожи на уже известные вам страницы и контроллеры Razor. Вы узнаете, как создать модель API для возврата данных и кодов состояния HTTP в ответ на запрос таким образом, чтобы их могли понимать клиентские приложения.

Изучив, как паттерн проектирования MVC применяется к контроллерам веб-API, вы увидите, как маршрутизация работает с веб-API. Мы посмотрим, как явная маршрутизация на основе атрибутов работает с методами действий, затрагивая многие из тех же концепций, которые мы рассмотрели в главах 6 и 14.

Одной из важных функций, добавленных в ASP.NET Core 2.1, был атрибут `[ApiController]`. Он применяет несколько общих соглашений, используемых в веб-API, что уменьшает объем кода, который вы должны писать самостоятельно. В разделе 20.5 вы узнаете, как автоматические сообщения `400 Bad Request` для недопустимых запросов, вывод параметров привязки модели и объекты `ProblemDetail` могут упростить создание API.

Вы также узнаете, как форматировать модели API, возвращаемые методами действий, используя согласование содержимого, чтобы гарантировать, что вы генерируете ответ, который может понять вызывающий клиент. В рамках этой темы вы узнаете, как добавить поддержку дополнительных форматов, таких как XML, чтобы можно было генерировать ответы в формате XML, если их запрашивает клиент.

Наконец, я расскажу о некоторых различиях между контроллерами API и приложениями с минимальным API, а также о том, когда следует выбирать одно из них. Прежде чем мы перейдем к этой теме, мы посмотрим, с чего начать. В разделе 20.1 вы увидите, как создать проект веб-API и добавить свой первый контроллер API.

20.1 Создание первого проекта веб-API

В этом разделе вы узнаете, как создать проект с веб-API в ASP.NET Core и свои первые контроллеры веб-API. Вы увидите, как использовать методы действий контроллера для обработки HTTP-запросов и объекты `ActionResult` для генерации ответа.

ПРИМЕЧАНИЕ Как я упоминал ранее, проект веб-API – это стандартный проект ASP.NET Core, использующий фреймворк MVC и контроллеры веб-API.

Некоторые думают, что паттерн проектирования MVC применяется только к приложениям, которые напрямую визуализируют пользовательский интерфейс, например представления Razor, которые вы видели в предыдущих главах. В ASP.NET Core паттерн MVC одинаково хорошо применяется при создании веб-API, но часть *представления* MVC предполагает создание удобного ответа для машины, а не для пользователя.

Параллельно с этим вы создаете контроллеры веб-API в ASP.NET Core точно так же, как создаете традиционные контроллеры MVC. Единственное, что их отличает с точки зрения кода, – это тип возвращаемых данных: контроллеры MVC обычно возвращают объект `ViewResult`, а контроллеры веб-API – низкоуровневые объекты .NET из своих методов действий или экземпляр `IActionResult`, например `StatusResult`, как вы видели в главе 15.

Вы можете создать новый проект веб-API в Visual Studio, используя тот же процесс, который вы видели ранее в Visual Studio. Выберите **File** (Файл) > **Create** (Создать) и в диалоговом окне **Create a new project** (Создать новый проект) выберите шаблон **ASP.NET Core Web API**. Введите имя своего проекта в диалоговом окне **Configure your new project** (Настройка нового проекта) и просмотрите поле **Additional Information** (Дополнительная информация), показанное на рис. 20.1, прежде чем выбрать **Create** (Создать). Если вы используете интерфейс командной строки (CLI), то можете создать аналогичный шаблон с помощью команды `dotnet new webapi`.

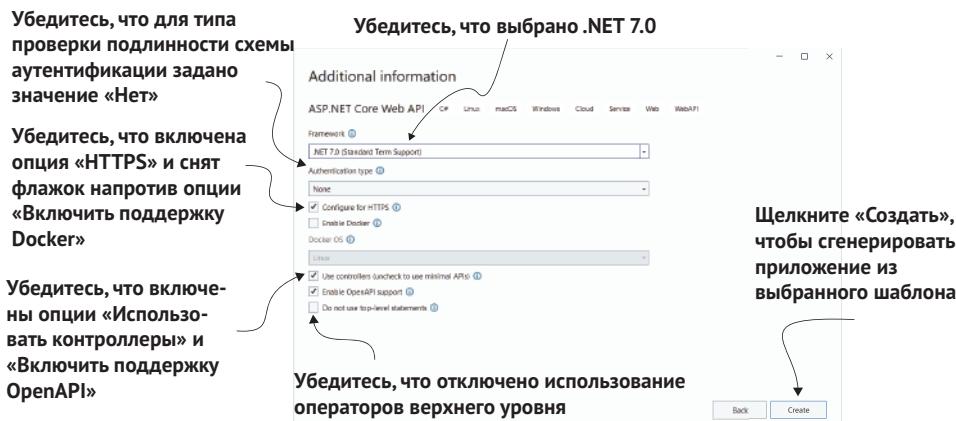


Рис. 20.1 Экран дополнительной информации. Этот экран следует за диалоговым окном «Настройка нового проекта» и позволяет настроить шаблон, который генерирует приложение

Шаблон веб-API настраивает проект ASP.NET Core для контроллеров веб-API только в файле `Program.cs`, как показано в листинге 20.1. Если вы сравните этот шаблон с проектом контроллера MVC из главы 19, то увидите, что вместо метода `AddControllersWithViews()` проект веб-API использует метод `AddControllers()`. При этом добавляются только сервисы, необходимые для контроллеров, но отсутствуют сервисы для отрисовки представлений Razor. Кроме того, контроллеры API добавляются с помощью метода `MapControllers()` вместо метода `MapControllerRoute()`, поскольку контроллер веб-API обычно использует явную маршрутизацию вместо маршрутизации на основе соглашений. Шаблон веб-API по умолчанию также добавляет сервисы и конечные точки OpenAPI, необходимые для Swagger UI, как было показано в главе 11.

Листинг 20.1 Файл Program.cs для проекта веб-API по умолчанию

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

WebApplication app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();    | Добавляет промежуточное ПО Swagger UI
                           | для изучения вашего веб-API.
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers(); <-- Метод MapControllers настраивает действия контроллера API в вашем приложении как конечные точки

app.Run();
```

Метод AddControllers добавляет в ваше приложение необходимые сервисы для контроллеров API. Добавляет сервисы, необходимые для создания документа спецификации Swagger/OpenAPI

Добавляет промежуточное ПО Swagger UI для изучения вашего веб-API.

Метод MapControllers настраивает действия контроллера API в вашем приложении как конечные точки

Код из листинга 20.1 инструктирует приложение найти все контроллеры веб-API и настроить их в EndpointMiddleware. Каждый метод действия становится конечной точкой и может получать запросы, когда RoutingMiddleware сопоставляет входящий URL-адрес с методом действия.

ПРИМЕЧАНИЕ Технически можно включать Razor Pages, минимальные API и контроллеры веб-API в одно приложение, но я предпочитаю по возможности хранить их отдельно. Существуют определенные аспекты (например, обработка ошибок и аутентификация), которые можно упростить, если разделить их. Конечно, у запуска двух отдельных приложений есть свои трудности!

Можно добавить контроллер веб-API в проект, создав новый файл с расширением .cs в любом месте проекта. Традиционно этот файл помещается в папку Controllers, но это не является техническим требованием.

СОВЕТ Архитектура вертикальных срезов и папки функций (к счастью) становится все более популярными в кругах .NET. Используя эти подходы, вы организуете свой проект на основе функций, а не технических концепций, таких как контроллеры и модели.

В листинге 20.2 показан пример простого контроллера с одной конечной точкой, который при выполнении возвращает I Enumerable<string>. В этом примере подчеркивается сходство с традиционными контроллерами MVC (с использованием методов действий и базового класса) и минимальными API (возвращающими объект генератор напрямую для последующей сериализации).

Листинг 20.2 Простой контроллер веб-API

```
[ApiController] <-- Атрибут [ApiController] подчиняется общим соглашениям
public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear",           Обычно используется сервис, внедренный
        "Lemon",         с помощью внедрения
        "Peach"          зависимостей
    };
    [HttpGet("fruit")]
    public IEnumerable<string> Index() <-- Атрибут [HttpGet] определяет шаблон маршрута, используемый для вызова действия
    {
        return _fruit; <-- Контроллер предоставляет отдельный метод действия, который возвращает список фруктов
    }
}
```

Класс ControllerBase предоставляет вспомогательные функции

Имя метода действия Index не используется для маршрутизации. Это может быть что угодно

При вызове эта конечная точка возвращает список строк, сериализованных в JSON, как показано на рис. 20.2.

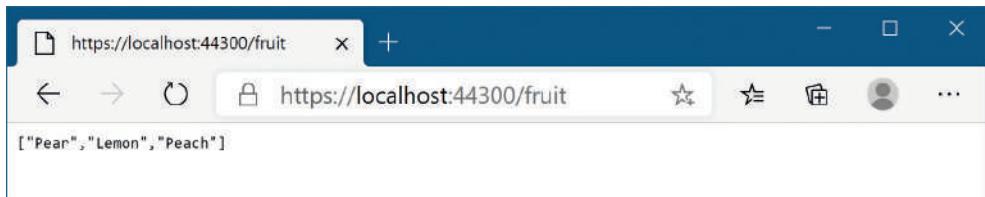


Рис. 20.2 Тестирование веб-API в листинге 20.2 путем доступа к URL-адресу в браузере. К URL-адресу /fruit отправляется GET-запрос, который возвращает List<string>, сериализованный в JSON

Веб-API обычно используют атрибут [ApiController] (появившийся в .NET Core 2.1) и являются производными от класса ControllerBase. Базовый класс предоставляет несколько вспомогательных методов для генерации результатов, а атрибут [ApiController] автоматически применяет некоторые общие соглашения, как вы увидите в разделе 20.5.

СОВЕТ Существует также базовый класс Controller, который обычно используется, когда вы применяете контроллеры MVC с представлениями Razor. Он не является обязательным для контроллеров веб-API, поэтому класс ControllerBase – наиболее подходящий вариант.

В листинге 20.2 видно, что метод действия Index возвращает список строк непосредственно из метода действия. Когда вы возвращаете данные из такого действия, то предоставляете модель API для запроса. Клиент получит эти данные. Они форматируются в соответствующий ответ, являющийся списком в формате JSON, как показано на рис. 20.2, и отправляются в браузер с кодом состояния 200 OK.

СОВЕТ Контроллеры веб-API по умолчанию форматируют данные в формат JSON. Вы увидите, как отформатировать возвращаемые данные другими способами, в разделе 20.6. Конечные точки минимального API, которые возвращают данные напрямую (а не через `IResult`), будут форматировать данные только в формате JSON; других вариантов нет.

URL-адрес, по которому предоставляется действие контроллера веб-API, обрабатывается так же, как и в случае с традиционными контроллерами MVC и страницами Razor, – используя маршрутизацию. Атрибут `[HttpGet("fruit")]`, примененный к методу `Index`, указывает, что метод должен использовать шаблон маршрута "fruit" и отвечать на GET-запросы. Подробнее о маршрутизации на основе атрибутов вы узнаете в разделе 20.4, но это похоже на маршрутизацию на основе минимальных API, с которой вы уже знакомы.

В листинге 20.2 данные возвращаются непосредственно из метода действия, но вам не нужно этого делать. Вместо этого можно вернуть объект `IActionResult`, и часто это и требуется. В зависимости от желаемого поведения API иногда вам может понадобиться вернуть данные, а в других случаях нужно будет вернуть низкоуровневый код состояния HTTP, указывающий, был ли запрос успешным. Например, если выполняется вызов API с запросом сведений о продукте, которого не существует, вы можете вернуть код состояния 404 `Not Found`.

ПРИМЕЧАНИЕ Это похоже на паттерны, которые мы использовали в минимальных API. Но помните, что минимальные API используют `IResult`, а контроллеры веб-API, контроллеры MVC и Razor Pages используют `IActionResult`.

В листинге 20.3 показан пример того, как нужно вернуть объект `IActionResult`. Здесь видно еще одно действие с тем же классом `FruitController`, что и раньше. Данный метод предоставляет клиентам возможность получить конкретный фрукт по идентификатору, который, как мы предполагаем, в нашем примере является индексом в списке `_fruit`, который мы определили в предыдущем листинге. Привязка модели используется, чтобы задать значение параметра `id` из запроса.

ПРИМЕЧАНИЕ Контроллеры API используют ту же инфраструктуру привязки модели, что и Razor Pages для привязки параметров метода действия к входящему запросу. Привязка модели и валидация работают так же, как было показано в главе 16: вы можете привязать запрос к простым примитивам, а также к сложным объектам C#. Единственное отличие состоит в том, что здесь нет `PageModel` со свойствами `[BindProperty]`; можно выполнить привязку только к параметрам метода действия.

Листинг 20.3 Действие веб-API, возвращающее IActionResult для обработки условий ошибки

```
[HttpGet("fruit/{id}")] ← Определяет шаблон маршрута
public ActionResult<string> View(int id) ← для метода действия
{
    if (id >= 0 && id < _fruit.Count) ← Метод действия возвращает
    { ← ActionResult<string>, поэтому
        return _fruit[id]; ← он может возвращать строку
    } ← или объект IActionResult
    } ← Элемент может быть возвращен
    return NotFound(); ← только в том случае, если зна-
} ← чение id является допустимым
    В случае успеха данные будут возвращены с кодом состояния 200
    Вызов метода NotFound возвращает объект NotFoundResult, который отправит код состояния 404
    
```

В успешном сценарии для метода параметр `id` будет иметь значение больше 0 и меньшее, чем количество элементов в списке `_fruit`. Если это так, то элемент возвращается вызывающему компоненту. Как и в листинге 20.2, это достигается простым возвратом данных, что генерирует код состояния 200 и возвращает элемент в теле ответа, как показано на рис. 20.3. Также можно было бы обернуть данные с помощью объекта `OkResult`, вызвав `Ok(_fruit [id])`, используя вспомогательный метод `Ok` класса `ControllerBase` – «под капотом» результат идентичен.

ПРИМЕЧАНИЕ Некоторым становится не по себе, когда они видят фразу «вспомогательный метод», но во вспомогательных методах класса `ControllerBase` нет ничего волшебного – это сокращение для создания нового объекта `IActionResult` заданного типа. Впрочем, не нужно верить мне на слово. Вы всегда можете просмотреть исходный код базового класса на GitHub на странице <http://mng.bz/5wQB>.

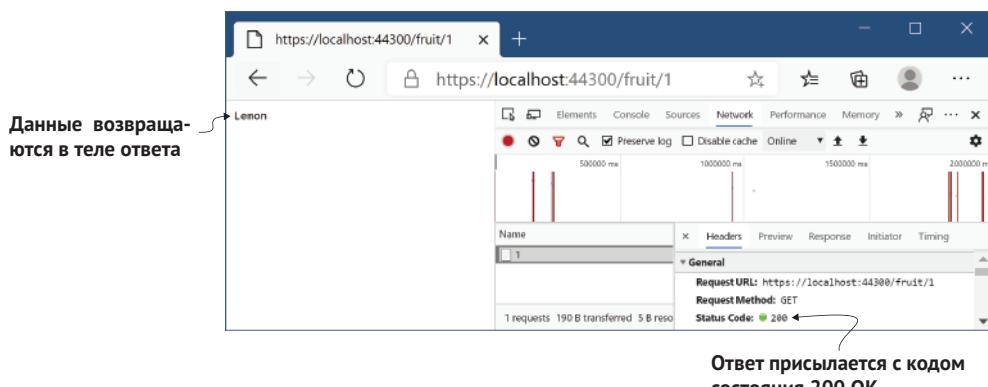


Рис. 20.3 Данные, возвращаемые из метода действия, сериализуются в тело ответа, и он генерирует ответ с кодом состояния 200 OK

Если параметр `id` находится за границами списка `_fruit`, вызывается метод `NotFound` для создания объекта `NotFoundResult`. При выполнении этот метод генерирует код состояния `404 Not Found`. Атрибут `[ApiController]` автоматически преобразует ответ в стандартный экземпляр `ProblemDetails`, как показано на рис. 20.4.

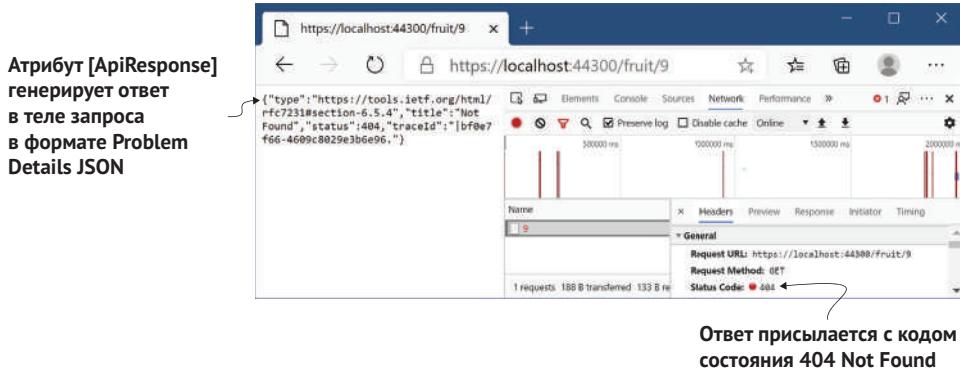


Рис. 20.4 Атрибут `[ApiController]` преобразует ответы об ошибках (в данном случае ответ 404) в стандартный формат `ProblemDetails`

Один аспект из листинга 20.3, который может вас сбить с толку, заключается в том, что в случае успеха мы возвращаем экземпляр строки, но в сигнатуре метода `View` говорится, что мы возвращаем тип `ActionResult<string>`. Как такое возможно? Почему это не ошибка компилятора?

В обобщенном типе `ActionResult<T>` для этого используются некоторые особенности языка C# для работы с неявными преобразованиями. Использование типа `ActionResult<T>` имеет два преимущества:

- вы можете вернуть либо экземпляр `T`, либо реализацию объекта `ActionResult`, например `NotFoundResult`, из того же метода. Это может быть удобно, как в листинге 20.3;
- это обеспечивает лучшую интеграцию с поддержкой OpenAPI в ASP.NET Core.

Вы можете возвращать любой тип `ActionResult` из контроллеров веб-API, но обычно это будут экземпляры `StatusCodeResult`, которые задают ответ на определенный код состояния, с ассоциированными данными или без них. Например, `NotFoundResult` и `OkResult` являются производными от `StatusCodeResult`. Еще один часто используемый код состояния – это `400 Bad Request`, который обычно возвращается, когда данные, предоставленные в запросе, не проходят валидацию. Его можно сгенерировать с помощью `BadRequestResult`. Во многих случаях атрибут `[ApiController]` может автоматически сгенерировать для вас ответы с кодом `400`, как вы увидите в разделе 20.5.

СОВЕТ В главе 15 вы познакомились с различными объектами `ActionResult`. `BadRequestResult`, `OkResult` и `NotFoundResult` – все они являются производными от `StatusCodeResult` и задают соответствующий код состояния для своего типа (`400`, `202` и `404` соответственно). Ис-

пользование этих классов-оболочек делает намерения вашего кода более ясными, вместо того чтобы полагаться на то, что другие разработчики поймут значение различных номеров кодов состояния.

Как только вы вернули `ActionResult` (или другой объект) из своего контроллера, он сериализуется в соответствующий ответ. Это работает несколькими способами, в зависимости от:

- форматеров, которые поддерживает ваше приложение;
- данных, которые вы возвращаете из своего метода;
- форматов данных, которые запрашивающий клиент может обрабатывать.

О форматерах и сериализации данных вы подробнее узнаете в разделе 20.6, но, прежде чем мы продолжим, стоит изучить параллели между традиционными приложениями с отрисовкой на стороне сервера и конечными точками веб-API. Эти два понятия похожи, поэтому важно установить общие закономерности и различия.

20.2 Применение паттерна проектирования MVC к веб-API

В ASP.NET Core одна и та же базовая платформа используется в сочетании с контроллерами веб-API, страницами Razor и контроллерами MVC с представлениями. Вы уже видели это сами; веб-API `FruitController`, созданный вами в разделе 20.2, похож на контроллеры MVC, которые вы видели в главе 19.

Следовательно, даже если вы создаете приложение, полностью состоящее из веб-API, без использования отрисовки HTML-кода на стороне сервера, паттерн проектирования MVC все равно применим. Независимо от того, создаете ли вы традиционные веб-приложения или веб-API, вы можете структурировать свое приложение практически одинаково.

Надеюсь, теперь вы хорошо знакомы с тем, как ASP.NET Core обрабатывает запросы. Но на всякий случай на рис. 20.5 показано, как фреймворк обрабатывает типичный запрос к приложению Razor Pages после прохождения через конвейер промежуточного ПО. В этом примере показано, как может выглядеть запрос на просмотр доступных фруктов на традиционном веб-сайте продуктового магазина.

`RoutingMiddleware` направляет запрос на просмотр всех фруктов, перечисленных в категории `apples`, к странице Razor, `Fruit.cshtml`. Затем `EndpointMiddleware` создает модель привязки, проверяет ее, задает ее в качестве свойства `PageModel` и задает свойство `ModelState` в базовом классе `PageModel` с подробной информацией обо всех ошибках валидации. Обработчик страницы взаимодействует с моделью приложения, вызывая сервисы, обращаясь к базе данных и получая все необходимые данные.

Наконец, страница Razor выполняет свое представление Razor, используя `PageModel` для генерации ответа в виде HTML. Ответ возвращается через конвейер промежуточного ПО и отправляется в браузер пользователя.



Рис. 20.5 Обработка запроса к традиционному приложению Razor Pages, в котором представление генерирует HTML-ответ, отправляемый обратно пользователю. Эта диаграмма теперь должна быть вам хорошо знакома!

Как бы это изменилось, если бы запрос пришел из клиентского или мобильного приложения? Если вы хотите использовать машиночитаемый формат JSON вместо HTML, насколько велика разница? Как показано на рис. 20.6, ответ – «она совсем небольшая». Основные изменения связаны с переключением с Razor Pages на контроллеры и действия, но, как вы видели в главе 19, оба подхода используют одни и те же общие парадигмы.

Заштрихованная часть диаграммы идентична той, что изображена на рис. 20.5

Как и раньше, компонент маршрутизации выбирает конечную точку для вызова на основе входящего URL-адреса. Для контроллеров API это контроллер и действие, а не страница Razor.

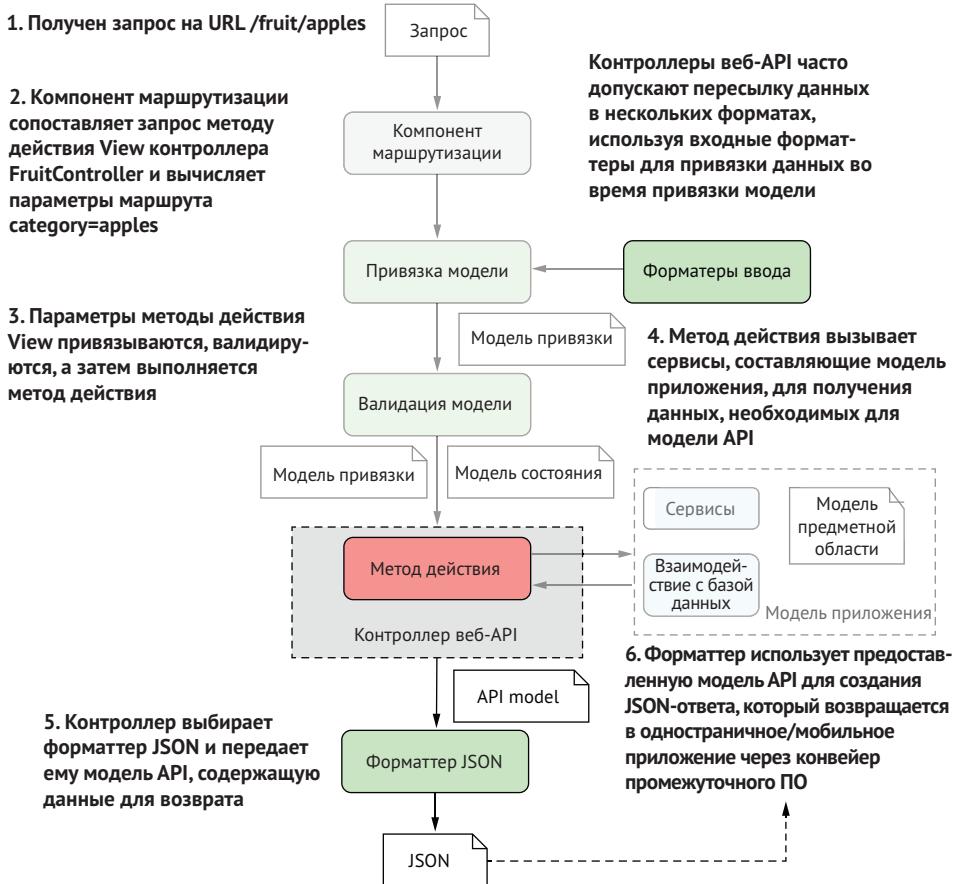


Рис. 20.6 Вызов конечной точки веб-API в веб-приложении для онлайн-торговли

После маршрутизации идет привязка модели, в которой связыватель создает модель привязки и заполняет ее значениями из запроса. Веб-API часто принимают данные в большем количестве форматов, чем Razor Pages, например XML, но в остальном процесс привязки модели такой же, как и для запроса Razor Pages. Валидация происходит таким же образом, и свойство `ModelState` в базовом классе `ControllerBase` заполняется ошибками валидации.

ПРИМЕЧАНИЕ Веб-API используют форматтеры ввода, чтобы принимать данные, отправляемые им в различных форматах. Обычно это форматы JSON или XML, но можно создавать форматтеры ввода для любого типа, например CSV. Я показываю, как активировать форматтер ввода XML, в разделе 20.6. Как создать собственный форматтер ввода, можно увидеть на странице <http://mng.bz/e5gG>.

Метод действия эквивалентен обработчику Razor Page; он точно так же взаимодействует с моделью приложения. Это важный момент; пу-

тем отделения поведения вашего приложения в модель приложения, вместо того чтобы включать его в свои представления и сами контроллеры, можно повторно использовать бизнес-логику своего приложения с несколькими парадигмами пользовательского интерфейса.

СОВЕТ По возможности делайте обработчики страниц и контроллеры максимально простыми. Перенесите все решения бизнес-логики в сервисы, которые улучшат модель вашего приложения, и следите за тем, чтобы ваши страницы Razor и контроллеры API были сосредоточены на механике взаимодействия с пользователем.

После того как модель приложения вернет данные, необходимые для обслуживания запроса – объекты фруктов в категории `apples`, – вы увидите первое существенное различие между контроллерами API и Razor Pages. Вместо того чтобы добавлять значения в `PageModel` для использования в представлении Razor, метод действия создает *модель API*. Это аналог `PageModel`, но в ней нет данных, используемых для генерации представления HTML. Она содержит данные, которые будут отправлены обратно в ответе.

ОПРЕДЕЛЕНИЕ *Модели представления* и `PageModel` содержат *данные*, необходимые для построения ответа, и *метаданные* о том, как это сделать. Обычно модели API содержат только те данные, которые должны быть возвращены в ответе.

Когда мы изучали приложение Razor Pages, то использовали `PageModel` в сочетании с шаблоном представления Razor для создания окончательного ответа. В приложении Web API мы используем API-модель в сочетании с *форматером вывода*. Форматер вывода, как следует из названия, сериализует модель API в машиночитаемый ответ, такой как JSON или XML. Он формирует букву «V» в веб-API-версии MVC, выбирая соответствующее представление возвращаемых данных.

Наконец, что касается приложения Razor Pages, сгенерированный ответ затем отправляется обратно через конвейер промежуточного ПО, проходя через каждый из сконфигурированных компонентов, и возвращается обратно к исходному вызывающему компоненту.

Надеюсь, параллели между Razor Pages и веб-API очевидны; большая часть поведения идентична – меняется только ответ. Все, от момента поступления запроса до взаимодействия с моделью приложения, очень похоже в обеих парадигмах.

Большинство различий между Razor Pages и веб-API связаны не с тем, как фреймворк работает под капотом, а с тем, как используются разные парадигмы. Например, в следующем разделе вы узнаете, как конструкции маршрутизации, которые мы рассматривали в главах 6 и 15, используются с веб-API с помощью маршрутизации на основе атрибутов.

20.3 Маршрутизация на основе атрибутов: связывание методов действий с URL-адресами

В этом разделе вы узнаете о маршрутизации на основе атрибутов: механизме связывания действий контроллера API с заданным шаблоном маршрута. Вы увидите, как связать действия контроллера с определенными HTTP-методами, такими как GET и POST, и как избежать дублирования в шаблонах.

Мы подробно рассматривали шаблоны маршрутов в главе 6 в контексте Razor Pages, и вам будет приятно узнать, что точно такие же шаблоны маршрутов используются и с контроллерами API. Единственная разница заключается в том, как мы *определяем* шаблоны: в Razor Pages для этого используется директива `@page`, в минимальных API – методы `MapGet()` или `MapPost()`, а в контроллерах API – атрибуты маршрутизации.

ПРИМЕЧАНИЕ Все три парадигмы используют явную маршрутизацию «под капотом». Альтернативный вариант, маршрутизация на основе соглашений, обычно используется с традиционными контроллерами и представлениями MVC, как описано в главе 19. Как я уже упоминал, я не рекомендую использовать этот подход в целом, поэтому в этой книге данный вид маршрутизации не рассматривается.

С помощью маршрутизации на основе атрибутов мы декорируем каждый метод действия в контроллере API атрибутом и предоставляем ассоциированный шаблон маршрута для метода действия, как показано в следующем листинге.

Листинг 20.4 Пример маршрутизации на основе атрибутов

```
public class HomeController: Controller
{
    [Route("")]
    public IActionResult Index()
    {
        /* Реализация метода */
    }
    [Route("contact")]
    public IActionResult Contact()
    {
        /* Реализация метода */
    }
}
```

Каждый атрибут `[Route]` определяет шаблон маршрута, который должен быть ассоциирован с методом действия. В приведенном примере URL-адрес `/` сопоставляется напрямую с методом `Index`, а URL-адрес `/contact` – с методом `Contact`.

Маршрутизация на основе атрибутов сопоставляет URL-адреса с определенным методом действия, но один метод действия по-прежнему может иметь несколько шаблонов маршрутов и, следовательно, может соответствовать нескольким URL-адресам. Каждый шаблон должен объявляться со своим собственным атрибутом `RouteAttribute`, как показано в этом листинге, представляющим собой каркас веб-API для гоночной игры.

Листинг 20.5 Маршрутизация на основе атрибутов с несколькими атрибутами

```
public class CarController
{
    [Route("car/start")]
    [Route("car/ignition")]
    [Route("start-car")]
    public IActionResult Start() <-- Метод Start будет выполнен
    {                                         при совпадении любого из
        /* Реализация метода */             этих шаблонов маршрута
    }

    [Route("car/speed/{speed}")]
    [Route("set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed) <-- Имя метода действия не
    {                                         влияет на шаблон маршрута
        /* Реализация метода */
    }
}
```

[Route("car/speed/{speed}")] | Шаблон `RouteAttribute` может содержать параметры маршрута, в данном случае `{speed}`

В этом листинге показаны два разных метода действий, к каждому из которых можно получить доступ из нескольких URL-адресов. Например, метод `Start` будет выполнен при запросе любого из следующих URL-адресов:

- `/car/start;`
- `/car/ignition;`
- `/start-car.`

Эти адреса полностью независимы от имен контроллера и методов действия; важно только значение `RouteAttribute`.

ПРИМЕЧАНИЕ По умолчанию при использовании `RouteAttribute` имена контроллера и действия не влияют на URL-адреса или шаблоны маршрутов.

Шаблоны, используемые в атрибутах маршрута, являются стандартными шаблонами маршрута. Это те же шаблоны, которые вы использовали в главе 6. Можно применить литеральные сегменты и определить параметры маршрутов, которые будут извлекать значения из URL-адреса, как показано в предыдущем листинге с участием метода `SetCarSpeed`. Данный метод определяет два шаблона маршрута, каждый из которых определяет параметр маршрута, `{speed}`.

В этом примере я использовал несколько атрибутов [Route] для каждого действия, но лучше всего декорировать действие одним URL-адресом. Так ваш API будет проще понять, и другим приложениям будет легче потреблять его.

Как и во всех частях ASP.NET Core, параметры маршрута представляют собой сегмент URL-адреса, который может меняться. Как и в случае с минимальными API и страницами Razor, параметры маршрута в шаблонах `RouteAttribute` могут:

- быть необязательными;
- иметь значения по умолчанию;
- использовать ограничения маршрута.

Например, можно было бы обновить метод `SetCarSpeed` из предыдущего листинга, чтобы ограничить параметр маршрута `{speed}` целым числом и значением по умолчанию, равным 20:

```
[Route("car/speed/{speed=20:int}")]
[Route("set-speed/{speed=20:int}")]
public IActionResult SetCarSpeed(int speed)
```

ПРИМЕЧАНИЕ Как мы уже обсуждали в главе 6, не стоит использовать ограничения маршрута для валидации. Например, если вы вызовете предыдущий маршрут "set-speed /{speed=20:int}" с недопустимым значением скорости, /set-speed/oops, то получите ответ 404 Not Found, поскольку маршрут не совпадает. Без ограничения `int` вы получите более разумный ответ 400 Bad Request.

Если вам удалось разобраться с маршрутизацией в главе 6, то маршрутизация с помощью контроллеров API не должна преподнести вам никаких сюрпризов. Одна вещь, которую вы, возможно, станете замечать, когда начнете использовать маршрутизацию на основе атрибутов с контроллерами API, – это количество повторов. Минимальные API используют группы маршрутов для уменьшения дублирования, а Razor Pages устраниет множество повторов, используя условные обозначения для вычисления шаблонов маршрутов на основе имени файла Razor Page. Так что же можно использовать с контроллерами веб-API?

20.3.1 Сочетание атрибутов маршрута для следования принципу DRY

Добавление атрибутов маршрута ко всем своим контроллерам API может быть немного утомительным занятием, особенно если вы в основном следите соглашениям, в которых ваши маршруты имеют стандартный префикс, например "эри" или имя контроллера. Как правило, нужно убедиться, что вы следите принципу DRY (don't repeat yourself), когда речь идет об этих строках. В следующем листинге показаны два метода действия с несколькими атрибутами [Route]. (Этот листинг приводится здесь только в показательных целях. По возможности придерживайтесь одного атрибута на каждое действие!)

Листинг 20.6 Дублирование в шаблонах RouteAttribute

```
public class CarController
{
    [Route("api/car/start")]
    [Route("api/car/ignition")]
    [Route("start-car")]
    public IActionResult Start()
    {
        /* Реализация метода */
    }

    [Route("api/car/speed/{speed}")]
    [Route("set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed)
    {
        /* Реализация метода */
    }
}
```

В нескольких шаблонах маршрутов используется один и тот же префикс api/car

Здесь довольно много дублирования – вы добавляете "api/car" к большинству своих маршрутов. Предположительно, если бы вы решили написать вместо этого "api/vehicles", вам пришлось бы просмотреть каждый атрибут и обновить его. Такой код так и напрашивается на опечатку!

Чтобы облегчить ситуацию, можно применить атрибуты `RouteAttribute` к контроллерам в дополнение к методам действий. Когда и у контроллера, и у метода действия есть атрибут маршрута, общий шаблон маршрута для метода рассчитывается путем объединения двух шаблонов.

Листинг 20.7 Объединение шаблонов RouteAttribute

```
[Route("api/car")]
public class CarController
{
    [Route("start")]
    [Route("ignition")]
    [Route("/start-car")]
    public IActionResult Start()
    {
        /* Реализация метода */
    }

    [Route("speed/{speed}")]
    [Route("/set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed)
    {
        /* Реализация метода */
    }
}
```

Объединяется для получения шаблона «api/car/start»

Не объединяется, потому что начинается с / и дает шаблон «start-car»

Объединяется для получения шаблона «api/car/speed/{speed}»

Не объединяется, потому что начинается с / и дает шаблон «/set-speed/{speed}»

Данное сочетание атрибутов может уменьшить дублирование в шаблонах маршрутов и упростить добавление или изменение префиксов (например, поменять "car" на "vehicle") для нескольких методов действия.

Чтобы игнорировать атрибут `RouteAttribute` контроллера и создать абсолютный шаблон маршрута, начните шаблон маршрута метода действия с косой черты (/). Применение `RouteAttribute` контроллера уменьшает дублирование, но есть способ лучше – использовать замену маркера.

20.3.2 Использование замены маркера для уменьшения дублирования при маршрутизации на основе атрибутов

Возможность сочетать маршруты атрибутов удобна, но дублирование все равно остается, если вы добавляете к маршрутам префикс с именем контроллера или если в шаблонах маршрутов всегда используется имя действия. Если хотите, можно сделать еще проще!

Маршруты атрибутов поддерживают автоматическую замену маркеров `[action]` и `[controller]` в маршрутах атрибутов. Они будут заменены именами действия и контроллера (без суффикса «Controller») соответственно. Маркеры заменяются после объединения всех атрибутов, поэтому это полезно, если у вас есть иерархия наследования контроллеров.

В следующем листинге показано, как создать класс `BaseController`, который можно использовать, чтобы применить согласованный префикс шаблона маршрута ко *всем* контроллерам API в приложении.

Листинг 20.8 Замена маркера в `RouteAttributes`

```
[Route("api/[controller]")] ← Можно применять атрибуты к базовому классу, а производные классы будут наследовать их
public abstract class BaseController { } ←

public class CarController : BaseController ← Замена маркера происходит последней, поэтому [controller] заменяется на «car», а не «base»
{
    [Route("[action]")]
    [Route("ignition")]
    [Route("/start-car")]
    public IActionResult Start()
    {
        /* Реализация метода */
    }
}
```

← Объединяет и заменяет маркеры, чтобы получить шаблон «api/car/start»

← Объединяет и заменяет маркеры, чтобы получить шаблон «api/car/ignition»

→ Не объединяется с базовыми атрибутами, потому что начинается с /, и поэтому остается «start-car»

ВНИМАНИЕ! Если вы используете замену для маркеров `[action]` или `[controller]`, помните, что переименование классов и методов приведет к изменению открытого API. Если вас это беспокоит, можно вместо этого использовать статические строки, например "car".

В сочетании с тем, что вы узнали в главе 6, мы рассмотрели почти все, что нужно знать о маршрутизации на основе атрибутов. Есть еще одна вещь, на которую следует обратить внимание: обработка различных HTTP-методов, среди которых GET и POST.

20.3.3 Обработка HTTP-методов с помощью маршрутизации на основе атрибутов

В Razor Pages HTTP-методы, такие как GET или POST, не участвуют в процессе маршрутизации. Компонент `RoutingMiddleware` определяет, какую страницу Razor выполнять, исключительно на основе шаблона маршрута, ассоциированного со страницей Razor. HTTP-метод используется только перед выполнением страницы Razor, чтобы решить, какой обработчик страницы выполнить: например, `OnGet` для метода GET или `OnPost` для метода POST.

Контроллеры веб-API работают как конечные точки минимальных API: HTTP-метод сам участвует в процессе маршрутизации. Таким образом, запрос с методом GET можно перенаправить на одно действие, а запрос с методом POST – на другое действие, даже если в запросе использовался тот же URL-адрес.

Атрибут `[Route]`, который мы использовали до сих пор, отвечает на все HTTP-методы. Вместо этого действие обычно должно обрабатывать только один метод. Вместо атрибута `[Route]` можно использовать:

- `[HttpPost]` для обработки запросов методом POST;
- `[HttpGet]` для обработки запросов методом GET;
- `[HttpPut]` для обработки запросов методом PUT.

Подобные атрибуты существуют для всех стандартных HTTP-методов, например `DELETE` и `OPTIONS`. Эти атрибуты можно использовать вместо атрибута `[Route]`, чтобы указать, что метод действия должен соответствовать одному HTTP-методу, как показано в следующем листинге.

Листинг 20.9 Использование атрибутов HTTP-метода с маршрутизацией на основе атрибутов

```
public class AppointmentController
{
    [HttpGet("/appointments")]
    public IActionResult ListAppointments()
    {
        /* Реализация метода */
    }
    [HttpPost("/appointments")]
    public IActionResult CreateAppointment()
    {
        /* Реализация метода */
    }
}
```

Выполняется только в ответ на GET-запрос к URL-адресу /appointments

Выполняется только в ответ на POST-запрос к URL-адресу /appointments

Если приложение получает запрос, совпадающий с шаблоном маршрута метода действия, но не совпадающий с нужным HTTP-методом, вы получите ошибку `405 Method not allowed`. Например, если вы отправите запрос с помощью метода `DELETE` на URL-адрес `/appointments` из предыдущего листинга, то получите ответ с ошибкой `405`.

Маршрутизация на основе атрибутов использовалась с контроллерами API с первых дней существования ASP.NET Core, поскольку она

позволяет строго контролировать URL-адреса, предоставляемые приложением. Когда вы будете создавать контроллеры API, вам придется постоянно писать какой-то повторяющийся код. Атрибут [ApiController] разработан для того, чтобы справиться с некоторыми из этих проблем и сократить объем шаблонного кода.

20.4 Использование общепринятых соглашений с атрибутом [ApiController]

В этом разделе вы узнаете об атрибуте [ApiController] и о том, как с его помощью можно уменьшить объем кода, который необходимо писать для создания согласованных контроллеров веб-API. Вы узнаете об используемых в нем соглашениях, о том, почему они полезны и как их отключить, если понадобится.

Атрибут [ApiController] появился в .NET Core 2.1, дабы упростить процесс создания контроллеров веб-API. Чтобы понять, что он делает, полезно взглянуть на пример того, как написать контроллер веб-API без атрибута [ApiController], и сравнить его с кодом, необходимым для того, чтобы добиться того же, но уже с атрибутом.

Листинг 20.10 Создание контроллера веб-API без атрибута [ApiController]

```
public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear", "Lemon", "Peach"
    };
    [HttpPost("fruit")]
    public ActionResult Update([FromBody] UpdateModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(
                new ValidationProblemDetails(ModelState));
        }

        if (model.Id < 0 || model.Id > _fruit.Count)
        {
            return NotFound(new ProblemDetails()
            {
                Status = 404,
                Title = "Not Found",
                Type = "https://tools.ietf.org/html/rfc7231"
                    + "#section-6.5.4",
            });
        }

        _fruit[model.Id] = model.Name;
        return Ok();
    }
}
```

Нужно проверить, была ли валидация модели успешной, и вернуть ответ с кодом состояния 400, если она не удалась

В этом примере список строк служит моделью бизнес-правил приложения

Бесп-АПИ используют маршрутизацию на основе атрибутов для определения шаблонов маршрутов

Атрибут [FromBody] указывает на то, что параметр должен быть привязан к телу запроса

Если отправленные данные не содержат допустимого идентификатора, вернуть ответ 404 ProblemDetails

Обновляем модель и возвращаем ответ с кодом состояния 200

```
public class UpdateModel
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }
}
```

UpdateModel действителен только в том случае, если указано значение **Name**, как установлено атрибутом **[Required]**

В этом примере демонстрируется множество распространенных функций и паттернов, используемых с контроллерами веб-API:

- контроллеры веб-API считывают данные из тела запроса, обычно отправляемого в формате JSON. Чтобы тело читалось как JSON, а не как значения формы, необходимо применить атрибут **[FromBody]** к параметрам метода, дабы убедиться, что оно правильно привязано к модели;
- как обсуждалось в главе 16, после привязки выполняется валидация модели, но нужно действовать в соответствии с результатами валидации. Вы должны вернуть ответ **400 Bad Request**, если предоставленные значения не прошли валидацию. Обычно требуется предоставить подробную информацию о том, почему запрос был недействительным: это делается в листинге 20.10 путем возврата объекта **ValidationProblemDetails** в теле ответа, созданного из **ModelState**;
- каждый раз, когда вы возвращаете статус ошибки, например **404 Not Found**, по возможности нужно возвращать сведения о проблеме, которые позволяют вызывающему компоненту диагностировать проблему. В ASP.NET Core это рекомендуется делать с помощью класса **ProblemDetails**.

Код из листинга 20.10 представляет то, что вы могли видеть в контроллере API ASP.NET Core до .NET Core 2.1. Появление атрибута **[ApiController]** в .NET Core 2.1 (и его последующие улучшения в более поздних версиях) делает тот же самый код намного проще, как показано в следующем листинге.

Листинг 20.11 Создание контроллера веб-API с атрибутом [ApiController]

```
[ApiController]
public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear", "Lemon", "Peach"
    };

    [HttpPost("fruit")]
    public ActionResult Update(UpdateModel model)
```

При добавлении атрибута **[ApiController]** применяется несколько соглашений, общих для контроллеров API

Атрибут **[FromBody]** предполагается для сложных параметров метода действия

```

{
    if (model.Id < 0 || model.Id > _fruit.Count)
    {
        return NotFound(); ← Модель проверяется автоматически, и если она невалидная, то возвращается ответ с кодом состояния 400
    }

    _fruit[model.Id] = model.Name;

    return Ok();
}

public class UpdateModel
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }
}
}

```

Коды состояния ошибки автоматически преобразуются в объект **ProblemDetails**

Если вы сравните листинги 20.10 и 20.11, то увидите, что весь выделенный жирным шрифтом код из листинга 20.10 можно удалить и заменить атрибутом `[ApiController]` из листинга 20.11. Атрибут `[ApiController]` автоматически применяет к контроллерам несколько соглашений:

- *маршрутизация на основе атрибутов* – вы должны использовать маршрутизацию на основе атрибутов со своими контроллерами; нельзя использовать маршрутизацию на основе соглашений. Скорее всего, вы и не будете, так как мы обсуждали именно подход на основе атрибутов для контроллеров веб-API;
- *автоматические ответы с кодом 400* – в главе 16 я говорил, что всегда следует проверять значение `ModelState.IsValid` в обработчиках страниц Razor и действиях MVC, но атрибут `[ApiController]` делает это за вас, добавляя фильтр, как мы это делали с минимальными API в главе 7. Мы подробно рассмотрим фильтры MVC в главах 21 и 22;
- *вывод источника привязки модели* – если атрибута `[ApiController]` нет, то предполагается, что сложные типы передаются в виде значений формы в теле запроса. В случае с веб-API данные гораздо чаще передаются в формате JSON, что обычно требует добавления атрибута `[FromBody]`. Атрибут `[ApiController]` позаботится об этом за вас;
- *ProblemDetails для кодов ошибок* – часто нужно вернуть согласованный набор данных при возникновении ошибки в API. `ProblemDetails` – это тип, основанный на веб-стандарте, который служит этими согласованными данными. Атрибут `[ApiController]` будет перехватывать любые коды состояния ошибки, возвращаемые вашим контроллером (например, `404 Not Found`), и автоматически преобразовывать их в тип `ProblemDetails`.

Ключевой особенностью атрибута `[ApiController]`, когда он появился, была поддержка сведений о проблеме, но, как я писал в главе 5, такое же автоматическое преобразование в сведения о проблеме теперь поддер-

живается компонентами `ExceptionHandlerMiddleware` и `StatusCodesPagesMiddleware` по умолчанию. Тем не менее соглашения `[ApiController]` могут значительно сократить объем шаблонного кода, который вам придется писать, и обеспечить, например, автоматическую обработку ошибок проверки.

Как это часто бывает в ASP.NET Core, ваша работа будет более продуктивной, если вы будете следовать соглашениям, а не пытаться с ними бороться. Однако если какие-то соглашения вам не нравятся или вы хотите их изменить, можно легко это сделать.

Можно настроить соглашения, используемые приложением, вызвав метод `ConfigureApiBehaviorOptions()` объекта `IMvcBuilder`, возвращаемого из метода `AddControllers()` в файле `Startup.cs`. Например, можно отключить автоматические ответы с кодом `400` при ошибке валидации, как показано в следующем листинге.

Листинг 20.12 Настройка поведения атрибута `[ApiController]`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
    .ConfigureApiBehaviorOptions(options => {
        {
            options.SuppressModelStateInvalidFilter = true;
        });
    // ...
}

Управляем применяемыми соглашениями с помощью лямбда-функции конфигурации
Отключит автоматические ответы с кодом состояния 400 на недопустимые запросы
```

СОВЕТ Можно отключить все автоматические функции, активированные атрибутом `[ApiController]`, но я рекомендую придерживаться значений по умолчанию, если вам действительно не нужно их менять. Подробнее об отключении функций можно прочитать на странице <https://docs.microsoft.com/aspnet/core/web-api>.

Возможность настройки каждого аспекта контроллеров веб-API является одним из ключевых отличий минимальных API. В следующем разделе вы узнаете, как управлять форматом данных, возвращаемых контроллерами веб-API, будь то JSON, XML или другой собственный формат.

20.5 Генерация ответа от модели

Мы подошли к последней теме этой главы: форматирование ответа. Сейчас контроллеры API часто возвращают данные в формате JSON, но это не всегда так. В данном разделе вы узнаете о согласовании содержимого и о том, как активировать дополнительные форматы вывода, например XML.

Рассмотрим следующий сценарий: вы создали метод действия веб-API для возврата списка автомобилей, как показано в следующем листинге. Он вызывает метод модели приложения, который возвращает список данных контроллеру. Теперь нужно отформатировать ответ и вернуть его вызывающему компоненту.

Листинг 20.13 Контроллер веб-API для возврата списка автомобилей

```
[ApiController]
public class CarsController : Controller
{
    [HttpGet("api/cars")]
    public IEnumerable<string> ListCars()
    {
        return new string[]
        {
            "Nissan Micra", "Ford Focus"
        };
    }
}
```

Эти данные обычно берутся из модели приложения

Действие выполняется с запросом к /api/cars

Модель API, содержащая данные, – это IEnumerable<string>

В разделе 20.2 вы видели, что можно возвращать данные непосредственно из метода действия, и в этом случае промежуточное ПО форматирует их и возвращает отформатированные данные вызывающему компоненту. Но как оно узнает, какой формат использовать? В конце концов, можно сериализовать их, используя JSON, XML или даже с помощью простого вызова метода `ToString()`.

ВНИМАНИЕ! Помните, что в этой главе я говорю только об ответах контроллера веб-API. Минимальные API поддерживают лишь автоматическую сериализацию в JSON и ничего больше.

Процесс определения формата данных для отправки клиентам обычно известен как *согласование содержимого*. В общих чертах это работает следующим образом: клиент отправляет заголовок, указывающий типы контента, которые он может понять, – заголовок `Accept`, – а сервер выбирает один из них, форматирует ответ и отправляет в ответе заголовок `Content-Type`, указывающий, какой тип был выбран.

Вы не обязаны отправлять только тот заголовок `Content-Type`, которого ожидает клиент, а в некоторых случаях вы даже не сможете обрабатывать типы, которые он запрашивает. Что, если в запросе указано, что он может принимать только таблицы Excel? Вряд ли вы это поддержите, даже если это единственный заголовок `Content-Type`, который содержится в запросе.

Когда вы возвращаете модель API из метода действия, независимо от того, делаете ли вы это напрямую (как в листинге 20.13) или через `OkResult` либо другой объект `StatusCodesResult`, ASP.NET Core всегда будет что-то возвращать. Если он не может удовлетворить какому-либо из типов, указанных в заголовке `Accept`, то он будет использовать формат JSON по умолчанию. На рис. 20.7 показано, что хотя был запрошен ответ в формате XML, контроллер API отформатировал ответ в формате JSON.

ВНИМАНИЕ! В предыдущей версии ASP.NET объекты сериализовались в JSON, используя стиль написания PascalCase, когда свойства начинаются с прописной буквы. В ASP.NET Core по умолчанию объекты сериализуются с использованием стиля camelCase, когда свойства начинаются со строчной буквы.

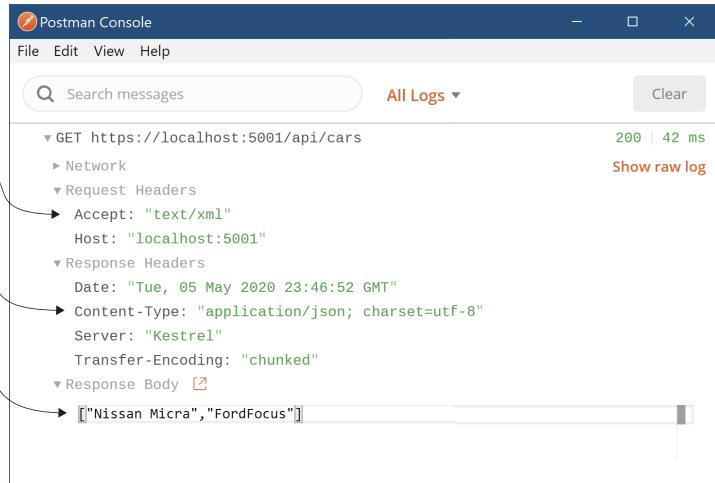


Рис. 20.7 Несмотря на то что запрос был сделан с заголовком Accept: "text/xml", ответ пришел в формате JSON, поскольку сервер не был настроен на возврат ответа в формате XML

Заголовки Accept и Content-Type

Заголовок Accept отправляется клиентом как часть запроса, чтобы указать тип содержимого, который может обработать клиент. Он состоит из нескольких MIME-типов, с дополнительными весовыми коэффициентами (от 0 до 1), чтобы указать, какой тип будет предпочтительнее. Например, заголовок application/json, text/xml;q=0.9, text/plain;q=0.6 указывает на то, что клиент может принимать форматы JSON, XML и простой текст с весовыми коэффициентами 1.0, 0.9 и 0.6 соответственно. JSON имеет весовой коэффициент 1.0, поскольку явного коэффициента предоставлено не было. Весовые коэффициенты можно использовать во время согласования содержимого для выбора оптимального представления для обеих сторон.

Заголовок Content-Type описывает данные, отправленные в запросе или ответе. Он содержит MIME-тип данных с необязательной кодировкой символов. Например, заголовок application/json; charset=utf-8 будет указывать на то, что тело запроса или ответ – это формат JSON, закодированный с использованием UTF-8.

Дополнительную информацию о MIME-типа см. в документации Mozilla на странице <http://mng.bz/gop8>. Документ RFC по согласованию содержимого можно найти на странице <http://mng.bz/6DXo>.

Как бы ни отправлялись данные, они сериализуются с помощью реализации IOutputFormatter. В комплект ASP.NET Core входит ограниченное количество форматеров «из коробки», но, как и всегда, можно легко добавить дополнительные форматеры или изменить их способ работы по умолчанию.

20.5.1 Настройка форматеров по умолчанию: добавляем поддержку XML

Форматеры веб-API в ASP.NET Core являются полностью настраиваемыми. По умолчанию настроены только форматеры для обычного текста (`text/plain`), HTML (`text/html`) и JSON (`application/json`). Учитывая распространенный вариант использования одностороничных и мобильных приложений, в большинстве случаев этого будет достаточно. Но иногда нужно иметь возможность возвращать данные в другом формате, например XML.

Newtonsoft.Json и System.Text.Json

Пакет `Newtonsoft.Json`, также известный как `Json.NET`, долгое время был каноническим способом работы с JSON в .NET. Он совместим со всеми существующими версиями .NET и, несомненно, знаком практически всем разработчикам .NET. Его охват был настолько велик, что даже ASP.NET Core стал зависеть от него!

Все изменилось с появлением в ASP.NET Core 3.0 новой библиотеки `System.Text.Json`, ориентированной на производительность. Начиная с версии 3.0 ASP.NET Core по умолчанию использует `System.Text.Json` вместо `Newtonsoft.Json`.

Основное различие между этими библиотеками заключается в том, что `System.Text.Json` очень требовательна к JSON. Обычно она десериализует только те данные JSON, которые соответствуют ее ожиданиям. Например, `System.Text.Json` не выполняет десериализацию JSON, в котором строки заключаются в одинарные кавычки – нужно использовать двойные.

Если вы создаете новое приложение, то обычно это не проблема – вы быстро научитесь генерировать правильный формат JSON. Но если вы переносите приложение из ASP.NET Core или получаете данные в формате JSON от стороннего приложения, то эти ограничения могут стать серьезным препятствием.

К счастью, можно легко вернуться к библиотеке `Newtonsoft.Json`. Установите пакет `Microsoft.AspNetCore.Mvc.NewtonsoftJson` в свой проект и обновите метод `AddControllers()` из файла `Program.cs`:

```
builder.Services.AddControllers()
    .AddNewtonsoftJson();
```

Так вы переключите форматеры ASP.NET Core на использование `Newtonsoft.Json` вместо `System.Text.Json`. Дополнительные сведения о различиях между библиотеками см. в статье на странице <http://mng.bz/OmRJ>. Дополнительные советы о том, когда следует переключаться на форматер `Newtonsoft.Json`, см. в разделе «Добавление поддержки формата JSON на основе `Newtonsoft.Json`» на странице <http://mng.bz/zx11>.

Можно добавить вывод XML в приложение, добавив форматер вывода. Форматеры приложения конфигурируются в файле `Program.cs` путем настройки объекта `IMvcBuilder`, возвращаемого из метода `AddControllers()`. Чтобы добавить форматер вывода XML, используйте следующий код:

```
services.AddControllers()
    .AddXmlSerializerFormatters();
```

ПРИМЕЧАНИЕ Технически таким образом вы также добавляете и форматер ввода XML, а это означает, что ваше приложение теперь может также получать формат XML в запросах. Раньше отправка запроса с XML в теле приводила к ответу 415 Unsupported Media Type. Подробное описание форматеров, включая создание собственного форматера, см. в документации на странице <http://mng.bz/e5gG>.

Благодаря этому простому изменению ваши контроллеры API теперь могут форматировать ответы как XML. Выполнение того же запроса, что и на рис. 20.7, с включенной поддержкой XML означает, что приложение будет учитывать значение `text/xml` для принимаемого заголовка `Accept`. Форматер сериализует строковый массив в XML по запросу, а не в JSON по умолчанию, как показано на рис. 20.8.



Рис. 20.8 После добавления форматера вывода XML значение `text/xml` заголовка `Accept` учитывается, и ответ может быть сериализован в XML

Это пример согласования содержимого, когда клиент указал, какие форматы он может обрабатывать, а сервер выбирает один из них в зависимости от того, что он может создать. Данный подход является частью протокола HTTP, но есть некоторые особенности, о которых следует знать, когда вы используете его в ASP.NET Core. Вы нечасто будете встречаться с ними, но знать о них нужно обязательно, чтобы не попасть впросак!

20.5.2 Выбор формата ответа с помощью согласования типа содержимого

Согласование типа содержимого – это когда клиент сообщает, какие типы данных он может принимать, используя заголовок `Accept`, а сервер выбирает наиболее подходящий из них, который он может об-

работать. В целом это работает так, как и следовало ожидать: сервер форматирует данные, используя тип, понятный клиенту.

В реализации ASP.NET Core есть несколько особых случаев, которые стоит учитывать:

- по умолчанию ASP.NET Core возвращает только MIME-типы `application/json`, `text/plain` и `text/html`. Можно добавить дополнительные форматеры `IOutputFormatter`, чтобы сделать доступными и другие типы, как вы видели в предыдущем разделе в случае с `text/xml`;
- по умолчанию если вы возвращаете `null` в качестве модели API, будь то из метода действия или путем передачи `null` в `StatusCodeResult`, то промежуточное ПО вернет ответ `204 No Content`;
- когда вы возвращаете строку в качестве модели API, если не задан заголовок `Accept`, промежуточное ПО отформатирует ответ как `text/plain`;
- когда вы используете любой другой класс в качестве модели API и при этом отсутствует заголовок `Accept`, либо не запрашивается ни один из поддерживаемых форматов, будет использоваться первый форматер, который может генерировать ответ (по умолчанию JSON);
- если промежуточное ПО обнаружит, что запрос, вероятно, исходит от браузера (заголовок `Accept` содержит `*/*`), то он *не* будет использовать согласование содержимого. Вместо этого он отформатирует ответ, как если бы заголовок `Accept` не был предоставлен, используя форматер по умолчанию.

Эти настройки по умолчанию относительно разумны, но они, безусловно, могут доставить вам немало неприятностей, если вы о них не знаете. В частности, последний пункт, когда ответ на запрос из браузера практически всегда форматируется как JSON, определенно подловил меня при попытке протестировать XML-запросы локально!

Как и следовало ожидать, все эти правила можно настроить; вы можете легко изменить поведение по умолчанию в приложении, если оно не соответствует вашим требованиям. Например, в следующем листинге показано, как заставить промежуточное ПО учитывать заголовок браузера `Accept` и удалить форматер `text/plain` для строк.

Листинг 20.14 Настройка MVC с учетом заголовка `Accept` в веб-API

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options => {
    options.RespectBrowserAcceptHeader = true;
    options.OutputFormatters.RemoveType<StringOutputFormatter>();
```

По умолчанию имеет значение `false`, но можно задать и ряд других свойств

У `AddControllers` есть перегруженный вариант, который принимает лямбда-функцию

Удаляет форматер вывода, который форматирует строки как `text/plain`

В большинстве случаев согласование содержимого должно работать «из коробки», независимо от того, создаете вы одностраничное

приложение или мобильное. В некоторых случаях вы можете обнаружить, что нужно обойти обычные механизмы согласования содержимого для определенных методов действий, и существует ряд способов сделать это, но я не буду описывать их в этой книге, так как обнаружил, что мне редко приходится использовать их. Подробности см. в документации Microsoft «Форматирование данных ответа в веб-API ASP.NET Core»: <http://mng.bz/zx11>.

На данном этапе мы изучили основные моменты использования контроллеров API, но у вас, вероятно, все еще остается один главный вопрос: зачем использовать контроллеры веб-API вместо минимальных API? Это отличный вопрос, и его мы рассмотрим в разделе 20.6.

20.6 Контроллеры веб-API и минимальные API: что выбрать?

В первой части книги вы узнали все об использовании минимальных API для создания API JSON. Минимальные API – это новинка, представленная в .NET 6, но они быстро растут. Со всеми нововведениями, представленными в .NET 7 (обсуждаемыми в главе 5), минимальные API становятся отличным способом создания HTTP API в современной платформе .NET.

Напротив, контроллеры веб-API существуют с самого первого дня. В своей текущей форме они были представлены в ASP.NET Core 1.0, и их создатели во многом вдохновлялись фреймворком веб-API из устаревшей версии ASP.NET. Варианты проектирования, паттерны и концепции, используемые контроллерами веб-API, с тех пор не сильно изменились, поэтому если вы когда-либо использовали контроллеры веб-API, они должны выглядеть знакомыми в .NET 7.

Сложный вопрос, касающийся .NET 7: если вам нужно создать API, что следует использовать: минимальные API или контроллеры веб-API? У всех них есть свои плюсы и минусы, и в значительной степени решение будет зависеть от личных предпочтений, но чтобы принять его, следует задать себе несколько вопросов:

- 1 Вам нужно возвращать данные в нескольких форматах с помощью согласования содержимого?
- 2 Критична ли производительность для вашего приложения?
- 3 У вас есть сложные требования к фильтрации?
- 4 Это новый проект?
- 5 Есть ли у вас уже опыт работы с контроллерами веб-API?
- 6 Вы предпочитаете соглашения, а не конфигурацию?

Первые три вопроса из этого списка посвящены техническим различиям между минимальными API и контроллерами веб-API. Контроллеры веб-API поддерживают согласование содержимого, которое позволяет клиентам запрашивать возврат данных в определенном формате: например, JSON, XML или CSV, как вы узнали в разделе 20.5. Контроллеры веб-API поддерживают эту функцию «из коробки», поэтому, если это важно для приложения, возможно, лучше выбрать контроллеры веб-API, а не минимальные API.

СОВЕТ Если вы хотите использовать согласование содержимого с минимальными API, то это возможно, но данная функциональность не является встроенной. В своем блоге я показываю, как добавить согласование содержимого к минимальным API с помощью библиотеки Carter с открытым исходным кодом: <http://mng.bz/o12d>.

Вопрос № 2 касается производительности. Всем хочется иметь самое производительное приложение, но возникает вопрос, насколько это важно. Собираетесь ли вы регулярно тестировать приложение и выявлять какие-либо отклонения? Если это так, то минимальные API, вероятно, будут лучшим выбором, поскольку зачастую они более производительны, чем контроллеры веб-API.

Фреймворк MVC, который используют контроллеры веб-API, описывается на множество соглашений и рефлексию для обнаружения контроллеров и сложного конвейера фильтров. Они, очевидно, высокопрограммированы, но если вы пишете приложение, в котором вам нужно выжимать каждую каплю пропускной способности, минимальные API, скорее всего, помогут вам быстрее достичь этой цели. Для большинства приложений накладные расходы фреймворка MVC будут незначительными по сравнению с любым доступом к базе данных или сети в приложении, поэтому об этом стоит беспокоиться, только если речь идет о приложениях, чувствительных к производительности.

Вопрос № 3 посвящен фильтрации. Мы познакомились с фильтрацией с помощью минимальных API в главе 5: фильтры позволяют присоединить конвейер обработки к конечным точкам минимального API и могут использоваться для таких вещей, как автоматическая валидация. Контроллеры веб-API (а также контроллеры MVC и страницы Razor) также имеют конвейер фильтров, но он гораздо сложнее, чем простой конвейер, используемый минимальными API, как вы увидите в главах 21 и 22.

В большинстве случаев фильтрация, обеспечиваемая минимальными API, будет вполне адекватной вашим потребностям. Основные случаи, когда такая фильтрация не работает, – это когда у вас уже есть приложение, применяющее контроллеры веб-API, и вы хотите повторно использовать сложные фильтры. В этих случаях может оказаться невозможно транслировать существующие фильтры веб-API в фильтры минимальных API. Если фильтрация важна, возможно, вам придется использовать контроллеры веб-API.

Отсюда вытекает вопрос № 4: вы создаете новое приложение или работаете над существующим? Если это новое приложение, я был бы категорически за использование минимальных API. С концептуальной точки зрения минимальные API проще, чем контроллеры веб-API, поэтому они работают быстрее, получая большое количество улучшений от команды ASP.NET Core. Если нет другой веской причины выбирать контроллеры веб-API в новом проекте, я предлагаю по умолчанию использовать минимальные API.

С другой стороны, если у вас уже есть приложение, использующее контроллеры веб-API, я был бы склонен придерживаться контроллеров веб-API. Хотя вполне возможно смешивать минимальные API

и контроллеры веб-API в одном приложении, я бы предпочел последовательность, вместо того чтобы рисковать.

В вопросе № 5 рассматривается, насколько вы уже знакомы с контроллерами веб-API. Если вы используете устаревшую версию ASP.NET или уже использовали контроллеры веб-API в ASP.NET Core и вам необходимо быстро повысить производительность, возможно, вы решите придерживаться контроллеров веб-API.

Я считаю это одним из самых слабых аргументов, поскольку минимальные API концептуально проще, чем контроллеры веб-API; если вы уже знакомы с контроллерами веб-API, то, скорее всего, легко освоите минимальные API. Тем не менее различия в подходах к привязке моделей могут немного сбивать с толку, и вы можете решить, что вложения не стоят того, или разочароваться, если что-то пойдет не так, как вы ожидаете.

Последний вопрос полностью зависит от вкуса и предпочтений: вам нравятся минимальные API? Контроллеры веб-API в значительной степени следуют парадигме «соглашение важнее конфигурации» (хотя и не в такой степени, как контроллеры MVC и Razor Pages). Наоборот, вы должны быть гораздо более явными с минимальными API. Минимальные API также не требуют какой-либо конкретной группировки, в отличие от контроллеров веб-API, которые следуют шаблону «методы действий в классе контроллера».

Разные люди предпочитают разные подходы. Контроллеры веб-API означают меньшее число подключений компонентов вручную, но это непременно означает больше магии и больше жесткости в структуре приложений.

Конечные точки минимальных API, напротив, должны явно добавляться в экземпляр `WebApplication`, но это также означает, что у вас появляется больше гибкости в отношении группировки конечных точек. Вы можете поместить все свои конечные точки в файл `Program.cs`, создать для них естественные группы в отдельных классах или создать файл для каждой конечной точки либо любого выбранного вами шаблона.

СОВЕТ Вы также можете упростить наложение вспомогательных фреймворков на минимальные API, например, используя Carter (<https://github.com/CarterCommunity/Carter>), который может предоставить структуру и поддержку, если вы этого хотите.

В целом что лучше для вашего приложения – контроллеры веб-API или минимальные API, – решать вам. В табл. 20.1 собраны вопросы по этой теме и указано, в каких случаях следует отдать предпочтение тому или иному подходу, но окончательный выбор остается за вами!

На этом мы подошли к концу главы, посвященной веб-API. В следующей главе мы рассмотрим одну из более сложных тем MVC и Razor Pages: конвейер фильтров и способы его использования для уменьшения дублирования в коде. Хорошая новость заключается в том, что в принципе он похож на фильтры минимальных API. А плохая новость состоит в том, что он гораздо сложнее!

Таблица 20.1. Минимальные API и контроллеры веб-API: что выбрать?

Вопрос	Минимальные API	Контроллеры веб-API
Вам нужна согласованность содержимого?	Не могут использовать согласованность содержимого «из коробки»	Встроенные и расширяемые
Насколько критична производительность?	Более производительные, чем контроллеры веб-API	Менее производительные, чем минимальные API
Сложная фильтрация?	Обладают простым и расширяемым конвейером фильтров	Обладают сложным, нелинейным конвейером фильтров
Это новый проект?	Минимальные API получают большое количество новых возможностей и находятся в центре внимания команды ASP.NET Core	Фреймворк MVC получает небольшие новые функции, но ему уделяется меньше внимания
Есть ли у вас опыт работы с контроллерами веб-API?	В значительной мере минимальные API разделяют те же концепции, но имеют небольшие различия в привязке моделей	Контроллеры веб-API могут быть знакомы пользователям устаревших версий ASP.NET или более старых версий ASP.NET Core
Вы предпочитаете соглашения, а не конфигурацию?	Требуют много явных настроек	Основаны на соглашениях и открытиях, которые могут показаться более магическими, если вы с ними незнакомы

Резюме

- Методы действий веб-API могут возвращать данные напрямую или использовать тип `ActionResult<T>` для генерации произвольного ответа. Если вы возвращаете несколько типов результата из метода действия, то сигнатура метода должна возвращать `ActionResult<T>`;
- данные, возвращаемые действием веб-API, называются моделью API. Она содержит данные, которые промежуточное ПО сериализует и отправит обратно клиенту. Это отличается от моделей представления и `PageModel`, которые содержат и данные, и метаданные о том, как сгенерировать ответ;
- веб-API ассоциируются с шаблонами маршрутов путем применения атрибутов `RouteAttributes` к методам действий. Это дает вам полный контроль над URL-адресами, образующими API вашего приложения;
- атрибуты маршрута, применяемые к контроллеру, объединяются с атрибутами методов действий, чтобы сформировать окончательный шаблон. Они также сочетаются с атрибутами унаследованных базовых классов. Можно использовать унаследованные атрибуты, чтобы уменьшить объем дублирующегося кода в атрибутах, например если вы используете общий префикс в маршрутах;

- по умолчанию имена контроллера и действия не имеют отношения к URL-адресам или шаблонам маршрутов, когда вы используете маршрутизацию на основе атрибутов. Однако можно использовать маркеры "[controller]" и "[action]" в своих шаблонах маршрутов, чтобы уменьшить повторы. Они будут заменены на имена текущего контроллера и действия;
- атрибуты [HttpPost] и [HttpGet] позволяют делать выбор между действиями на основе HTTP-метода запроса, когда два действия соответствуют одному и тому же URL-адресу. Это распространенный паттерн в REST-совместимых приложениях;
- атрибут [ApiController] применяет несколько общепринятых соглашений к контроллерам. Контроллеры, декорированные атрибутом, будут автоматически привязываться к телу запроса, вместо того чтобы использовать значения формы, автоматически сгенерируют ответ 400 Bad Request для недопустимых запросов и будут возвращать объекты ProblemDetails для ошибок кода состояния. Это может значительно сократить объем шаблонного кода, который вам приходится писать;
- вы можете контролировать, какое из соглашений применять, используя метод ConfigureApiBehaviorOptions() и предоставив лямбда-выражение конфигурации. Это полезно, например, если нужно подогнать свой API к существующей спецификации;
- по умолчанию ASP.NET Core форматирует модель API, возвращаемую из контроллера веб-API, как JSON. В отличие от предыдущей версии ASP.NET, данные в формате JSON сериализуются с использованием стиля camelCase, а не PascalCase. Следует учитывать это изменение, если вы получаете ошибки или отсутствующие значения при использовании данных из вашего API;
- в ASP.NET Core 3.0 и последующих версиях используется System.Text.Json, представляющая собой строгую высокопроизводительную библиотеку для сериализации и десериализации JSON. Этот сериализатор можно заменить на обычный форматер Newtonsoft.Json, вызвав метод AddNewtonsoftJson() для возвращаемого значения из метода services.AddControllers();
- согласование содержимого происходит, когда клиент указывает тип данных, которые он может обработать, а сервер на основе этого выбирает возвращаемый формат. Это позволяет нескольким клиентам вызывать ваш API и получать данные в понятном им формате;
- по умолчанию ASP.NET Core может возвращать text/plain, text/html и application/json, но можно добавить дополнительные форматеры, если нужно поддерживать другие форматы;
- можно добавить форматеры XML, вызвав метод AddXmlSerializerFormatters() для возвращаемого значения из метода services.AddControllers() в классе Startup. Это позволит форматировать ответ в формате XML, а также получить XML в теле запроса;
- согласование содержимого не используется, если заголовок Accept содержит */*, как в большинстве браузеров. Вместо этого прило-

жение будет использовать форматер по умолчанию, JSON. Можно отключить эту опцию, задав для параметра `RespectBrowserAcceptHeader` значение `true` при добавлении сервисов контроллера в файл `Program.cs`:

- можно сочетать контроллеры веб-API и конечные точки минимальных API в одном приложении, но, возможно, будет проще использовать что-то одно;
- выбирайте контроллеры веб-API, когда вам нужно согласование содержимого, когда у вас сложные требования к фильтрации, когда у вас есть опыт работы с веб-контроллерами или когда вы предпочитаете соглашения, а не конфигурацию для своих приложений;
- выбирайте конечные точки минимальных API, когда производительность имеет решающее значение, когда вы предпочитаете явную настройку автоматическим соглашениям или при запуске нового приложения.

21

Конвейер фильтров MVC и Razor Pages

В этой главе:

- конвейер фильтров и чем он отличается от промежуточного ПО;
- различные типы фильтров;
- порядок выполнения фильтров.

В третьей части мы подробно рассмотрели фреймворки MVC и Razor Pages. Вы узнали, как используется маршрутизация для выбора метода действия или страницы Razor, которую нужно выполнить. Вы также увидели, что такая привязка модели, валидация и как генерировать ответ с помощью возврата объекта `IActionResult` из действий и обработчиков страниц. В этой главе мы подробно рассмотрим фреймворки MVC и Razor Pages и познакомимся с *конвейером фильтров*, который иногда называют *конвейером вызова действий*, который аналогичен конвейеру фильтров конечных точек минимальных API, о котором вы узнали в главе 5.

MVC и Razor Pages используют несколько встроенных фильтров для решения сквозных задач, например авторизации (контроля того, какие пользователи к каким методам действий и страницам могут получить доступ в приложении). Любое приложение с концепцией пользователей будет использовать как минимум фильтры авторизации, но фильтры гораздо мощнее. В разделах 21.1 и 21.2 вы узнаете обо всех типах фильтров

и о том, как они объединяются для создания конвейера фильтров MVC для запроса, который достигает фреймворка MVC или Razor Pages.

Рассматривайте конвейер фильтров как мини-конвейер промежуточного ПО, работающий внутри MVC и Razor Pages, подобно конвейеру фильтров конечных точек минимальных API. Как и конвейер промежуточного ПО в ASP.NET Core, конвейер фильтров MVC состоит из ряда компонентов, соединенных в виде некоего «трубопровода», поэтому выходные данные одного фильтра поступают на вход следующего. В разделе 21.3 мы рассмотрим сходства и различия между этими двумя конвейерами, а также когда следует предпочесть один из них другому.

В разделе 21.4 вы увидите, как создать простой собственный фильтр. Вместо того чтобы сосредоточиваться на функциональности самого фильтра, в разделе 21.5 вы узнаете, как применить его к нескольким конечным точкам. В разделе 21.6 вы увидите, как выбор места применения атрибутов влияет на порядок выполнения фильтров.

Конвейер фильтров – сложная тема, но он может обеспечить расширенные возможности поведения в приложении и потенциально снизить общую сложность. В этой главе вы познакомитесь с основами конвейера и тем, как он работает. В главе 22 мы подробно рассмотрим практические примеры, изучим фильтры, входящие в состав ASP.NET Core, а также создадим собственные фильтры для извлечения общего кода из контроллеров и страниц Razor.

Прежде чем мы сможем приступить к написанию кода, следует разобраться с основами. В первом разделе этой главы объясняется, что такое конвейер, почему можно его использовать и чем он отличается от конвейера промежуточного ПО.

21.1 Что такое конвейер фильтров MVC

В этом разделе вы узнаете все о конвейере фильтров. Вы увидите, как он вписывается в жизненный цикл типичного запроса, и узнаете о шести типах фильтров.

Конвейер фильтров – относительно простая концепция, поскольку он обеспечивает *точки подключения* в обычном MVC-запросе, как показано на рис. 21.1. Например, вы хотите убедиться, что пользователи могут создавать или редактировать продукты в приложении для онлайн-торговли, только если они выполнили вход в приложение. Приложение будет перенаправлять анонимных пользователей на страницу входа, вместо того чтобы выполнить действие.

Если бы не было фильтров, то вам нужно было бы включить одинаковый код для проверки авторизованного пользователя на выполнении каждого конкретного метода действия. При таком подходе фреймворк MVC по-прежнему выполнял бы привязку и проверку модели, даже если пользователь не выполнил вход.

С помощью фильтров вы можете использовать *точки подключения* в запросе MVC для выполнения общего кода во всех или какой-то их

части. Таким образом, можно делать множество разных вещей, например:

- убедиться, что пользователь выполнил вход, перед выполнением метода действия, привязки модели или валидации;
- настроить выходной формат определенных методов действий;
- обработать ошибки валидации модели до вызова метода действия;
- перехватить исключения из метода действия и обработать их особым образом.

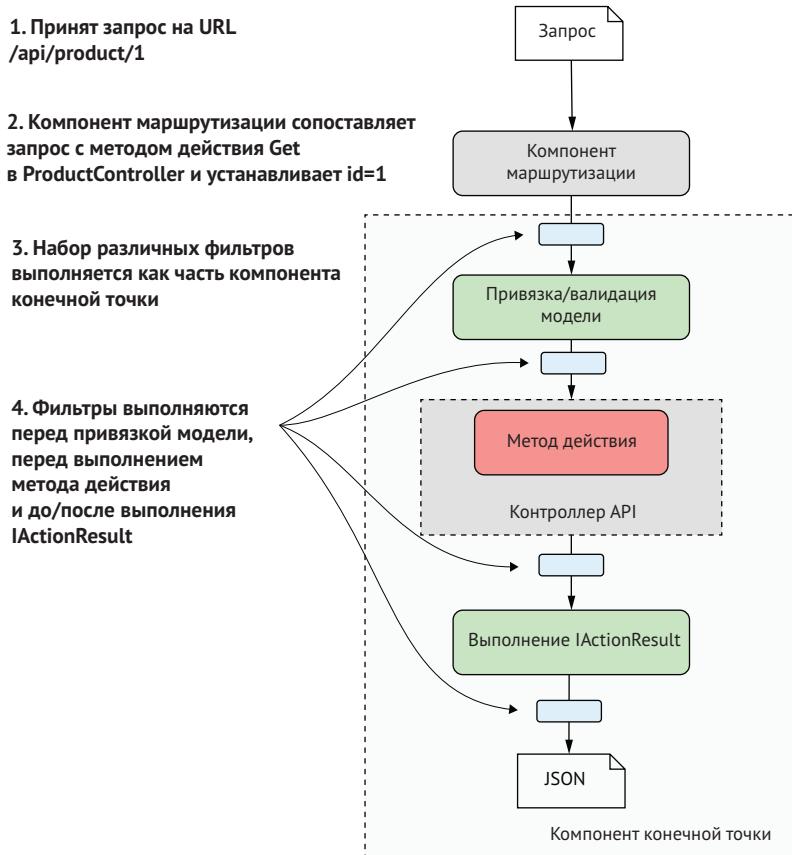


Рис. 21.1 Фильтры запускаются в нескольких точках EndpointMiddleware как часть обычной обработки запроса MVC. Аналогичный конвейер существует и для запросов Razor Page

Во многих отношениях конвейер фильтров похож на конвейер промежуточного ПО, но ограничен только запросами MVC и Razor Pages. Как и промежуточное ПО, фильтры хорошо подходят для решения сквозных задач приложения и во многих случаях являются полезным инструментом для уменьшения дублирования кода.

Линейное представление запроса MVC и конвейера фильтров, которое я использовал до сих пор, не *совсем* соответствует тому, как рабо-

тают эти фильтры. Есть пять типов фильтров, которые применяются к запросам MVC, каждый из которых выполняется на разных этапах фреймворка MVC, как показано на рис. 21.2:

- *фильтры авторизации* – запускаются первыми в конвейере, поэтому полезны для защиты ваших API и методов действий. Если фильтр авторизации считает, что запрос неавторизован, он завершит запрос, предотвращая выполнение остальной части конвейера (или действия);
- *фильтры ресурсов* – следующими после авторизации запускаются фильтры ресурсов. Они тоже могут выполнятся в *конце* конвейера, почти так же, как компоненты промежуточного ПО могут обрабатывать входящий запрос и исходящий ответ. Кроме того, фильтры ресурсов могут полностью прервать выполнение конвейера запросов и напрямую вернуть ответ. Благодаря более близкому расположению в конвейере фильтры ресурсов можно использовать по-разному. Можно добавить метрики к методу действия, предотвратить выполнение метода действия, если запрошен неподдерживаемый тип содержимого, или, когда они выполняются перед привязкой модели, контролировать способ работы привязки модели для этого запроса;
- *фильтры действий* – фильтры действий запускаются непосредственно до и после выполнения метода действия. Поскольку привязка модели уже произошла, фильтры действий позволяют управлять аргументами метода – до выполнения – или могут полностью прервать выполнение действия и вернуть другой объект `IActionResult`. Поскольку они тоже запускаются после выполнения действия, то могут дополнительно настроить объект `IActionResult`, возвращаемый действием, до того, как результат действия будет выполнен;
- *фильтры исключений* – фильтры исключений могут перехватывать исключения, возникающие в конвейере фильтров, и обрабатывать их соответствующим образом. Можно использовать фильтры исключений, чтобы написать специальный код обработки ошибок для MVC, который может быть полезен в некоторых ситуациях. Например, вы можете перехватить исключения в действиях API и отформатировать их иначе, чем исключения на страницах Razor;
- *фильтры результатов* – фильтры результатов запускаются до и после выполнения объекта метода действия `IActionResult`. Вы можете использовать фильтры результатов для управления выполнением результата или даже прервать его выполнение.

Какой именно фильтр вы выберете для реализации, будет зависеть от функциональности, которую вы пытаетесь ввести. Хотите как можно раньше прервать выполнение запроса? Фильтры ресурсов хорошо подходят для этого. Нужен доступ к параметрам метода действия? Используйте фильтр действий.



Рис. 21.2 Конвейер фильтров MVC и пять различных этапов обработки запроса. Некоторые фильтры (ресурсов, действий и результатов) запускаются дважды, до и после оставшейся части конвейера

Рассматривайте конвейер фильтров как небольшой конвейер промежуточного ПО, который живет сам по себе во фреймворке MVC. Также можно рассматривать фильтры как точки подключения в процессе вызова действия MVC, позволяющие выполнять код в определенный момент жизненного цикла запроса.

ПРИМЕЧАНИЕ Конструкция конвейера фильтров MVC сильно отличается от конвейера фильтров конечных точек минимальных API, который вы видели в главе 5. Конвейер фильтров конечных точек линеен и не имеет нескольких типов фильтров.

В этом разделе описывается, как конвейер фильтров работает с контроллерами MVC, например для создания API. Razor Pages использует почти идентичный конвейер фильтров.

21.2 Конвейер фильтров Razor Pages

Razor Pages использует ту же базовую архитектуру, что и контроллеры API, поэтому, возможно, неудивительно, что его конвейер фильтров практически идентичен. Единственная разница между конвейерами

заключается в том, что Razor Pages не использует фильтры действий, а использует *фильтры страниц*, как показано на рис. 21.3.

Фильтры авторизаций, ресурсов, исключений и результатов – это те же самые фильтры, которые вы видели в конвейере MVC. Они действуют одинаково, служат одним и тем же целям, и прервать их выполнение можно таким же образом.

ПРИМЕЧАНИЕ Эти фильтры – буквально те же классы, которые используются в Razor Pages и MVC.

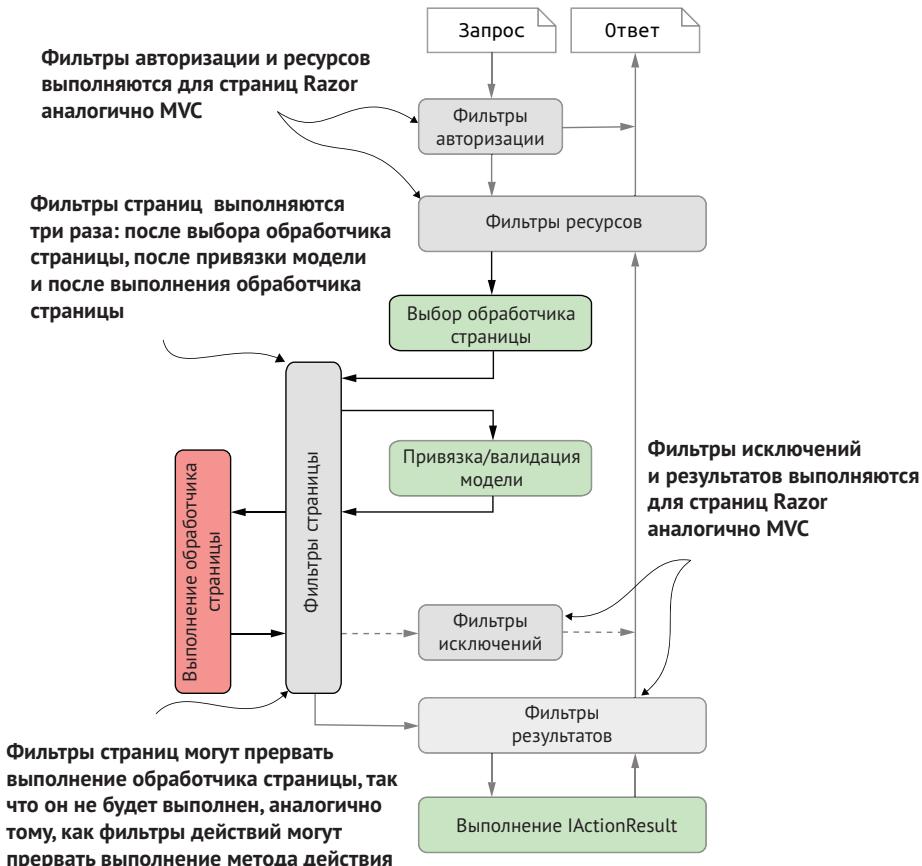


Рис. 21.3 Конвейер фильтров Razor Pages и пять различных этапов обработки запроса

Фильтры авторизаций, ресурсов, исключений и результатов выполняются точно так же, как и для конвейера MVC. Фильтры страниц – специфическая особенность Razor Pages.

Они выполняются в трех местах: после выбора обработчика страницы, после привязки модели и валидации и после выполнения обработчика страницы

Отличие конвейера фильтров Razor Pages заключается в том, что он использует фильтры страниц вместо фильтров действий. В отли-

чие от других типов фильтров, фильтры страниц запускаются в конвейере трижды:

- *после выбора обработчика страницы* – после выполнения фильтров ресурсов выбирается обработчик страницы на основе HTTP-метода запроса и значения маршрута {handler}, как вы узнали в главе 15. После выбора обработчика страницы метод фильтра страницы выполняется впервые. При этом прервать выполнение конвейера на данном этапе нельзя, а привязка модели и валидация еще не выполнены;
- *после привязки модели* – после выполнения первого фильтра страницы запрос привязывается к моделям привязки страницы Razor, и выполняется валидация. Это очень похоже на выполнение фильтра действий для контроллеров API. Здесь вы можете манипулировать данными, привязанными к модели, или полностью прервать выполнение обработчика страницы, возвращая другой объект IActionResult;
- *после выполнения обработчика страницы* – если вы не прерываете выполнение обработчика страницы, то фильтр страниц запускается в третий и последний раз после выполнения обработчика страницы. На данном этапе вы можете настроить IActionResult, возвращаемый обработчиком страницы перед выполнением результата.

Тройное выполнение фильтров страниц немножко затрудняет визуализацию конвейера, но вы можете рассматривать их просто как расширенные фильтры действий. Все, что можно делать с фильтром действий, можно делать и с фильтром страницы. Кроме того, при необходимости вы можете выполнить подключение после выбора обработчика страницы.

При каждом выполнении фильтра осуществляется отдельный метод соответствующего интерфейса, поэтому легко узнать, в каком месте конвейера вы находитесь, и при желании выполнить фильтр только в одном из возможных мест.

Один из основных вопросов, который я слышу, когда кто-то узнает о фильтрах ASP.NET Core: «Зачем они нам нужны?» Если конвейер фильтров похож на мини-конвейер промежуточного ПО, почему бы не использовать его компонент напрямую, вместо того чтобы вводить концепцию фильтра? Это интересный момент, о котором я расскажу в следующем разделе.

21.3 Фильтры или промежуточное ПО: что выбрать?

Конвейер фильтров во многом похож на конвейер промежуточного ПО, но здесь есть несколько тонких отличий, которые следует учитывать при выборе того, какой подход использовать. При рассмотрении схожих особенностей у них есть три основные параллели:

- *запросы проходят через компонент промежуточного ПО на пути «внутрь», а ответы снова проходят через него уже «на выходе».* Фильтры ресурсов, действий и результатов также являются двусторонними, хотя фильтры авторизации и исключений запускаются только один раз для запроса, а фильтры страниц – трижды;

- промежуточное ПО может завершить запрос, вернув ответ, вместо того чтобы передать его более позднему компоненту. Фильтры также могут прервать выполнение конвейера, возвращая ответ;
- промежуточное ПО часто используется для решения сквозных задач приложения, таких как журналирование, профилирование производительности и обработка исключений. Фильтры также подходят для решения такого рода проблем.

И напротив, между ними есть три основных различия:

- промежуточное ПО может выполняться для всех запросов; фильтры будут работать только для запросов, доходящих до компонента `EndpointMiddleware` и выполняющих действие контроллера API или страницу Razor;
- фильтры имеют доступ к конструкциям MVC, таким как `ModelState` и `IActionResult`s. Промежуточное ПО, как правило, не зависит от MVC и Razor Pages и работает на «более низком уровне», поэтому не может использовать эти концепции;
- фильтры можно легко применить к подмножеству запросов; например, всем действиям в одном контроллере или одной странице Razor. Промежуточное ПО не имеет этой концепции в качестве полноправной сущности (хотя подобного можно было бы добиться с помощью специальных компонентов промежуточного ПО).

Все это хорошо, но как интерпретировать эти различия? Когда и что следует выбрать?

Мне нравится рассматривать промежуточное ПО и фильтры как вопрос специфики. Промежуточное ПО – более общая концепция, которая работает с низкоуровневыми примитивами, такими как `HttpContext`, поэтому имеет более широкий охват. Если необходимая вам функциональность не имеет специфических требований, связанных с MVC, то вы должны использовать компонент промежуточного ПО.

«Промежуточное ПО или фильтры» – тонкий вопрос, и не важно, что вы выберете, если это вам подходит. Вы даже можете использовать компоненты промежуточного ПО внутри конвейера фильтров как фильтры, но эта тема выходит за рамки данной книги.

СОВЕТ Промежуточное ПО в качестве фильтров появилось в ASP.NET Core версии 1.1 и также доступно в более поздних версиях. Классический вариант использования – локализация запросов на несколько языков. У меня есть серия статей о том, как использовать эту функцию: <http://mng.bz/RXa0>.

По отдельности фильтры могут быть немного абстрактными, поэтому в следующем разделе мы взглянем на код и узнаем, как написать собственный фильтр в ASP.NET Core.

21.4 Создание простого фильтра

В этом разделе я покажу, как создать свои первые фильтры; в разделе 21.5 вы увидите, как применить их к контроллерам и действиям

MVC. Начнем с малого – создадим фильтры, которые просто осуществляют вывод в консоль, а в главе 22 рассмотрим более практические примеры и обсудим некоторые их нюансы.

Фильтр определенного этапа реализуется с помощью одной из пары интерфейсов – одного синхронного (sync) и одного асинхронного (async):

- *фильтры авторизации* – `IAuthorizationFilter` или `IAsyncAuthorizationFilter`;
- *фильтры ресурсов* – `IResourceFilter` или `IAsyncResourceFilter`;
- *фильтры действий* – `IActionFilter` или `IAsyncActionFilter`;
- *фильтры страниц* – `IPageFilter` или `IAsyncPageFilter`;
- *фильтры исключений* – `IExceptionFilter` или `IAsyncExceptionFilter`;
- *фильтры результатов* – `IResultFilter` или `IAsyncResultFilter`.

Для реализации фильтра можно использовать любой класс РОСО, но обычно они реализуются как атрибуты C#, которые можно использовать для декорирования контроллеров, действий и страниц Razor Pages, как будет показано в разделе 21.5. Синхронный и асинхронный интерфейсы позволяют добиться одинаковых результатов, поэтому вариант, который выберете вы, должен зависеть от того, требуется ли сервисам, которые вы вызываете в фильтре, поддержка асинхронного режима.

ПРИМЕЧАНИЕ Следует реализовать один из интерфейсов, но не оба. Если вы реализуете оба варианта, будет использоваться только асинхронный интерфейс.

В листинге 21.1 показан фильтр ресурсов, который реализует интерфейс `IResourceFilter` и осуществляет вывод в консоль при выполнении. Метод `OnResourceExecuting` вызывается, когда запрос сначала достигает этапа обработки, на котором вызывается фильтр ресурсов. А вот метод `OnResourceExecuted` вызывается после выполнения остальной части конвейера: после привязки модели, выполнения действия, выполнения результата и запуска всех промежуточных фильтров.

Листинг 21.1 Пример фильтра ресурсов, реализующего интерфейс `IAsyncResourceFilter`

```
public class LogResourceFilter : Attribute, IResourceFilter
{
    public void OnResourceExecuting(
        ResourceExecutingContext context)
    {
        Console.WriteLine("Executing!");
    }

    public void OnResourceExecuted(
        ResourceExecutedContext context)
    {
        Console.WriteLine("Executed");
    }
}
```

Содержит дополнительную информацию, как, например, `IActionResult`, который возвращает результат метода действия

Выполняется в начале конвейера после фильтров авторизации

Контекст содержит объект текущего `HttpContext`, детали маршрутизации и информацию о текущем действии

Выполняется после привязки модели, выполнения метода действия и метода выполнения результата

Методы интерфейса просты и аналогичны для каждого этапа конвейера, передавая объект контекста в качестве параметра метода. У каждого из синхронных фильтров с двумя методами есть методы *Executing и *Executed. Тип аргумента разный для каждого фильтра, но он содержит все сведения о конвейере.

Например, `ResourceExecutingContext`, передаваемый фильтру ресурсов, содержит сам объект `HttpContext`, подробные сведения о маршруте, который выбрал это действие, детали самого действия и т. д. Контексты для более поздних фильтров будут содержать дополнительные подробности, такие как аргументы метода действия для фильтра действий и `ModelState`.

Объект контекста для метода `ResourceExecutedContext` похож, но он также содержит сведения о том, как исполнилась остальная часть конвейера. Вы можете проверить, есть ли необработанное исключение, посмотреть, не было ли прервано выполнение конвейера фильтров на этом же этапе другим фильтром, или увидеть `IActionResult`, используемый для генерации ответа.

Эти мощные объекты являются ключом к расширенному поведению фильтров, такому как прерывание выполнения конвейера и обработка исключений. Мы будем использовать их в главе 22 при создании более сложных примеров фильтров.

Асинхронная версия фильтра ресурсов требует реализации одного метода, как показано в листинге 21.2. Что касается синхронной версии, то вам передается объект `ResourceExecutingContext` в качестве аргумента и делегат, представляющий оставшуюся часть конвейера. Вы должны вызвать этот делегат (асинхронно), чтобы выполнить оставшуюся часть конвейера и вернуть экземпляр `ResourceExecutedContext`.

Листинг 21.2 Пример фильтра ресурсов, реализующего интерфейс `IAsyncResourceFilter`

```
public class LogAsyncResourceFilter : Attribute, IAsyncResourceFilter
{
    public async Task OnResourceExecutionAsync(
        ResourceExecutingContext context,
        ResourceExecutionDelegate next) {
        {
            Console.WriteLine("Executing async!");
            ResourceExecutedContext executedContext = await next();
            Console.WriteLine("Executed async!");
        }
    }
}
```

Вызывается до выполнения остальной части конвейера

Выполняется в начале конвейера после фильтров авторизации

Вам предоставляется делегат, который инкапсулирует остальную часть конвейера фильтров

Выполняет остальную часть конвейера и получает ResourceExecutedContext

Реализации синхронных и асинхронных фильтров имеют небольшие различия, но для большинства целей они идентичны. Я рекомендую по возможности реализовать синхронную версию и возвращаться к асинхронной только по необходимости.

Мы создали несколько фильтров, поэтому мы должны посмотреть, как использовать их в приложении. В следующем разделе мы рассмотрим две конкретные проблемы: как контролировать, какие запросы

выполняют ваши новые фильтры, и как контролировать порядок, в котором они выполняются.

21.5 Добавляем фильтры к действиям и страницам Razor Pages

В разделе 21.3 мы обсуждали сходства и различия между промежуточным ПО и фильтрами. Одно из этих различий заключается в том, что фильтры могут быть привязаны к определенным действиям или контроллерам, чтобы они запускались только для определенных запросов.

Как вариант можно применить фильтр глобально, чтобы он запускался для каждого действия MVC и страницы Razor. Добавляя фильтры разными способами, можно добиться разных результатов. Представьте, что у вас есть фильтр, который заставляет вас выполнить вход в приложение для выполнения действия. То, как вы добавляете фильтр в приложение, существенно изменит его поведение:

- *применение фильтра к отдельному действию или странице Razor* – анонимные пользователи могут просматривать приложение как обычно, но если они попытаются получить доступ к защищенному действию или странице Razor, их вынудят выполнить вход;
- *применение фильтра к контроллеру* – анонимные пользователи могут получить доступ к действиям из других контроллеров, но доступ к любому действию на защищенном контроллере заставит их выполнить вход;
- *применение фильтра глобально* – пользователи не могут использовать приложение, не выполнив вход. Любая попытка получить доступ к действию или странице Razor перенаправит пользователя на страницу входа.

ПРИМЕЧАНИЕ В ASP.NET Core уже есть такой фильтр: AuthorizeFilter. Я расскажу о нем в главе 22, а подробности вы узнаете в главе 24.

Как я уже писал в предыдущем разделе, обычно фильтры создаются как атрибуты, и для этого есть веская причина – так проще применять их к контроллерам MVC, действиям и страницам Razor. В этом разделе вы увидите, как применить фильтр LogResourceFilter из листинга 21.1 к действию, контроллеру, странице Razor и глобально. Уровень, на котором применяется фильтр, называется *областью действия*.

ОПРЕДЕЛЕНИЕ *Область действия* фильтра определяет, к какому количеству действий он применяется. Она может быть ограничена методом действия, контроллером, страницей Razor или глобально. Мы начнем с наиболее конкретной области – применим фильтры к одному действию. В следующем листинге показан пример контроллера MVC, у которого есть два метода действия: один с фильтром LogResourceFilter и второй без него.

Листинг 21.3 Применение фильтров к методу действия

```
public class RecipeController : ControllerBase
{
    [LogResourceFilter]
    public IActionResult Index()
    {
        return Ok();
    }
    public IActionResult View()
    {
        return OK();
    }
}
```

LogResourceFilter будет работать как часть конвейера при выполнении этого действия

У этого метода действия нет фильтров на уровне действия

В качестве альтернативы, если вы хотите применить один и тот же фильтр к каждому методу действия, можно добавить атрибут в область контроллера, как показано в следующем листинге. Каждый метод действия в контроллере будет использовать `LogResourceFilter`, без необходимости специально декорировать методы.

Листинг 21.4 Применение фильтров к контроллеру

```
[LogResourceFilter]
public class RecipeController : ControllerBase
{
    public IActionResult Index ()
    {
        return Ok();
    }
    public IActionResult View()
    {
        return Ok();
    }
}
```

LogResourceFilter добавляется к каждому действию контроллера

Каждое действие в контроллере декорировано фильтром

Что касается страниц Razor, то можно применять атрибуты к `PageModel`, как показано в следующем листинге. Фильтр применяется ко всем обработчикам страниц на странице Razor – невозможно применить фильтры к одному обработчику; вы должны применять их на уровне страницы.

Листинг 21.5 Применение фильтров к странице Razor

```
[LogResourceFilter]
public class IndexModel : PageModel
{
    public void OnGet()
    {
    }
    public void OnPost()
    {
    }
}
```

LogResourceFilter добавляется к PageModel страницы Razor

Фильтр применяется к каждому обработчику страницы на странице

Фильтры, которые вы применяете в качестве атрибутов к контроллерам, действиям и страницам Razor, автоматически обнаруживаются фреймворком при запуске приложения. Что касается общих атрибутов, можно пойти еще дальше и применить фильтры глобально, без необходимости декорировать отдельные классы.

Глобальные фильтры добавляются иначе, нежели фильтры, применяемые к контроллерам или действиям, – они добавляются непосредственно к сервисам MVC при конфигурировании контроллеров и страниц Razor.

В этом листинге показаны три эквивалентных способа глобального применения фильтра.

Листинг 21.6 Глобальное применение фильтров к приложению

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.Filters.Add(new LogResourceFilter());
    options.Filters.Add(typeof(LogResourceFilter));
    options.Filters.Add<LogResourceFilter>();
});
```

Кроме того, фреймворк может создать глобальный фильтр, используя параметр обобщенного типа

Добавляет фильтры с помощью объекта MvcOptions

Вы можете передать экземпляр фильтра напрямую...

...или передать тип фильтра и позволить фреймворку создать его

Можно настроить MvcOptions с помощью перегруженного варианта метода AddControllers(). Когда вы настраиваете фильтры глобально, они применяются как к контроллерам, так и ко всем страницам Razor в приложении. Если вы используете в приложении страницы Razor, то перегруженного варианта для настройки MvcOptions нет. Вместо этого нужно использовать метод расширения AddMvcOptions() для настройки фильтров, как показано в следующем листинге.

Листинг 21.7 Глобальное применение фильтров к приложению Razor Pages

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.RazorPages()
    .AddMvcOptions(options =>
{
    options.Filters.Add(new LogResourceFilter());
    options.Filters.Add(typeof(LogResourceFilter));
    options.Filters.Add<LogResourceFilter>();
});
```

Данный метод не позволяет передавать лямбда-функцию для настройки MvcOptions

Вы должны использовать метод расширения, чтобы добавить фильтры к объекту MvcOptions

Вы можете настроить фильтры любым из способов, показанных ранее

Имея в наличии три потенциально разные области действия, вам часто будут встречаться методы действий, к которым применено несколько фильтров: некоторые применяются непосредственно к методу действия, а другие унаследованы от контроллера или глобально. Тогда возникает вопрос: какой фильтр запускается первым?

21.6 Порядок выполнения фильтров

Вы видели, что конвейер фильтров содержит пять различных этапов, по одному для каждого типа фильтра. Эти этапы всегда выполняются в фиксированном порядке, описанном в разделах 21.1 и 21.2. Но на каждом этапе у вас также может быть несколько фильтров одного типа (например, несколько фильтров ресурсов), которые являются частью конвейера одного метода действия. Все они могут иметь несколько *областей действия*, в зависимости от того, как вы их добавили, как было показано в последнем разделе.

В этом разделе мы рассмотрим *порядок фильтров в рамках определенного этапа* и то, как на него влияет область действия. Мы начнем с рассмотрения порядка по умолчанию, а затем перейдем к способам настройки порядка в соответствии с вашими требованиями.

21.6.1 Порядок выполнения фильтров по умолчанию

Размышляя о порядке выполнения фильтров, важно помнить, что фильтры ресурсов, действий и результатов реализуют два метода: **Executing* перед методом и **Executed* после метода. Кроме того, фильтры страниц реализуют три метода!

Порядок, в котором выполняется каждый метод, зависит от области действия фильтра, как показано на рис. 21.4 для этапа обработки запроса, на котором вызывается фильтр ресурсов.

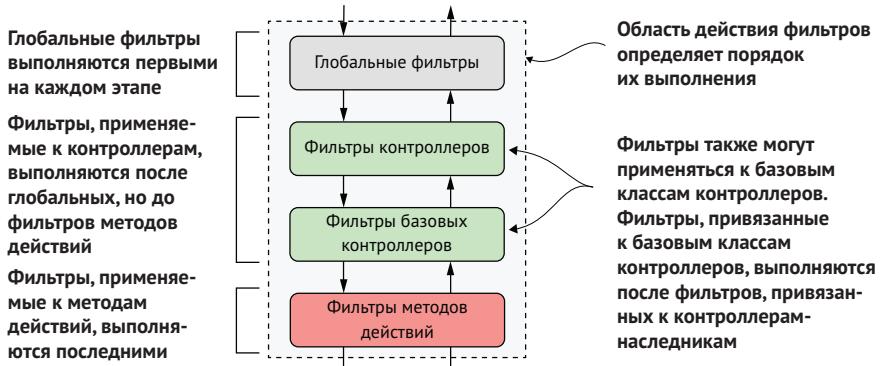


Рис. 21.4 Порядок выполнения фильтров по умолчанию на определенном этапе в зависимости от области действия фильтров. Для метода **Executing* сначала запускаются глобальные фильтры, затем фильтры, применяемые к контроллерам, и, наконец, фильтры, применяемые к действиям. Что касается метода **Executed*, то здесь фильтры запускаются в обратном порядке

По умолчанию фильтры выполняются от самой широкой области (глобальные) до самой узкой (действие) при выполнении метода **Executing* для каждого этапа. Методы **Executed* для фильтров выполняются в обратном порядке.

Порядок выполнения для страниц Razor несколько проще, учитывая, что у вас всего две области действия – глобальные фильтры и фильтры,

которые применяются к страницам Razor. Для страниц Razor глобальные фильтры сначала выполняют методы `*Executing` и `PageHandlerSelected`, а затем фильтры страниц. Для методов `*Executed` фильтры работают в обратном порядке. Иногда вы будете сталкиваться с тем, что вам нужно более тщательно контролировать этот порядок, особенно если у вас, например, есть несколько фильтров действий, применяемых в одной и той же области действия. Конвейер фильтров удовлетворяет это требование посредством интерфейса `IOrderedFilter`.

21.6.2 Переопределение порядка выполнения фильтров по умолчанию с помощью интерфейса `IOrderedFilter`

Фильтры отлично подходят для выделения кода решения сквозных задач из действий вашего контроллера и страницы Razor, но если к действию применено несколько фильтров, вам часто придется контролировать точный порядок, в котором они выполняются.

Область действия может помочь вам в этом, но для других случаев можно реализовать интерфейс `IOrderedFilter`. Он состоит из единственного свойства `Order`:

```
public interface IOrderedFilter
{
    int Order { get; }
}
```

Вы можете реализовать это свойство в фильтрах, чтобы задать порядок их выполнения. Конвейер фильтров упорядочивает фильтры на определенном этапе сначала на основе этого значения, от наименьшего до наивысшего, и использует порядок выполнения фильтров по умолчанию для упорядочивания фильтров с одинаковыми значениями порядка выполнения, как показано на рис. 21.5.

Фильтры для `Order = -1` выполняются первыми, поскольку имеют наименьшее значение `Order`. Фильтр контроллеров выполняется первым, потому что имеет более широкую область действия, чем фильтр действий. Далее выполняются фильтры с `Order = 0` в порядке по умолчанию, как показано на рис. 21.5. В конце выполняется фильтр с `Order = 1`.

По умолчанию, если фильтр не реализует интерфейс `IOrderedFilter`, предполагается, что у него `Order = 0`. Все фильтры, поставляемые как часть ASP.NET Core, имеют `Order = 0`, поэтому вы можете добавлять собственные фильтры как перед, так и после встроенных.

ПРИМЕЧАНИЕ Вы можете полностью настроить способ построения конвейера фильтров, настроив соглашения о модели приложения фреймворка MVC. Они контролируют все, что касается того, как обнаруживаются контроллеры и страницы Razor, как они добавляются в конвейер и как обнаруживаются фильтры. Это продвинутая концепция, которая вам не часто понадобится, но иногда она может пригодиться. О модели приложения MVC можно прочитать в документации на странице <http://mng.bz/nWNa>.

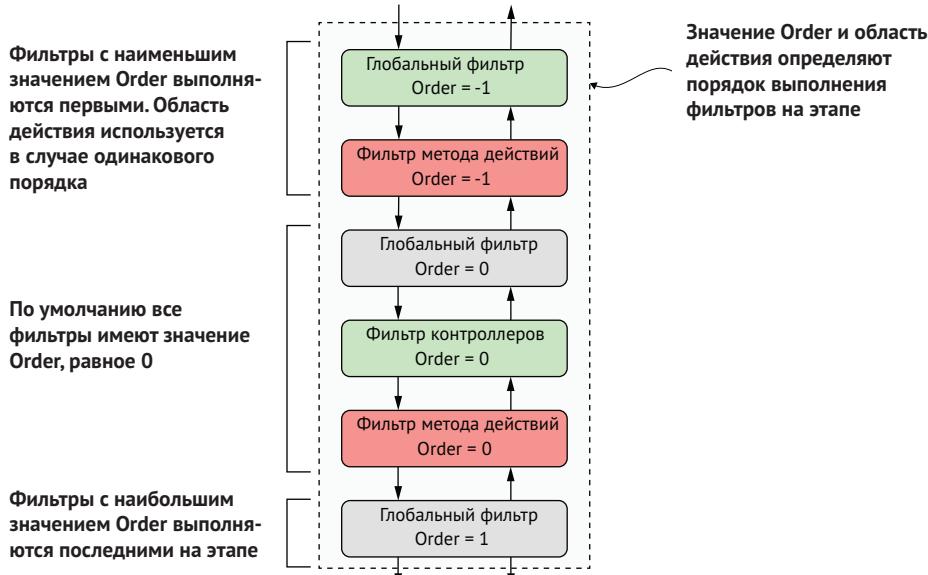


Рис. 21.5 Управление порядком выполнения фильтров для определенного этапа с помощью интерфейса `IOrderedFilter`. Фильтры сначала упорядочиваются по свойству `Order`, а затем по области действия

В этой главе представлено много сведений о конвейере фильтров MVC, и мы рассмотрели большинство технических деталей, необходимых для использования фильтров и создания настраиваемых реализаций для собственного приложения. В главе 22 вы увидите встроенные фильтры, предоставляемые ASP.NET Core, а также практические примеры фильтров, которые вы, возможно, захотите использовать в своих приложениях.

Резюме

- Конвейер фильтров предоставляет точки подключения для запроса MVC, поэтому вы можете выполнять функции в различных точках запроса MVC. С помощью фильтров можно запускать код в определенных точках процесса MVC для всех запросов или подмножества запросов. Это особенно полезно для решения сквозных задач, характерных для MVC;
- конвейер фильтров выполняется как часть MVC или Razor Pages. Он состоит из фильтров авторизации, ресурсов, действий, страниц, исключений и фильтров результатов. Каждый тип фильтра группируется в этап, и его можно использовать для достижения конкретного эффекта для этого этапа;
- фильтры ресурсов, действий и результатов запускаются в конвейере дважды: метод `*Executing` на входе и метод `*Executed` на выходе. Фильтры страниц запускаются трижды: после выбора обработчика страницы и до и после выполнения обработчика страницы;

- фильтры авторизации и исключений запускаются только один раз как часть конвейера; они не запускаются после генерации ответа;
- у каждого типа фильтра есть синхронная и асинхронная версии. Например, фильтры ресурсов могут реализовывать либо интерфейс `IResourceFilter`, либо интерфейс `IAsyncResourceFilter`. Следует использовать синхронный интерфейс, если только фильтру не нужно использовать вызовы асинхронных методов;
- можно добавлять фильтры глобально, на уровне контроллера, на уровне страницы Razor Page или на уровне действия. Это называется *областью применения* фильтра. Какую область применения следует выбрать, зависит от того, насколько широко вы хотите применить фильтр;
- на определенном этапе сначала выполняются фильтры с глобальной областью применения, затем фильтры, применяемые к контроллерам и, наконец, фильтры для действий. Вы также можете переопределить порядок по умолчанию, реализовав интерфейс `IOrderedFilter`. Фильтры будут запускаться в порядке возрастания значения свойства `Order` и использовать указанную область применения.



Создание собственных фильтров MVC и страниц Razor

В этой главе:

- создание собственных фильтров для рефакторинга сложных методов действия;
- использование фильтров авторизации для защиты методов действий и Razor Pages;
- прерывание выполнения конвейера фильтров, чтобы игнорировать действие и выполнение обработчика страниц;
- внедрение зависимостей в фильтры.

В предыдущей главе вы познакомились с конвейером фильтров MVC и Razor Pages и увидели, какое место они занимают в жизненном цикле запроса. Вы узнали, как применять фильтры к методу действий, контроллерам и страницам Razor, а также о влиянии области применения на порядок выполнения фильтра.

В этой главе мы воспользуемся этими знаниями и применим их к конкретному примеру. Вы научитесь создавать собственные фильтры, которые можно использовать в своих приложениях, и применять их для уменьшения дублирования кода в методах действий.

В разделе 22.1 я подробно расскажу о типах фильтров, о том, как они вписываются в конвейер MVC и для чего их использовать. Для каждого из них я приведу примеры реализаций, которые вы можете использовать в своем приложении, и опишу доступные встроенные опции.

Ключевой особенностью фильтров является возможность прерывания выполнения запроса путем генерации ответа и остановки прохождения через конвейер. Это напоминает замыкание в конвейере промежуточного ПО, но для фильтров MVC есть небольшие различия. Кроме того, поведение каждого фильтра немного отличается, и я расскажу об этом в разделе 22.2.

Обычно фильтры MVC добавляют в конвейер, реализуя их как атрибуты, добавляемые в классы контроллера, методы действий и страницы Razor. К сожалению, нельзя просто использовать внедрение зависимостей с атрибутами из-за ограничений языка C#. В разделе 22.3 я покажу вам, как использовать базовые классы `ServiceFilterAttribute` и `TypeFilterAttribute` для активации внедрения зависимостей в фильтрах.

Мы рассмотрели все основы фильтров в главе 21, поэтому в следующем разделе мы сразу перейдем к коду и начнем создавать собственные фильтры MVC.

22.1 Создание собственных фильтров для приложения

ASP.NET Core включает в себя ряд фильтров, которые можно использовать «из коробки», но зачастую самые полезные фильтры – это собственные фильтры, созданные конкретно для приложения. В этом разделе мы проработаем каждый из шести типов фильтров. Я объясню подробнее, для чего они нужны и когда нужно их использовать. Я выделю примеры фильтров, которые являются частью самого фреймворка ASP.NET Core, и вы увидите, как создавать собственные фильтры для приложения, используемого в качестве примера.

Чтобы вы могли работать с чем-то реалистичным, начнем с контроллера веб-API для доступа к приложению рецептов из главы 12. Этот контроллер содержит два действия: одно для получения `RecipeDetailViewModel` и второе, чтобы обновить сущность `Recipe`, добавляя новые значения. В этом листинге показана наша отправная точка для этой главы, включая оба метода действия.

Листинг 22.1 Контроллер веб-API Recipe перед рефакторингом для использования фильтров

```
[Route("api/recipe")]
public class RecipeApiController : ControllerBase
{
    private readonly bool IsEnabled = true; ←
    public RecipeService _service;
    public RecipeApiController(RecipeService service)
```

Это поле будет передано как конфигурация и используется для управления доступом к действиям

```

    {
        _service = service;
    }

    [HttpGet("{id}")]
    public IActionResult Get(int id)
    {
        if (!IsEnabled) { return BadRequest(); } ← Если API не активирован, блокируем дальнейшее выполнение
        try
        {
            if (!_service.DoesRecipeExist(id))
            {
                return NotFound(); ← Если запрашиваемый рецепт не существует, возвращаем ответ 404
            }
            var detail = _service.GetRecipeDetail(id);
            Response.GetTypedHeaders().LastModified =
                detail.LastModified; ← Получаем модель представления RecipeDetailViewModel
            return Ok(detail); ← Возвращаем модель представления с ответом 200
        }
        catch (Exception ex)
        {
            return GetErrorResponse(ex); ← Если возникает исключение, перехватываем его и возвращаем ошибку в ожидаемом формате, например ошибка 500
        }
    }

    [HttpPost("{id}")]
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
        if (!IsEnabled) { return BadRequest(); } ← Если API не включен, заблокируйте дальнейшее выполнение
        try
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState); ← Проверяем модель привязки и возвращаем ответ 400, если есть ошибки
            }
            if (!_service.DoesRecipeExist(id))
            {
                return NotFound(); ← Обновляем рецепт из команды и возвращаем ответ 200
            }
            _service.UpdateRecipe(command);
            return Ok();
        }
        catch (Exception ex)
        {
            return GetErrorResponse(ex); ← Если возникает исключение, перехватываем его и возвращаем ошибку в ожидаемом формате, например ошибка 500
        }
    }

    private static IActionResult GetErrorResponse(Exception ex)
    {
        var error = new ProblemDetails
    }

```

Задаем для заголовка ответа LastModified значение в модели

Возвращаем модель представления с ответом 200

Если запрашиваемый рецепт не существует, возвращаем ответ 404

Если API не активирован, блокируем дальнейшее выполнение

Если запрашиваемый рецепт не существует, возвращаем ответ 404

Получаем модель представления RecipeDetailViewModel

Если возникает исключение, перехватываем его и возвращаем ошибку в ожидаемом формате, например ошибка 500

Если API не включен, заблокируйте дальнейшее выполнение

Проверяем модель привязки и возвращаем ответ 400, если есть ошибки

Обновляем рецепт из команды и возвращаем ответ 200

Если возникает исключение, перехватываем его и возвращаем ошибку в ожидаемом формате, например ошибка 500

```
{
    Title = "An error occurred",
    Detail = context.Exception.Message,
    Status = 500,
    Type = "https://httpstatuses.com/500"
};

return new ObjectResult(error)
{
    StatusCode = 500
};
}
}
```

Эти методы действий содержат большое количество кода, который скрывает намерение каждого действия. Между методами также довольно много дублирования кода, например когда проверяется, существует ли сущность `Recipe`, и при форматировании исключений.

В этом разделе мы проведем рефакторинг этого контроллера, чтобы использовать фильтры для всего кода в методах, который не связан с целью каждого действия. К концу главы у нас будет гораздо более простой контроллер, который будет намного легче понять.

Листинг 22.2 Контроллер веб-API Recipe после рефакторинга

```
[Route("api/recipe")]
[ValidateModel]
[HandleException]
[FeatureEnabled(IsEnabled = true)]
public class RecipeApiController : ControllerBase
{
    public RecipeService _service;
    public RecipeApiController(RecipeService service)
    {
        _service = service;
    }

    [HttpGet("{id}")]
    [EnsureRecipeExists]
    [AddLastModifiedHeader]
    public IActionResult Get(int id)
    {
        var detail = _service.GetRecipeDetail(id);
        return Ok(detail);
    }

    [HttpPost("{id}")]
    [EnsureRecipeExists]
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
```

Фильтры инкапсулируют большую часть логики, общей для нескольких методов действий

Размещение фильтров на уровне действия ограничивает их одним действием

Намерение действия – вернуть модель представления `Recipe` – намного яснее

Размещая фильтры на уровне действий, вы можете контролировать порядок, в котором они выполняются

```

    _service.UpdateRecipe(command);
    return Ok();
}
}

```

**Намерение действия – обновить
Recipe – гораздо яснее**

Я думаю, вы должны согласиться, что контроллер из листинга 22.2 намного легче читать! В этом разделе мы проведем постепенный рефакторинг контроллера, удаляя сквозной код, чтобы получить нечто, более управляемое. Все фильтры, которые мы создадим в этом разделе, будут использовать интерфейсы синхронных фильтров – что касается создания их асинхронных аналогов, то я оставлю это вам в качестве упражнения. Мы начнем с рассмотрения фильтров авторизации и посмотрим, как они связаны с безопасностью в ASP.NET Core.

22.1.1 Фильтры авторизации: защита API

Аутентификация и авторизация – это взаимосвязанные фундаментальные концепции безопасности, которые мы подробнее рассмотрим в главах 23 и 24.

ОПРЕДЕЛЕНИЕ Аутентификация касается определения того, кто сделал запрос. Авторизация связана с тем, к чему пользователю разрешен доступ.

Фильтры авторизации запускаются в конвейере фильтров MVC первыми перед всеми остальными фильтрами. Они контролируют доступ к методу действия, тотчас же прерывая выполнение конвейера, если запрос не соответствует необходимым требованиям.

В ASP.NET Core имеется встроенный фреймворк для авторизации, который следует использовать, когда вам необходимо защитить приложение MVC или веб-API. Его можно сконфигурировать с помощью специальных политик, которые позволяют точно контролировать доступ к действиям.

СОВЕТ Можно написать собственные фильтры авторизации, реализовав интерфейсы `IAuthorizationFilter` или `IAsyncAuthorizationFilter`, но я настоятельно рекомендую не делать этого. Фреймворк для авторизации ASP.NET Core легко настраивается и должен отвечать всем вашим потребностям.

В основе фреймворка авторизации ASP.NET Core лежит фильтр авторизации `AuthorizeFilter`, который можно добавить в конвейер фильтров, декорировав действия или контроллеры атрибутом `[Authorize]`. В самом простом виде добавление атрибута `[Authorize]` к действию, как показано в следующем листинге, означает, что запрос должен быть выполнен проверенным пользователем, чтобы ему было позволено двигаться дальше. Если вы не выполнили вход, выполнение конвейера прерывается, а в браузер возвращается ответ 401 Unauthorized.

Листинг 22.3 Добавление атрибута [Authorize] к методу действия

```
public class RecipeApiController : ControllerBase
{
    public IActionResult Get(int id)
    {
        // Тело метода;
    }

    [Authorize]           ← Добавляем фильтр авторизации в конвейер фильтров с помощью атрибута [Authorize]
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
        // Тело метода;
    }
}
```

Метод `Get` не имеет атрибута `[Authorize]`, поэтому его может выполнить кто угодно

Метод `Edit` может быть выполнен только в том случае, если вы выполнили вход

Как и в случае со всеми фильтрами, можно применить атрибут `[Authorize]` на уровне контроллера, чтобы защитить все действия в контроллере, к странице Razor, на все методы обработчиков страниц или даже глобально для защиты всех конечных точек приложения.

ПРИМЕЧАНИЕ Мы подробно рассмотрим авторизацию в главе 24, в том числе и то, как добавить более подробные требования, чтобы только определенные группы пользователей могли выполнять действие.

Следующие фильтры в конвейере – это фильтры ресурсов. В разделе ниже мы извлечем часть общего кода из контроллера `RecipeApiController` и посмотрим, как создать фильтр прерывания выполнения.

22.1.2 Фильтры ресурсов: прерывание выполнения методов действий

Фильтры ресурсов – это первые фильтры общего назначения в конвейере фильтров MVC. В главе 21 были показаны небольшие примеры синхронных и асинхронных фильтров ресурсов, которые осуществляли журналирование в консоль. В приложениях вы можете использовать фильтры ресурсов для широкого диапазона целей, благодаря тому что они выполняются так рано (и поздно) в конвейере.

ASP.NET Core включает в себя несколько реализаций фильтров ресурсов, которые можно использовать в приложениях:

- `ConsumesAttribute` – может использоваться для ограничения разрешенных форматов, которые может принимать метод действия. Если действие декорировано атрибутом `[Consumes("application/json")]`, но клиент отправляет запрос в формате XML, фильтр ресурсов прервет выполнение конвейера и вернет ответ `415 Unsupported Media Type`;
- `SkipStatusCodePagesAttribute` – данный фильтр предотвращает запуск компонента `StatusCodePagesMiddleware` для получения ответа. Это может быть полезно, если, например, у вас есть и контроллеры

веб-API, и страницы Razor в одном приложении. Вы можете применить этот атрибут к контроллерам, чтобы гарантировать, что ответы об ошибках от API передаются без изменений, а все ответы об ошибках от Razor Pages обрабатываются промежуточным ПО.

Фильтры ресурсов полезны, когда вы хотите убедиться, что фильтр запускается на ранней стадии конвейера, перед привязкой модели. Они обеспечивают логике подключение к конвейеру на раннем этапе, поэтому при необходимости можно быстро прервать выполнение запроса.

Вернемся к листингу 22.1 и посмотрим, можно ли переделать что-либо в фильтре ресурсов. В начале методов `Get` и `Edit` есть одна строка:

```
if (!Enabled) { return BadRequest(); }
```

Она представляет собой *переключатель функциональности*, который можно использовать, чтобы отключить доступность всего API на основе поля `Enabled`. На практике вы, вероятно, загрузите это поле из базы данных или файла конфигурации, чтобы иметь возможность контролировать доступность динамически во время выполнения, но в этом примере я использую вшитое в код значение.

СОВЕТ Чтобы узнать больше об использовании переключателей функциональности в приложениях, см. мою серию статей: <http://mng.bz/2e40>.

Этот фрагмент кода представляет собой автономную сквозную логику, которая не совсем соответствует основной цели каждого метода действия, и это идеальный кандидат на роль фильтра. Вам нужно выполнить переключение функциональности на ранней стадии конвейера, до любой другой логики, поэтому фильтр ресурсов имеет смысл.

СОВЕТ Технически для этого примера также можно было бы использовать фильтр авторизации, но я следую своему совету: «Не пишите собственные фильтры авторизации!»

В следующем листинге показана реализация `FeatureEnabledAttribute`, которая извлекает логику из методов действия и перемещает ее в фильтр. Я также предоставил поле `Enabled` в качестве свойства фильтра.

Листинг 22.4 Фильтр ресурсов FeatureEnabledAttribute

```
public class FeatureEnabledAttribute : Attribute, IResourceFilter
{
    public bool Enabled { get; set; }
    public void OnResourceExecuting(
        ResourceExecutingContext context)
    {
        if (!Enabled)
        {
            context.Result = new BadRequestResult();
        }
    }
}
```

Определяет, активирована ли функциональность

Выполняется перед привязкой модели, на ранней стадии конвейера фильтров

Если функциональность не активирована, прерывает выполнение конвейера, задав свойство context.Result

```
public void OnResourceExecuted(
    ResourceExecutedContext context) { }
```

Должен быть реализован для удовлетворения **IResourceFilter**, но в данном случае он не требуется

Этот простой фильтр ресурсов демонстрирует ряд важных концепций, которые применимы к большинству типов фильтров:

- фильтр также может быть атрибутом. Им можно декорировать контроллер, методы действий и страницы Razor с помощью `[FeatureEnabled(IsEnabled = true)]`;
- интерфейс фильтра состоит из двух методов: `*Executing`, который вызывается перед привязкой модели, и `*Executed`, который вызывается после выполнения результата. Вы должны реализовать оба этих метода, даже если для вашего варианта использования вам нужен только один;
- методы выполнения фильтра предоставляют объект контекста. Он обеспечивает доступ, среди прочего, к объекту `HttpContext` для запроса и метаданные о методе действия, который будет выполнять промежуточное ПО;
- чтобы прервать выполнение конвейера, задайте для свойства `context.Result` значение экземпляра `IActionResult`. Фреймворк использует этот результат, чтобы сгенерировать ответ, игнорируя все оставшиеся фильтры в конвейере и пропуская целиком метод действия (или обработчик страницы). В этом примере, если данная функциональность не активирована, вы прервете выполнение оставшегося конвейера, вернув `BadRequestResult`, который вернет клиенту ошибку с кодом `400`.

Переместив эту логику в фильтр ресурсов, можно удалить ее из методов действий и вместо этого декорировать весь контроллер API простым атрибутом:

```
[FeatureEnabled(IsEnabled = true)]
[Route("api/recipe")]
public class RecipeApiController : ControllerBase
```

Пока что вы извлекли только две строчки кода из методов действий, но вы на правильном пути. В следующем разделе мы перейдем к фильтрам действий и извлечем еще два фильтра.

22.1.3 Фильтры действий: настройка привязки модели и результатов действий

Фильтры действий запускаются сразу после привязки модели, до выполнения метода действия. Благодаря такому расположению фильтры действий могут получить доступ ко всем аргументам, которые будут использоваться для выполнения метода действий, что делает их мощным способом извлечения общей логики действий.

Кроме того, они также запускаются сразу после выполнения метода действия и при желании могут полностью изменить или заменить `IAc-`

tionResult, возвращаемый действием. Они даже могут обрабатывать исключения, возбуждаемые в действии.

ПРИМЕЧАНИЕ Фильтры действий не выполняются для Razor Pages. Точно так же фильтры страниц не выполняются для методов действия.

ASP.NET Core включает в себя несколько стандартных фильтров действий. Один из этих часто используемых фильтров – ResponseCacheFilter, который задает HTTP-заголовки кеширования для ответов, возвращаемых методами действий.

ПРИМЕЧАНИЕ Я описал фильтры как атрибуты, но это не всегда так. Например, фильтр действий называется ResponseCacheFilter, но этот тип является внутренним для фреймворка ASP.NET Core. Чтобы применить фильтр, используется общедоступный атрибут [ResponseCache], и фреймворк автоматически настраивает ResponseCacheFilter соответствующим образом. Такое разделение атрибута и фильтра во многом является результатом внутренней конструкции, но оно может быть полезным, как показано в разделе 22.3.

Кеширование ответов и кеширование вывода

Кеширование – обширная тема, направленная на повышение производительности приложения по сравнению с примитивным подходом. Однако оно также может осложнить отладку и в некоторых ситуациях даже может быть нежелательным. Вследствие этого я часто применяю фильтр ResponseCacheFilter к методам действий, чтобы настроить HTTP-заголовки, отключающие кеширование! Об этом и других подходах к кешированию можно прочитать на странице <http://mng.bz/2eGd>.

Обратите внимание, что ResponseCacheFilter применяет заголовки управления кешием только к исходящим ответам; он не кеширует ответ на сервере. Эти заголовки сообщают клиенту (например, браузеру), может ли он пропустить отправку запроса и повторно использовать ответ. Если у вас относительно статичные конечные точки, это может значительно снизить нагрузку на приложение.

Это отличается от кеширования вывода, представленного в .NET 7. Кеширование вывода предполагает сохранение генерированного ответа на сервере и его повторное использование для последующих запросов. В простейшем случае ответ сохраняется в памяти и повторно используется для соответствующих запросов, но можно настроить ASP.NET Core для хранения вывода в другом месте, например в базе данных.

Кеширование вывода, как правило, более настраиваемо, чем кеширование ответов, поскольку вы можете точно выбрать, что кешировать и когда сделать это недействительным, но оно также требует гораздо больше ресурсов. Подробную информацию о том, как активировать кеширование вывода для конечной точки, см. в документации на странице <http://mng.bz/Bmlv>.

Истинная сила фильтров действий проявляется, когда вы создаете фильтры, адаптированные к собственным приложениям, извлекая общий код из методов действий. Чтобы продемонстрировать это, я создаю два специальных фильтра для RecipeApiController:

- `ValidateModelAttribute` – возвращает `BadRequestResult`, если состояние модели указывает на то, что модель привязки не является валидной и приведет к завершению выполнения действия. Раньше этот атрибут был основным в моих веб-API, но теперь атрибут `[ApiController]` выполняет всю эту работу (и многое другое) за вас. Тем не менее, думаю, полезно понимать, что происходит за кулисами;
- `EnsureRecipeExistsAttribute` – будет использовать аргумент каждого метода действия, `id`, чтобы убедиться, что запрашиваемая сущность `Recipe` существует до выполнения метода действия. В противном случае фильтр вернет `NotFoundResult`, и вы прервете выполнение конвейера.

Как уже было показано в главе 16, фреймворк MVC автоматически проверяет модели привязки, прежде чем выполнять действия, но вам решать, что с этим делать. В случае с контроллерами веб-API обычно возвращается ответ `400 Bad Request`, содержащий список ошибок, как показано на рис. 22.1.

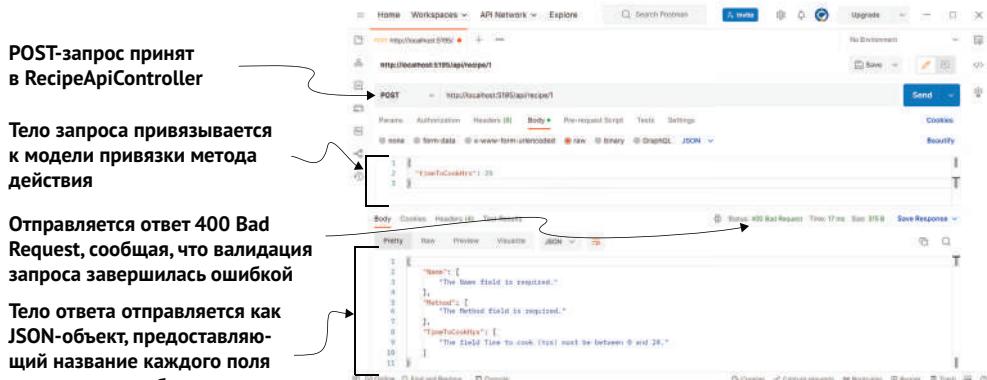


Рис. 22.1 Отправка данных в веб-API с помощью приложения Postman. Данные привязаны к модели привязки метода действия и провалидированы. Если валидация завершилась неудачно, обычно возвращается ответ `400 Bad request` со списком ошибок

Обычно следует использовать атрибут `[ApiController]` в контроллерах веб-API, что автоматически дает вам такое поведение. Но если вы не можете или не хотите использовать его, вместо этого можно создать собственный фильтр действий. В следующем листинге показана базовая реализация, аналогичная поведению, которое вы получаете при использовании атрибута `[ApiController]`.

Листинг 22.5 Фильтр действий для валидации ModelState

```
> public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
        Для удобства вы
        наследуете от
        базового класса
        ActionFilterAttribute
        if (!context.ModelState.IsValid) ←
            context.Result =
                new BadRequestObjectResult(context.ModelState);
        }
    }
}

Привязка модели и валидация
на этом этапе уже выполнены,
поэтому вы можете проверить
состояние

Переопределяет метод
Executing для запуска фильтра
до выполнения действия

Если модель невалидна, задаем свойство Result;
это приводит к прерыванию выполнения действия
```

Этот атрибут не требует пояснений и следует шаблону, аналогичному фильтру ресурсов из раздела 22.1.2, но здесь есть несколько интересных моментов:

- я наследую от абстрактного класса `ActionFilterAttribute`. Он реализует интерфейсы `IActionFilter` и `IResultFilter`, а также их асинхронные аналоги, поэтому вы можете переопределить нужные вам методы по мере необходимости. Это позволяет избежать необходимости добавлять неиспользуемый метод `OnActionExecuted()`, но использование базового класса совершенно необязательно. Это вопрос предпочтения;
- фильтры действий запускаются после привязки модели, поэтому `context.ModelState` содержит ошибки, если проверка не удалась;
- если задать для `context` свойство `Result`, это приведет к прерыванию выполнения конвейера. Но из-за расположения этапа, на котором применяется фильтр действий, только выполнение метода действий и более поздние фильтры действий будут игнорироваться; все остальные этапы конвейера работают так, словно действие выполняется как обычно.

Если вы примените этот фильтр действий к `RecipeApiController`, то можете удалить данный код из начальной части обоих методов действия, поскольку он будет выполняться автоматически в конвейере фильтров:

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

Мы будем использовать аналогичный подход для удаления повторяющегося кода, который проверяет, соответствует ли идентификатор, предоставленный в качестве аргумента для методов действия, существующей сущности `Recipe`.

В следующем листинге показан фильтр действия `EnsureRecipeExistsAttribute`. Он использует экземпляр `RecipeService`, чтобы проверить, существует ли сущность `Recipe`, и возвращает ошибку `404 Not Found`, если это не так.

Листинг 22.6 Фильтр действий для проверки существования Recipe

```

public class EnsureRecipeExistsAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
        var service = context.HttpContext.RequestServices
            .GetService<RecipeService>();
        var recipeId = (int) context.ActionArguments["id"];
        if (!service.DoesRecipeExist(recipeId))
        {
            context.Result = new NotFoundResult(); ←
        } ← Если сущности нет, возвращает результат
        } ← 404 Not Found и прерывает выполнение
    }
}

```

Получает экземпляра RecipeService из контейнера внедрения зависимостей

Извлекает параметр id, который будет передан методу действия при его выполнении

Проверяет, существует ли сущность Recipe с заданным Recipeld

Если сущности нет, возвращает результат 404 Not Found и прерывает выполнение конвейера

Как и раньше, чтобы было проще, вы наследуете от класса `ActionFilterAttribute` и переопределяете метод `OnActionExecuting`. Основная функциональность фильтра использует метод `DoesRecipeExist()` сервиса `RecipeService`, поэтому первым шагом будет получение экземпляра `RecipeService`. Параметр `context` обеспечивает доступ к объекту `HttpContext` для запроса, который, в свою очередь, позволяет получить доступ к контейнеру внедрения зависимостей и использовать `RequestServices.GetService()`, чтобы вернуть экземпляр `RecipeService`.

ВНИМАНИЕ! Данная техника получения зависимостей известна как *обнаружение сервисов* и обычно считается антипаттерном. В разделе 22.3 я покажу более приемлемый способ использования контейнера для внедрения зависимостей в фильтры.

Помимо `RecipeService`, нам понадобится дополнительная информация – это аргумент `id` методов действия `Get` и `Edit`. В фильтрах действий привязка модели уже произошла, поэтому аргументы, которые фреймворк будет использовать для выполнения метода действия, уже известны и предоставлены в `context.ActionArguments`.

Аргументы действия предоставлены как `Dictionary<string, object>`, поэтому вы можете получить параметр `id` с помощью строкового ключа "id". Не забудьте привести объект кциальному типу.

СОВЕТ Всякий раз, когда я вижу подобные волшебные строки, всегда стараюсь заменить их, используя оператор `nameof`. К сожалению, он не подходит для таких аргументов метода, поэтому будьте осторожны при рефакторинге кода. Я предлагаю явно применить фильтр действий к методу действия (а не глобально или к контроллеру), чтобы вы помнили об этой неявной связи.

Теперь, когда у нас есть `RecipeService` и `id`, нужно проверить, соответствует ли идентификатор существующей сущности `Recipe`, и если нет,

задать для `context.Result` значение `NotFoundResult`. Так вы прерываете выполнение конвейера и обходите метод действия.

ПРИМЕЧАНИЕ Помните, что у вас может быть несколько фильтров действий, которые вызываются на одном этапе. Описанное выше прерывание выполнения конвейера предотвратит выполнение фильтров, которые вызываются на более поздних этапах обработки запроса, а также позволит игнорировать выполнение метода действия.

Прежде чем мы продолжим, стоит упомянуть особый случай для фильтров действий. Базовый класс `Controller` сам реализует интерфейсы `IActionFilter` и `IAsyncActionFilter`. Если вы создаете фильтр действий для одного контроллера и хотите применять его к каждому действию в этом контроллере, то можно переопределить соответствующие методы, как показано в следующем листинге.

Листинг 22.7 Переопределение методов непосредственно для класса `ControllerBase`

```
public class HomeController : ControllerBase
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    { }
    public override void OnActionExecuted(
        ActionExecutedContext context)
    { }
}
```

← Наследует от класса ControllerBase

Выполняется перед всеми другими фильтрами действий для каждого действия в контроллере

Выполняется после всех остальных фильтров действий для каждого действия в контроллере

Если вы переопределите эти методы контроллера, они будут выполняться на этапе фильтра действий для каждого действия контроллера. Метод `OnActionExecuting` выполняется перед всеми другими фильтрами действий, независимо от порядка или области действия, а метод `OnActionExecuted` – после всех других фильтров действий.

СОВЕТ Реализация внутри контроллера может быть полезна в некоторых случаях, но с помощью нее нельзя контролировать порядок выполнения других фильтров. Лично я предпочитаю разбивать подобную логику на явные декларативные атрибуты фильтров, но, как и всегда, выбор за вами.

Теперь, когда мы закончили с фильтрами ресурсов и действий, наш контроллер выглядит намного аккуратнее, но есть один аспект, который хотелось бы устраниТЬ: обработка исключений. В следующем разделе мы рассмотрим, как создать собственный фильтр исключений для контроллера и почему у вас может возникнуть желание сделать именно это, вместо того чтобы использовать компонент обработки исключений.

22.1.4 Фильтры исключений: собственная обработка исключений для методов действий

В четвертой главе я подробно рассказывал о типах компонентов для обработки ошибок, которые вы можете добавлять в приложения. Это позволяет перехватывать исключения из любого компонента, находящегося на более позднем этапе, и обрабатывать их соответствующим образом. Если вы используете компонент обработки исключений, вам может быть интересно, для чего вообще нужны фильтры исключений.

Ответ на этот вопрос почти такой же, как я описал в разделе 21: фильтры отлично подходят для сквозных задач, когда вам нужно конкретное поведение для MVC или речь идет о применении только к определенным маршрутам.

При обработке исключений можно применять и то, и другое. Фильтры исключений являются частью фреймворка MVC, поэтому у них есть доступ к контексту, в котором произошла ошибка, как, например, действие или выполняемая страница Razor Page. Это может быть полезно при журналировании дополнительных деталей при возникновении ошибок, например параметров действия, вызвавших ошибку.

ПРЕДУПРЕЖДЕНИЕ Если вы используете фильтры исключений для записи аргументов метода действия, убедитесь, что вы не храните в журналах конфиденциальные данные, например пароли или данные кредитной карты.

Вы также можете использовать фильтры исключений для обработки ошибок из разных маршрутов разными способами. Представьте, что в вашем приложении есть и Razor Pages, и контроллеры веб-API, как в приложении рецептов. Что происходит, когда страница Razor Page возбуждает исключение?

Как вы видели в главе 4, исключение возвращается по конвейеру промежуточного ПО и перехватывается компонентом обработки исключений. Этот компонент повторно выполнит конвейер и сгенерирует страницу ошибки.

Это отлично подходит для страниц Razor Pages, а что насчет исключений в контроллерах веб-API? Если ваш API возбуждает исключение и, следовательно, возвращает HTML, сгенерированный компонентом обработки исключений, то это нарушит работу клиента, который вызвал API, ожидая ответа в формате JSON!

ПРИМЕЧАНИЕ Дополнительная сложность, связанная с необходимостью работы с этими двумя очень разными клиентами, является причиной, по которой я предпочитаю создавать отдельные приложения для API и приложения с отрисовкой на стороне сервера.

Вместо этого фильтры исключений позволяют обрабатывать исключение в конвейере фильтров и генерировать соответствующее тело ответа. Компонент обработчика исключений только перехватывает ошибки без тела, поэтому измененный ответ веб-API не затрагивается.

ПРИМЕЧАНИЕ Атрибут [ApiController] преобразует StatusCodeResult в объект ProblemDetails, но он не перехватывает исключения.

Фильтры исключений могут перехватывать исключения не только из методов действий и обработчиков страниц. Они будут запускаться, если исключение произойдет:

- во время привязки или валидации модели;
- когда выполняется метод действия или обработчик страницы;
- когда выполняется фильтр действий или фильтр страниц.

Обратите внимание, что фильтры исключений не перехватывают исключения, которые возбуждаются в каких-либо фильтрах, кроме фильтров действий и страниц, поэтому важно, чтобы фильтры ресурсов и результатов не возбуждали исключений. Также они не перехватывают исключения, возникающие при выполнении IActionResult, например при отрисовке представления Razor в HTML.

Теперь, когда вы знаете, почему вам может понадобиться фильтр исключений, пойдем дальше и реализуем один такой фильтр для RecipeApiController, как показано ниже. Это позволяет безопасно удалить блок try-catch из методов действий, при этом мы знаем, что наш фильтр будет перехватывать любые ошибки.

Листинг 22.8 Фильтр исключений HandleExceptionAttribute

ExceptionFilterAttribute – это абстрактный базовый класс, реализующий интерфейс IExceptionFilter

```
public class HandleExceptionAttribute : ExceptionFilterAttribute
```

```
{
```

```
    public override void OnException(ExceptionContext context)
```

```
{
```

Для
IExceptionFilter
можно пере-
определить
только один
метод

```
        var error = new ProblemDetails
```

```
{
```

```
            Title = "An error occurred",
            Detail = context.Exception.Message,
            Status = 500,
            Type = "https://httpwg.org/specs/rfc9110.html#status.500"
        };
    
```

Создание объекта
с подробностями
о проблеме, чтобы
вернуть его в от-
вете

```
    context.Result = new ObjectResult(error)
```

```
{
```

```
        StatusCode = 500
    };

```

```
    context.ExceptionHandled = true;
}
```

Создает ObjectResult
для сериализации
ProblemDetails и установки
кода состояния ответа

```
}
```

```
}
```

Помечает исключение как обра-
ботанное, чтобы предотвратить
его распространение в конвей-
ер промежуточного ПО

Наличие фильтра исключений в приложении – довольно распространенное явление, особенно если вы сочетаете контроллеры API и страницы Razor Pages в приложении, но это не всегда нужно. Если вы можете обрабатывать все исключения в приложении с помощью

одного компонента промежуточного ПО, то откажитесь от фильтров исключений и воспользуйтесь этим компонентом.

Мы почти закончили рефакторинг `RecipeApiController`. Осталось добавить только один тип фильтра: фильтр результатов. Специальные фильтры результатов, как правило, относительно редко встречаются в приложениях, которые писал я, но они, и вы это увидите, находят свое применение.

22.1.5 Фильтры результатов: настройка результатов действий перед их выполнением

Если в конвейере все работает успешно и вы не прерывали его выполнение, следующий этап, который идет после фильтров действий, – это фильтры результатов. Они запускаются непосредственно перед и после выполнения `IActionResult`, возвращаемого методом действия (или фильтрами действий).

ПРЕДУПРЕЖДЕНИЕ Если вы прервали выполнение конвейера из-за настройки `context.Result`, то этап, на котором вызывается фильтр результатов, не будет запущен, но `IActionResult` все равно будет выполняться для генерирования ответа. Исключениями из этого правила являются фильтры действий и страниц: они только завершают выполнение действия, как было показано в главе 21. Фильтры результатов работают как обычно, как если бы действие или обработчик страницы сами генерировали ответ.

Фильтры результатов запускаются сразу же после фильтров действий, поэтому многие из этих случаев использования похожи, но обычно фильтры результатов используются, чтобы настроить способ выполнения `IActionResult`. Например, в ASP.NET Core есть несколько фильтров результатов:

- `ProducesAttribute` – приводит к тому, что результат веб-API будет сериализован в определенный выходной формат. Например, декорируя метод действия `[Produces("application/xml")]` этим фильтром, вы заставляете форматеры попытаться отформатировать ответ в виде XML, даже если в заголовке клиента `Accept` этого формата нет;
- `FormatFilterAttribute` – декорируя метод действия этим фильтром, вы даете форматеру указание искать значение маршрута или параметр строки запроса, `format`, и использовать его для определения выходного формата. Например, можно вызвать `/api/recipe/11?format=json`, а `FormatFilter` отформатирует ответ в виде JSON, или вызовите `api/recipe/11?format=xml` – и получите ответ в формате XML.

ПРИМЕЧАНИЕ Помните, что вам необходимо явно настроить форматеры XML, если вы хотите сериализовать в XML, как описано в главе 20. Подробную информацию о форматировании результатов на основе URL-адреса см. в моей записи в блоге: <http://mng.bz/1rYV>.

Помимо управления форматерами вывода, можно использовать фильтры результатов, чтобы внести корректировки в последнюю минуту перед выполнением IActionResult и генерацией ответа.

В качестве примера доступной гибкости в следующем листинге я демонстрирую настройку заголовка LastModified на основе объекта, возвращаемого из действия. Это несколько надуманный пример – он достаточно специфичен для отдельного действия и не требует перемещения в фильтр результатов – но, надеюсь, идею вы уловили.

Листинг 22.9 Настройка заголовка ответа в фильтре результатов

```
ResultFilterAttribute предоставляет полезный базовый
класс, который можно переопределить
public class AddLastModifiedHeaderAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(
        ResultExecutingContext context)
    {
        if (context.Result is OkObjectResult result
            && result.Value is RecipeDetailViewModel detail)
        {
            var viewModelDate = detail.LastModified;
            context.HttpContext.Response
                .GetTypedHeaders().LastModified = viewModelDate;
        }
    }
}
```

Проверяет, вернул ли результат действия результат 200 Ok с моделью представления

Вы также можете переопределить метод Executed, но к тому времени ответ уже будет отправлен

Проверяет, является ли RecipeDetailViewModel типом модели представления...

...если это так, извлекает свойство LastModified и задает заголовок Last-Modified в ответе

Я использовал здесь еще один вспомогательный базовый класс, ResultFilterAttribute, поэтому только нужно переопределить один метод для реализации фильтра. Извлеките IActionResult, предоставленный в context.Result, и убедитесь, что это экземпляр OkObjectResult со значением RecipeDetailViewModel. Если это так, извлеките поле LastModified из модели представления и добавьте в ответ заголовок LastModified.

СОВЕТ GetTypedHeaders() – это метод расширения, обеспечивающий строго типизированный доступ к заголовкам запросов и ответов. Он заботится о разборе и форматировании значений. Его можно найти в пространстве имен Microsoft.AspNetCore.Http.

Как и фильтры ресурсов и действий, фильтры результатов могут реализовать метод, который выполняется *после* выполнения результата: OnResultExecuted. Вы можете использовать его, например, для проверки исключений, произошедших во время выполнения IActionResult.

ВНИМАНИЕ! Как правило, в методе OnResultExecuted ответ изменить нельзя, поскольку вы, возможно, уже начали потоковую передачу ответа клиенту.

Мы закончили работу с контроллером RecipeApiController. Путем извлечения различных фрагментов функциональности в фильтры ис-

ходный контроллер в листинге 22.1 был упрощен до версии, которую мы видим в листинге 22.2. Очевидно, что это немного утрированная и надуманная демонстрация, и я не утверждаю, что фильтры всегда должны быть для вас предпочтительным вариантом.

СОВЕТ В большинстве случаев фильтры должны использоваться в крайнем случае. Где это возможно, часто предпочтительнее применять простой закрытый метод в контроллере или поместить функциональность в предметную область. Обычно следует использовать фильтры для извлечения повторяющегося, связанного с протоколом HTTP или общего сквозного кода из контроллеров.

Есть еще один фильтр, который мы пока не рассматривали, потому что он применяется только к Razor Pages: фильтр страниц.

22.1.6 Фильтры страниц: настройка привязки модели для Razor Pages

Как уже говорилось, фильтры действий применяются только к контроллерам и действиям; они не оказывают никакого эффекта на Razor Pages, равно как на контроллеры и действия. Тем не менее фильтры страниц и фильтры действий выполняют схожие роли.

Как и в случае с фильтрами действий, ASP.NET Core включает в себя несколько уже готовых фильтров страниц. Один из них – эквивалент фильтра кеширования `ResponseCacheFilter` в Razor Page: `PageResponseCacheFilter`. Он работает так же, как и аналогичный фильтр, описанный в разделе 22.1.3, задавая заголовки кеширования HTTP для ответов на странице Razor.

Фильтры страниц несколько необычны, поскольку они реализуют три метода, как мы уже обсуждали в разделе 22.1.2. На практике я редко встречал фильтр страницы, в котором реализованы все три метода. Непривычно выполнять код сразу после выбора обработчика страницы и перед валидацией модели. Гораздо чаще фильтр страницы исполняет роль, прямо аналогичную фильтрам действий. Например, в следующем листинге показан фильтр страницы, эквивалентный фильтру действия `EnsureRecipeExistsAttribute`.

Листинг 22.10 Фильтр страниц для проверки существования сущности Recipe

Реализуем `IPageFilter` и как атрибут, чтобы можно было декорировать `PageModel` страницы Razor

```
public class PageEnsureRecipeExistsAttribute : Attribute, IPageFilter {  
    public void OnPageHandlerSelected(  
        PageHandlerSelectedContext context)  
    {}  
  
    public void OnPageHandlerExecuting(  
        PageHandlerExecutingContext context)
```

```

{
    var service = context.HttpContext.RequestServices <-- Получает
        .GetService<RecipeService>(); экземпляр
    var recipeId = (int) context.HandlerArguments["id"]; RecipeService
    if (!service.DoesRecipeExist(recipeId)) из контейнера
    {
        context.Result = new NotFoundResult(); внедрения за-
    }
}

Проверяет, существует ли сущность
Recipe с заданным Reciped

public void OnPageHandlerExecuted(
    PageHandlerExecutedContext context) Выполняется после обработ-
{}                                чика страницы (или после
}                                прерывания выполнения
}                                конвейера) – в данном при-
                                мере не используется
}

```

Извлекает параметр id, который будет передан методу обработчика страницы при его выполнении

Проверяет, существует ли сущность Recipe с заданным Reciped

Если ее не существует, возвращает результат 404 Not Found и прерывает выполнение конвейера

Фильтр страниц очень похож на фильтр действий. Самая очевидная разница состоит в необходимости реализовать три метода, чтобы соответствовать интерфейсу `IPageFilter`. Обычно нужно реализовать метод `OnPageHandlerExecuting`, который выполняется сразу после привязки и валидации модели и до выполнения обработчика страницы.

Тонкое различие между кодом фильтра действий и кодом фильтра страниц заключается в том, что фильтр действий обращается к аргументам действия, привязанным к модели, с помощью `context.ActionArguments`. В этом примере фильтр страниц использует `context.HandlerArguments`, но есть и другой вариант.

Вы помните из главы 16, что страницы Razor Pages часто привязываются к открытым свойствам `PageModel` с использованием атрибута `[BindProperty]`. Можно получить доступ к этим свойствам напрямую, вместо того чтобы использовать магические строки, путем приведения свойства `HandlerInstance` к правильному типу `PageModel` и обратившись к свойству напрямую. Например:

```
var recipeId = ((ViewRecipePageModel)context.HandlerInstance).Id
```

Так же, как класс `ControllerBase` реализует `IActionFilter`, `PageModel` реализует `IPageFilter` и `IAsyncPageFilter`. Если вы хотите создать фильтр действий для одной-единственной страницы Razor Page, то можете избавить себя от необходимости создавать отдельный фильтр страниц и переопределить эти методы прямо на странице.

СОВЕТ Обычно я считаю, что не стоит использовать фильтры страниц, если только у вас нет распространенных требований. Дополнительный уровень косвенных фильтров страниц в сочетании с типично индивидуализированной природой отдельных страниц Razor Pages означает, что обычно, по моему мнению, их не стоит использовать. Конечно же, все субъективно, но не стоит рассматривать их как первый доступный вариант.

На этом мы подошли к концу подробного рассмотрения всех фильтров в конвейере MVC. Оглядываясь назад и сравнивая листинги 22.1 и 22.2, вы видите, что фильтры позволили нам реорганизовать контроллеры и сделать цель каждого метода действия более понятной. Написание кода таким образом упрощает рассуждение о его поведении, поскольку каждый фильтр и действие имеют единственную ответственность.

В следующем разделе мы немного поговорим о том, что происходит, когда вы прерываете выполнение фильтра. Я описывал, как это сделать, задав для фильтра свойство `context.Result`, но еще не рассказывал, что именно при этом происходит. Например, что, если в этапе есть несколько фильтров, когда вы прерываете выполнение? Будут ли они по-прежнему работать?

22.2 Прерывание выполнения конвейера

В этом коротком разделе вы узнаете подробности прерывания выполнения конвейера фильтров. Вы увидите, что происходит с другими фильтрами на этапе, когда вы прерываете выполнение конвейера, и как прервать выполнение каждого типа фильтра.

Краткое предупреждение: тема прерывания выполнения фильтров может немного сбивать с толку. В отличие от компонентов промежуточного ПО, где все ясно и понятно, у конвейера фильтров есть свои нюансы. К счастью, вам нечасто придется вникать в суть всего этого, но, когда вы это сделаете, подробности вас обрадуют.

Чтобы прервать выполнение фильтров авторизации, ресурсов, действий, страниц и результатов, нужно задать для `context.Result` значение `IActionResult`. Так вы сможете игнорировать часть конвейера или все, что в нем осталось. Но конвейер фильтров не является полностью линейным, как уже было показано в главе 21, поэтому, прерывая его выполнение, не всегда удается сделать полный разворот назад. Например, фильтры действий, чье выполнение прервано, игнорируют только выполнение методов действий – фильтры результатов и этапы выполнения результатов по-прежнему будут работать.

Есть и другая трудность. Что произойдет, если у вас несколько типов фильтров? Допустим, у вас есть три фильтра ресурсов, выполняющихся в конвейере. Что будет, если второй фильтр вызывает прерывание выполнения? Все остальные фильтры игнорируются, но первый фильтр ресурсов уже выполнил свою команду `*Executing`, как показано на рис. 22.2.

Он также выполняет свою команду `*Executed` с `context.Cancelled = true`, а это указывает на то, что фильтр на данном этапе (фильтр ресурсов) прервал выполнение конвейера.

Понимание того, какие другие фильтры будут работать при прерывании выполнения одного из них, может быть довольно утомительным делом, но я собрал все фильтры в табл. 22.1. Вам также будет полезно обратиться к диаграммам конвейера из главы 21, что позволит визуализировать конвейер, когда вы будете размышлять над данной темой.

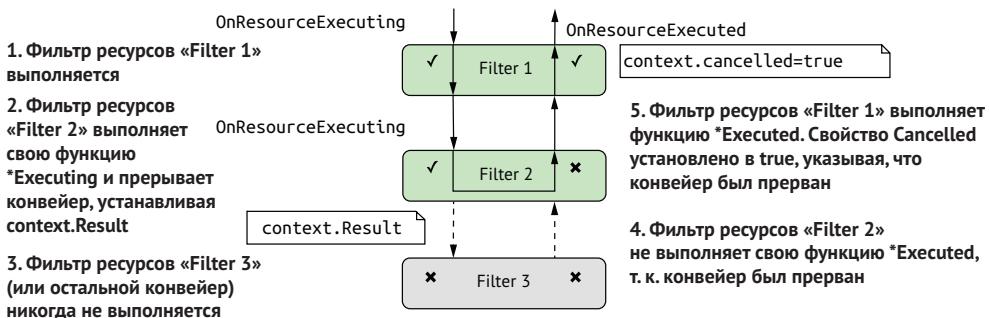


Рис. 22.2 Влияние прерывания выполнения фильтра ресурсов на другие фильтры ресурсов на этом этапе. Фильтры, которые идут после них, не запускаются, но более ранние фильтры выполняют свою функцию OnResourceExecuted

Запуск фильтров результатов после прерываний выполнения с помощью IAlwaysRunResultFilter

Фильтры результатов предназначены для обертывания выполнения объекта IActionResult, возвращаемого методом действия или фильтром действий, чтобы вы могли настроить способ выполнения результата действия.

Однако эта настройка не применяется к набору IActionResult, когда вы замыкаете конвейер фильтров, задавая context.Result в фильтре авторизации, фильтре ресурсов или фильтре исключений.

Зачастую это не проблема, поскольку многие фильтры результатов предназначены для обработки преобразований «счастливого пути». Но иногда требуется убедиться, что преобразование всегда применяется к IActionResult, независимо от того, было ли оно возвращено методом действия или фильтром замыкания.

В этих случаях вы можете реализовать IAlwaysRunResultFilter или IASyncAlwaysRunResultFilter. Эти интерфейсы расширяют стандартные интерфейсы фильтров результатов (и они идентичны им), поэтому они работают как обычные фильтры результатов в конвейере фильтров. Но эти интерфейсы отмечают, что фильтр также будет запущен, даже если фильтр авторизации, фильтр ресурсов или фильтр исключений замкнет конвейер, в то время как обычные фильтры результатов запущены не будут.

Вы можете использовать IAlwaysRunResultFilter, чтобы гарантировать, что результаты определенных действий всегда обновляются. Например, в документации показано, как использовать IAlwaysRunResultFilter для преобразования 415 StatusCodeResult в 422 StatusCodeResult независимо от источника результата действия. См. раздел «IAsyncAlwaysRunResultFilter и IASyncAlwaysRunResultFilter» в документации Microsoft «Фильтры в ASP.NET Core» на странице <http://mng.bz/IDo0>.

Таблица 22.1 Влияние прерывания выполнения фильтров на выполнение конвейера

Тип фильтра	Как прервать выполнение?	Что еще запускается?
Фильтры авторизации	Задать context.Result	Запускается только IAlwaysRunResultFilter
Фильтры ресурсов	Задать context.Result	Функции фильтров ресурсов *Executed из более ранних фильтров запускаются с context.Cancelled = true. IAlwaysRunResultFilter запускается перед выполнением IActionResult
Фильтры действий	Задать context.Result	Игнорируется только выполнение метода действия. Фильтры действий, которые идут первыми в конвейере, выполняют свои методы *Executed с context.Cancelled = true, потом методы фильтров результатов, выполнения результатов и фильтров ресурсов *Executed работают как обычно
Фильтры страниц	Задать context.Result в OnPageHandlerSelected	Игнорируется только выполнение обработчика страниц. Фильтры страниц, которые идут первыми в конвейере, выполняют свои методы *Executed с context.Cancelled = true, потом методы фильтров результатов, выполнения результатов и фильтров ресурсов *Executed работают как обычно
Фильтры исключений	Задать context.Result и Exception.Handled = true	Выполняются все функции фильтров ресурсов *Executed. IAlwaysRunResultFilter запускается перед выполнением IActionResult
Фильтры результатов	Задать context.Cancelled = true	Фильтры результатов, находящиеся первыми в конвейере, выполняют свои методы *Executed с context.Cancelled = true. Все функции фильтров ресурсов *Executed работают как обычно

Самым интересным моментом здесь является то, что прерывание выполнения фильтра действий (или фильтра страниц) не приводит к прерыванию выполнения большей части конвейера. Фактически так вы только игнорируете фильтры действий, которые идут после, и само выполнение метода действия. В первую очередь, создавая фильтры действий, вы можете гарантировать, что другие фильтры, такие как фильтры результатов, определяющие выходной формат, работают как обычно, даже когда ваши фильтры действий прерывают выполнение.

Последнее, о чем я хотел бы поговорить в этой главе, – как использовать внедрение зависимостей с фильтрами. В главах 8 и 9 показано, что внедрение зависимостей является неотъемлемой частью ASP.NET Core, а в следующем разделе вы увидите, как проектировать фильтры таким образом, чтобы фреймворк мог внедрять в них зависимости сервисов.

22.3 Использование внедрения зависимостей с атрибутами фильтра

В этом разделе вы узнаете, как внедрять сервисы в фильтры, чтобы воспользоваться преимуществами простоты внедрения зависимостей. Для этой цели вы научитесь использовать два вспомогательных фильтра, `TypeFilterAttribute` и `ServiceFilterAttribute`, и увидите, как применять их, чтобы упростить фильтр действий, который вы определили в разделе 22.1.3.

Фильтры, которые мы создавали до сих пор, были созданы как атрибуты. Это полезно для применения фильтров к методам действий и контроллерам, но означает, что вы не можете использовать внедрение зависимостей для внедрения сервисов в конструктор. Атрибуты C# не позволяют передавать зависимости в свои конструкторы (кроме постоянных значений) и создаются как объекты-одиночки, поэтому для жизненного цикла приложения существует только один экземпляр атрибута.

Что произойдет, если вам потребуется получить доступ к сервису с жизненным циклом типа `transient` или `scoped` из атрибута синглтона?

В листинге 22.11 показан один из способов сделать это, используя паттерн локатора служб, чтобы проникнуть в контейнер внедрения зависимостей и извлечь сервис `RecipeService` во время выполнения. Такой вариант сработает, но обычно это нежелательный паттерн, и предпочтение отдается внедрению зависимостей. Как добавить внедрение зависимостей в фильтры?

Ключевой момент состоит в том, чтобы разделить фильтр на две части. Вместо того чтобы создавать класс, который одновременно является атрибутом и фильтром, создайте класс фильтра, содержащий функциональность и атрибут, который сообщает фреймворку, когда и где использовать фильтр.

Применим это к фильтру действий из листинга 22.11. До этого я наследовал от класса `ActionFilterAttribute` и получил экземпляр `RecipeService` из `context`, переданного в метод. В следующем листинге я показываю два класса – `EnsureRecipeExistsFilter` и `EnsureRecipeExistsAttribute`. Класс фильтра отвечает за функциональность и принимает сервис `RecipeService` в качестве зависимости конструктора.

Листинг 22.11 Использование внедрения зависимостей в фильтре без наследования от Attribute

```
public class EnsureRecipeExistsFilter : IActionFilter <--| Не наследуется от
{                                         | класса Attribute
    private readonly RecipeService _service;
    public EnsureRecipeExistsFilter(RecipeService service)
    {
        _service = service;
    }
}
```

| RecipeService
внедряется
в конструктор

```

public void OnActionExecuting(ActionExecutingContext context)
{
    var recipeId = (int) context.ActionArguments["id"];
    if (!_service.DoesRecipeExist(recipeId))
    {
        context.Result = new NotFoundResult();
    }
}

public void OnActionExecuted(ActionExecutedContext context) { }

public class EnsureRecipeExistsAttribute : TypeFilterAttribute
{
    public EnsureRecipeExistsAttribute()
        : base(typeof(EnsureRecipeExistsFilter)) {}
}

Наследует от TypeFilter,
который используется
для заполнения зави-
симостей с помощью
контейнера внедрения
зависимостей
}
}

Вы долж-
ны реа-
лизовать
действие
Executed,
чтобы
удовлет-
ворить
инте-
фейс
В остал-
ьном
метод оста-
ется преж-
ним
}

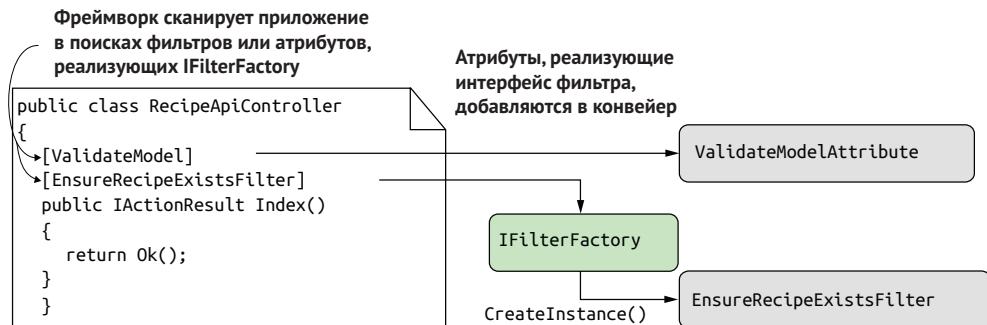
Передает тип EnsureRecipeExistsFilter в качестве аргу-
менту базовому конструктору TypeFilter в фильтре

```

`EnsureRecipeExistsFilter` – допустимый фильтр; его можно использовать отдельно, добавив в качестве глобального фильтра (поскольку глобальные фильтры не обязательно должны быть атрибутами). Но его нельзя использовать напрямую, декорируя классы контроллеров и методы действий, поскольку это не атрибут. Вот тут в действие вступает `EnsureRecipeExistsAttribute`.

Вы можете декорировать методы атрибутом `EnsureRecipeExistsAttribute`. Он наследует от `TypeFilterAttribute` и передает тип создаваемого фильтра в качестве аргумента базовому конструктору. Этот атрибут действует как *фабрика* для фильтра `EnsureRecipeExistsFilter` путем реализации интерфейса `IFilterFactory`.

Когда ASP.NET Core изначально загружает приложение, то сканирует действия и контроллеры, выискивая фильтры и фабрики фильтров, и использует их для формирования конвейера фильтров для каждого действия в приложении, как показано на рис. 22.3.



Фреймворк вызывает метод `CreateInstance()` на каждой `IFilterFactory`, когда получен запрос, чтобы создать экземпляр фильтра, который затем добавляется в конвейер

Рис. 22.3 Фреймворк сканирует приложение при запуске, чтобы найти фильтры и атрибуты, реализующие интерфейс `IFilterFactory`. Во время выполнения фреймворк вызывает метод `CreateInstance()`, чтобы получить экземпляр фильтра

Когда вызывается действие, декорированное атрибутом `EnsureRecipeExistsAttribute`, фреймворк вызывает метод `CreateInstance()` для атрибута. Он создает новый экземпляр `EnsureRecipeExistsFilter` и использует контейнер внедрения зависимостей для заполнения зависимостей (`RecipeService`).

Используя интерфейс `IFilterFactory`, вы получаете лучшее из обоих миров: вы можете декорировать контроллеры и действия атрибутами и можете использовать внедрение зависимостей в фильтрах. По умолчанию такую функциональность предоставляют два аналогичных класса, у которых немного разное поведение:

- `TypeFilterAttribute` – загружает все зависимости фильтра из контейнера внедрения зависимостей и использует их для создания нового экземпляра фильтра;
- `ServiceFilterAttribute` – загружает *сам* фильтр из контейнера. Контейнер заботится о жизненном цикле сервиса и построении графа зависимостей. К сожалению, вам также необходимо явно зарегистрировать фильтр в контейнере в `ConfigureServices` при запуске:

```
builder.services.AddTransient<EnsureRecipeExistsFilter>();
```

СОВЕТ Вы можете зарегистрировать свои сервисы с любым жизненным циклом по вашему выбору. Если ваш сервис зарегистрирован как синглтон, то можете задать флаг `IsReusable`, как описано в документации: <http://mng.bz/d1JD>.

Если вы решите использовать `ServiceFilterAttribute` вместо `TypeFilterAttribute` и зарегистрируете `EnsureRecipeExistsFilter` как сервис в контейнере внедрения зависимостей, вы можете применить `ServiceFilterAttribute` непосредственно к методу действия:

```
[ServiceFilter(typeof(EnsureRecipeExistsFilter))]
public IActionResult Index() => Ok();
```

Независимо от того, будете вы использовать `TypeFilterAttribute` или `ServiceFilterAttribute`, все это – в некоторой степени вопрос предпочтений, и при необходимости вы всегда можете реализовать собственный интерфейс `IFilterFactory`. Ключевой вывод заключается в том, что теперь вы можете использовать внедрение зависимостей в фильтрах.

Если вам не нужно использовать внедрение зависимостей для фильтра, реализуйте его напрямую как атрибут, чтобы было проще.

СОВЕТ При использовании данного паттерна мне нравится создавать фильтры как вложенный класс класса атрибутов. Это позволяет сохранить весь код в одном файле и указывает на отношения между классами.

На этом мы подошли к концу главы, посвященной конвейеру фильтров. Фильтры – немного сложная тема, поскольку они не являются строго необходимыми для создания базовых приложений, но я счи-

таю, что они чрезвычайно полезны, если нужно гарантировать, что ваши контроллеры и методы действий просты и их легко понять.

В следующей главе мы рассмотрим, как защитить приложение. Мы обсудим разницу между аутентификацией и авторизацией, понятие личности в ASP.NET Core и то, как использовать систему ASP.NET Core Identity, чтобы позволить пользователям регистрироваться и входить в приложение.

Резюме

- Конвейер фильтров выполняется как часть MVC или Razor Pages. Он состоит из фильтров авторизации, ресурсов, действий, страниц, исключений и фильтров результатов;
- ASP.NET Core включает множество встроенных фильтров, но вы также можете создавать собственные фильтры, адаптированные к вашему приложению. Вы можете использовать пользовательские фильтры для извлечения общих сквозных функций из контроллеров MVC и страниц Razor, уменьшая дублирование и обеспечивая согласованность между конечными точками;
- фильтры авторизации запускаются первыми в конвейере и управляют доступом к API. ASP.NET Core включает в себя атрибут [Authorization], который можно применить к методам действия, чтобы только выполнившие вход пользователи могли осуществить действие;
- фильтры ресурсов запускаются после фильтров авторизации и еще раз после выполнения IActionResult. Их можно использовать, чтобы прервать выполнение конвейера, дабы метод действия так и не был выполнен, а также для настройки процесса привязки модели для метода действия;
- фильтры действий запускаются после привязки модели, непосредственно перед выполнением метода действия, а также после выполнения этого метода. Их можно использовать для извлечения общего кода из метода действия, чтобы предотвратить дублирование кода. Они не выполняются для страниц Razor Pages, только для контроллеров MVC;
- базовый класс ControllerBase также реализует интерфейсы IActionFilter и IAsyncActionFilter. Они запускаются в начале и в конце конвейера фильтров, независимо от порядка или области применения других фильтров действий. Их можно использовать для создания фильтров действий, относящихся к конкретному контроллеру;
- фильтры страниц запускаются трижды: после выбора обработчика страницы, после привязки модели и после выполнения метода обработчика страницы. Вы можете использовать эти фильтры для похожих целей как фильтры действий. Фильтры страниц выполняются только для Razor Pages; они не подходят для контроллеров MVC;

- `PageModel` реализует интерфейсы `IPageFilter` и `IAsyncPageFilter`, поэтому их можно использовать для реализации фильтров конкретных страниц. Они редко используются, поскольку обычно аналогичных результатов можно добиться с помощью простых закрытых методов;
- фильтры исключений выполняются после фильтров действий и страниц, когда метод действия или обработчик страницы генерирует исключение. Их можно использовать для предоставления настраиваемой обработки ошибок для конкретного выполняемого действия;
- как правило, следует обрабатывать исключения на уровне промежуточного ПО, но вы можете использовать фильтры исключений, чтобы настроить способ обработки исключений для конкретных действий, контроллеров или страниц Razor;
- фильтры результатов выполняются до и после выполнения `IActionResult`. Вы можете использовать их для управления выполнением результата действия или для полного изменения результата действия, которое будет выполнено;
- все фильтры могут прерывать выполнение конвейера, устанавливая конкретный результат. Обычно это предотвращает дальнейшее продвижение запроса в конвейере, но точное поведение зависит от типа фильтра, в котором произошло прерывание;
- фильтры результатов не выполняются, если вы прерываете выполнение конвейера, используя фильтры авторизации, ресурсов или исключений. Для принудительного вызова фильтра результатов вы можете реализовать такие интерфейсы, как `IAlwaysRunResultFilter` или `IAsyncAlwaysRunResultFilter`;
- вы можете использовать атрибуты `ServiceFilterAttribute` и `TypeFilterAttribute`, чтобы допустить внедрение зависимостей в специальных фильтрах. `ServiceFilterAttribute` требует, чтобы вы зарегистрировали фильтр и все его зависимости в контейнере внедрения зависимостей, тогда как `TypeFilterAttribute` требует, только чтобы были зарегистрированы зависимости фильтра.

Часть IV

Защита и развертывание приложений

На данный момент вы узнали, как использовать минимальные API, Razor Pages и контроллеры MVC для создания приложений с отрисовкой на стороне сервера и API. Вы знаете, как динамически генерировать JSON и HTML-код на основе входящих запросов, а также как использовать конфигурацию и внедрение зависимостей для настройки поведения приложения во время выполнения. В этой части вы узнаете, как добавлять пользователей и профили в приложение, а также как публиковать и защищать свои приложения.

В главах с 23 по 25 вы узнаете, как защитить приложение с помощью аутентификации и авторизации. В главе 23 будет показано, как добавить ASP.NET Core Identity в свои приложения, чтобы пользователи могли выполнить вход и наслаждаться индивидуальным пользовательским опытом. В главе 24 вы узнаете, как защитить приложения Razor Pages с помощью авторизации, чтобы только некоторые пользователи могли получить доступ к определенным страницам приложения. В главе 25 вы узнаете, как применять одинаковые средства защиты к минимальным API и приложениям веб-API.

Добавление журналирования в приложение – это одно из тех действий, которое часто откладывается до тех пор, пока вы не обнаружите

жите проблему в рабочем окружении. Добавление разумного уровня журналирования с самого начала поможет вам быстро диагностировать и исправлять ошибки по мере их возникновения. В главе 26 представлена структура журналирования,строенная в ASP.NET Core. Вы увидите, как использовать ее для записи сообщений журнала в самые разные места, будь то консоль, файл или сторонняя служба удаленного журналирования.

К этому моменту у вас будут все основы для создания рабочего приложения с помощью ASP.NET Core. В главе 27 я расскажу о шагах, необходимых для того, чтобы развернуть приложение в промышленном окружении, в том числе о том, как опубликовать приложение в IIS и настроить URL-адреса, которые будет слушать приложение.

Прежде чем вы представите свое приложение миру, важной частью веб-разработки является его правильная защита. Даже если вы не считаете, что в нем есть какие-либо конфиденциальные данные, вы должны обеспечить защиту своих пользователей от атак, придерживаясь лучших практик безопасности. В главе 28 вы узнаете, как настроить протокол HTTPS для своего приложения и почему это жизненно важный шаг для современной веб-разработки. В главе 29 я описываю некоторые распространенные уязвимости безопасности, то, как злоумышленники могут ими воспользоваться и что можно сделать для защиты своих приложений.

23

Аутентификация: добавление пользователей в приложение с помощью ASP.NET Core Identity

В этой главе:

- как работает аутентификация в веб-приложениях ASP.NET Core;
- создание проекта с использованием ASP.NET Core Identity;
- добавление возможности работы с пользователями в существующее веб-приложение;
- настройка пользовательского интерфейса ASP.NET Core Identity по умолчанию.

Одним из преимуществ такого веб-фреймворка, как ASP.NET Core, является возможность предоставить динамическое приложение, настроенное для отдельных пользователей. Многие приложения имеют концепцию «учетной записи», в которую можно «войти» и получить другой пользовательский опыт.

В зависимости от приложения учетная запись дает разные возможности: в некоторых приложениях вам может потребоваться выполнить

вход, чтобы получить доступ к дополнительным функциям, а в других вы можете увидеть рекомендуемые вам статьи. В приложении для онлайн-торговли вы можете размещать заказы и просматривать свои прошлые заказы; на Stack Overflow можно публиковать вопросы и ответы, тогда как на новостном сайте вы можете получить индивидуальные рекомендации на базе статей, которые вы просматривали ранее.

Когда вы думаете над тем, как добавить пользователей в свое приложение, обычно есть два аспекта, которые следует учитывать:

- *аутентификация* – процесс создания пользователей и предоставления им возможности войти в приложение;
- *авторизация* – настройка индивидуального интерфейса пользователя и контроль над тем, что могут делать пользователи, основываясь на текущем пользователе, выполнившем вход.

В этой главе мы будем обсуждать первый из этих моментов, аутентификацию, а в следующей главе коснемся второго вопроса – авторизации. В разделе 23.1 обсуждается разница между аутентификацией и авторизацией, как работает аутентификация в традиционном веб-приложении ASP.NET Core, и способы разработки вашей системы, чтобы обеспечить возможность регистрации в ней. В этой главе я не буду подробно обсуждать API приложения, поскольку многие принципы аутентификации применимы к обоим стилям приложений. Я рассмотрю API приложения в главе 25.

В разделе 23.2 вы познакомитесь с системой управления пользователями под названием ASP.NET Core Identity (или просто Identity). Identity интегрируется с EF Core и предоставляет сервисы для создания и управления пользователями, хранения и проверки паролей, а также входа пользователей в приложение и выхода из него.

В разделе 23.3 мы создадим приложение, используя шаблон по умолчанию, который уже включает в себя ASP.NET Core Identity. Так вы увидите функции, которые предоставляет Identity, а еще все то, чего она не дает.

Создание приложения отлично подходит для того, чтобы увидеть, как его части сочетаются друг с другом, но вам часто будет необходимо добавлять пользователей и аутентификацию в существующее приложение. В разделе 23.4 вы увидите, какие шаги необходимы для добавления ASP.NET Core Identity в существующее приложение.

В разделах 23.5 и 23.6 вы узнаете, как заменить страницы из пользовательского интерфейса Identity по умолчанию путем скаффолдинга отдельных страниц. В разделе 23.5 вы увидите, как настроить шаблоны Razor для создания другой HTML-разметки на странице регистрации пользователя, а в разделе 23.6 – как настроить логику, связанную со страницей Razor. Вы увидите, как сохранить дополнительную информацию о пользователе (например, его имя или дату рождения) и как предоставить ему полномочия, которые впоследствии можно будет использовать для настройки поведения приложения (например, если это VIP-пользователь).

Прежде чем мы конкретно рассмотрим ASP.NET Core Identity, взглянем на аутентификацию и авторизацию в ASP.NET Core – что происходит, когда вы регистрируетесь на веб-сайте, и как проектировать приложения, чтобы обеспечить эту функциональность.

23.1 Знакомство с аутентификацией и авторизацией

Когда вы добавляете в приложение возможности входа и контролируете доступ к определенным функциям в зависимости от текущего пользователя, выполнившего вход, то используете два различных аспекта безопасности:

- *аутентификация* – процесс определения того, кто вы;
- *авторизация* – процесс определения того, что вам разрешено делать.

Как правило, вам нужно знать, *кто* этот пользователь, прежде чем вы сможете определить, *что* ему разрешено делать, поэтому сначала всегда идет аутентификация, а уже затем авторизация. В этой главе рассматривается только аутентификация; об авторизации речь пойдет в главе 24.

В этом разделе я начну с обсуждения того, как ASP.NET Core представляет себе пользователей, и приведу некоторые термины и концепции, имеющие ключевое значение для аутентификации. Я всегда считал, что при первом знакомстве с аутентификацией эти вещи сложнее всего понять, поэтому не буду торопиться.

Далее мы рассмотрим, что означает вход в традиционное веб-приложение. В конце концов, вы всего лишь указываете свой пароль и входите в приложение – откуда оно знает, что запрос поступил от вас, чтобы обрабатывать последующие запросы?

23.1.1 Пользователи и утверждения в ASP.NET Core

Концепция пользователя встроена в ASP.NET Core. В главе 3 мы узнали, что HTTP-сервер Kestrel создает объект `HttpContext` для каждого полученного запроса. Этот объект отвечает за хранение всех деталей, связанных с данным запросом, таких как URL-адрес запроса, все отправляемые заголовки, тело запроса и т. д.

Объект `HttpContext` также предоставляет текущего *принципала* запроса в качестве свойства `User`. Это видение ASP.NET Core относительно того, какой пользователь сделал запрос. Всякий раз, когда приложению необходимо знать, кто этот текущий пользователь или что ему позволено делать, оно смотрит на свойство `HttpContext.User`.

ОПРЕДЕЛЕНИЕ Можно рассматривать *принципала* как пользователя приложения.

В ASP.NET Core принципалы реализованы в виде объектов `ClaimsPrincipal`, у которых есть коллекция утверждений, как показано на рис. 23.1.

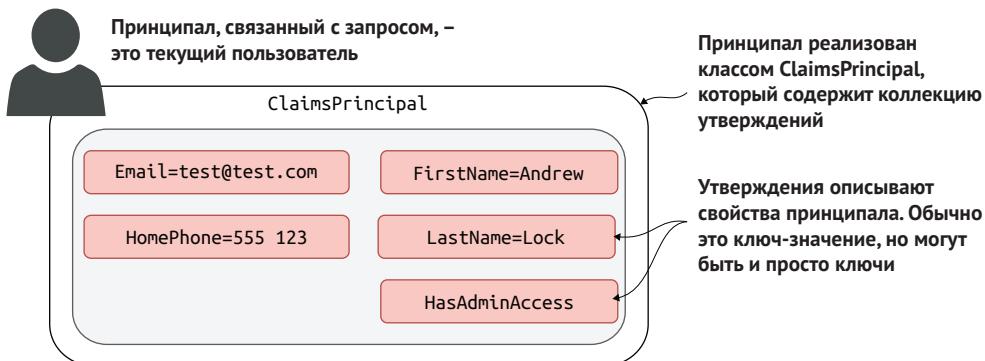


Рис. 23.1 Принципал – это текущий пользователь, реализованный как ClaimsPrincipal. Он содержит коллекцию утверждений, описывающих пользователя

Можно рассматривать утверждения как свойства текущего пользователя. Например, у вас могут быть такие свойства для адреса электронной почты, имени или даты рождения.

ОПРЕДЕЛЕНИЕ Утверждение (Claim) – это отдельная информация о принципале. Она состоит из *типа* и необязательного *значения*.

Утверждения также могут быть косвенно связаны с полномочиями и авторизацией, поэтому у вас может быть свойство HasAdminAccess или IsVipCustomer. Храниться они будут точно так же – как объекты, ассоциированные с принципалом.

ПРИМЕЧАНИЕ В более ранних версиях ASP.NET использовался ролевой подход к безопасности вместо подхода, основанного на утверждениях. Объект ClaimsPrincipal, применяемый в ASP.NET Core, совместим с этим подходом, когда речь идет об унаследованном коде, но для новых приложений следует использовать утверждения.

Kestrel назначает принципала каждому запросу, поступающему в приложение. Первоначально это универсальный, анонимный принципал без аутентификации и утверждений. Как вы выполняете вход и как ASP.NET Core узнает, что это были вы, когда последуют другие запросы?

В следующем разделе мы рассмотрим, как работает аутентификация в традиционном веб-приложении с помощью ASP.NET Core и процесса входа в учетную запись пользователя.

23.1.2 Аутентификация в ASP.NET Core: сервисы и промежуточное ПО

Добавление аутентификации в любое веб-приложение включает в себя ряд действий. Один и тот же процесс применяется независимо от того, создаете ли вы традиционное веб-приложение или клиентское приложение (хотя часто случаются различия в реализации, о чем я расскажу в главе 25).

- 1 Клиент отправляет в приложение идентификатор и секрет, которые определяют текущего пользователя. Например, можно отправить адрес электронной почты (идентификатор) и пароль (секрет).
- 2 Приложение проверяет, соответствует ли идентификатор пользователю, который известен приложению, и что соответствующий секрет правильный.
- 3 Если идентификатор и секрет являются валидными, приложение может задать принципала для текущего запроса, но ему также нужен способ хранения этих данных для последующих запросов. Что касается традиционных веб-приложений, то обычно это достигается за счет хранения зашифрованной версии принципала в файле cookie.

Это типичный процесс для большинства веб-приложений, но в данном разделе мы рассмотрим, как это работает в ASP.NET Core. В целом весь процесс точно такой же, но приятно видеть, что этот паттерн подходит для сервисов, промежуточного ПО и MVC аспектов ASP.NET Core приложения. Мы рассмотрим различные элементы типичного приложения, когда вы выполняете вход как пользователь, что это означает и как выполнять последующие запросы в качестве данного пользователя.

Вход в приложение ASP.NET Core

Когда вы впервые заходите на сайт и входите в традиционное веб-приложение, приложение отправляет вас на страницу входа и просит ввести свое имя пользователя и пароль. После отправки формы на сервер приложение перенаправляет вас на новую страницу, и вы волшебным образом авторизуетесь! На рис. 23.2 показано, что происходит за кулисами приложения ASP.NET Core при отправке формы.

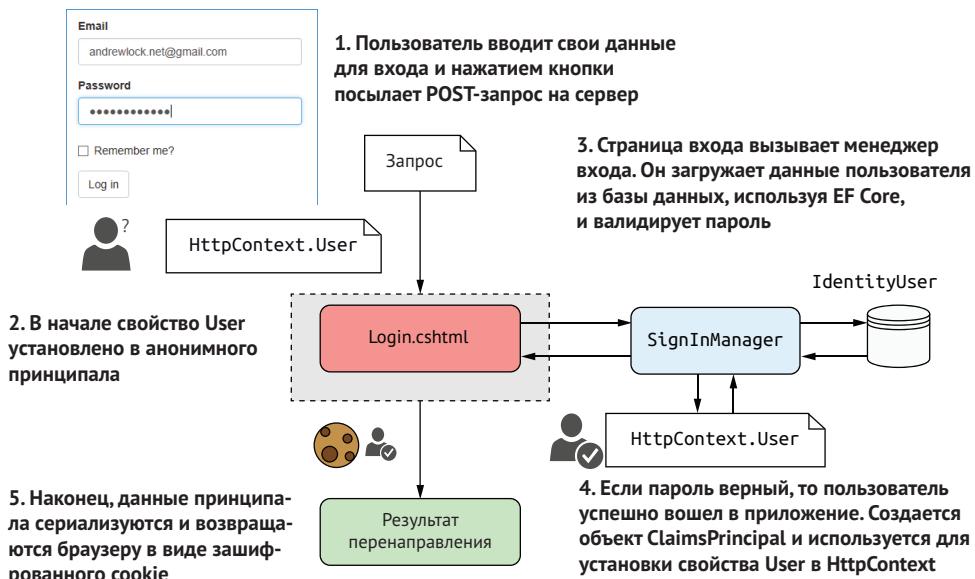


Рис. 23.2 Вход в приложение ASP.NET Core. Сервис SignInManager отвечает за настройку свойства HttpContext.User для нового принципала и сериализацию этого принципала в зашифрованный cookie

Здесь показана последовательность шагов с момента отправки формы входа на странице Razor до того момента, когда перенаправление возвращается в браузер. Когда запрос только поступает, Kestrel создает анонимного принципала и назначает его свойству `HttpContext.User`. Затем запрос маршрутизируется на страницу `Login.cshtml`, которая считывает адрес электронной почты и пароль из запроса, используя привязку модели.

Основная работа происходит внутри сервиса `SignInManager`. Он отвечает за загрузку сущности пользователя с указанным именем пользователя из базы данных и выполняет проверку, чтобы удостовериться, что предоставленный пароль верен.

ВНИМАНИЕ! Никогда не храните пароли в чистом виде в базе данных. Их нужно хешировать с использованием мощной односторонней функции. ASP.NET Core Identity делает это за вас, но всегда разумно проверить все еще раз самостоятельно!

Если пароль правильный, `SignInManager` создает новый объект `ClaimsPrincipal` из пользовательской сущности, загружаемой из базы данных, и добавляет соответствующие утверждения, например адрес электронной почты. Затем он заменяет старого анонимного принципала в `HttpContext.User` на нового аутентифицированного пользователя.

Наконец, `SignInManager` сериализует принципала, шифрует его и сохраняет в виде файла cookie. Данный файл – это небольшой фрагмент текста, который пересыпается между браузером и приложением вместе с каждым запросом. Он состоит из имени и значения.

Этот процесс аутентификации объясняет, как задать пользователя для запроса, когда он впервые входит в приложение, а как насчет последующих запросов? Вы отправляете свой пароль только при первом входе в приложение, так как же оно узнает, что запрос делает тот же пользователь?

Аутентификация пользователей для последующих запросов

Ключ к сохранению вашей личности при выполнении нескольких запросов находится на последнем этапе рис. 23.2, где вы сериализуете принципала в cookie. Браузеры автоматически отправляют этот cookie со всеми запросами приложению, поэтому не нужно указывать пароль при каждом запросе.

ASP.NET Core использует cookie аутентификации, отправляемый с запросами, чтобы восстановить объект `ClaimsPrincipal` и задать значение свойства `HttpContext.User` для запроса, как показано на рис. 23.3. Важно отметить, где этот процесс происходит, – в компоненте `AuthenticationMiddleware`.

После получения запроса, содержащего cookie аутентификации, Kestrel создает не прошедшего аутентификацию анонимного принципала и назначает его `HttpContext.User`. Любое промежуточное ПО, работающее на этом этапе, до компонента `AuthenticationMiddleware` будет считать, что запрос не прошел аутентификацию, даже при наличии допустимого cookie.

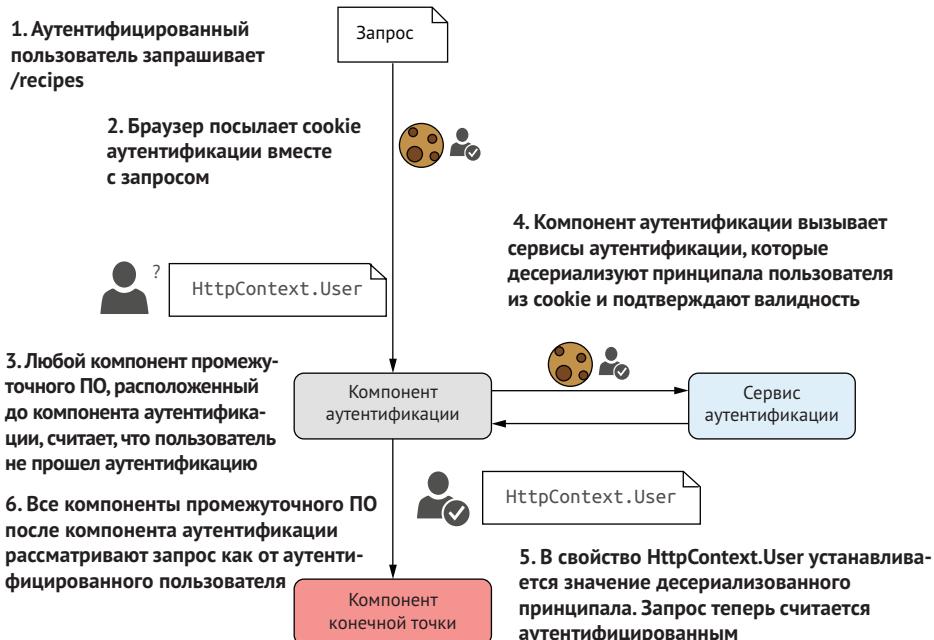


Рис. 23.3 Последующий запрос после входа в приложение. Файл cookie, отправляемый с запросом, содержит принципала, который проверяется и используется для аутентификации запроса

СОВЕТ Если вам кажется, что ваша система аутентификации не работает, проверьте еще раз свой конвейер промежуточного ПО. Только компоненты, которые выполняются после AuthenticationMiddleware, будут считать запрос аутентифицированным.

Компонент AuthenticationMiddleware отвечает за настройку текущего пользователя для запроса. Он вызывает сервисы аутентификации, которые считывают cookie из запроса, расшифровывают его и десериализуют, чтобы получить объект ClaimsPrincipal, созданный, когда пользователь выполнил вход.

AuthenticationMiddleware задает для свойства HttpContext.User нового, проверенного принципала. Все последующие компоненты теперь будут знать принципала запроса и смогут соответствующим образом скорректировать свое поведение (например, отображать имя пользователя на главной странице или ограничивать доступ к некоторым областям приложения).

ПРИМЕЧАНИЕ AuthenticationMiddleware отвечает только за аутентификацию входящих запросов и установку объекта ClaimsPrincipal, если запрос содержит cookie аутентификации. Он не несет ответственности за перенаправление запросов, не прошедших аутентификацию, на страницу входа или отклонение неавторизованных запросов – этим занимается компонент AuthorizationMiddleware, как будет показано в главе 24.

Описанный здесь процесс, при котором одно приложение аутентифицирует пользователя, когда он выполняет вход и устанавливает файл cookie, который будет считываться при последующих запросах, является обычным для традиционных веб-приложений, но это не единственная возможность. В следующем разделе мы рассмотрим аутентификацию для веб-API, используемых клиентскими приложениями или приложениями для мобильных устройств, и как меняется система аутентификации для этих сценариев.

Еще одна вещь, которую следует учитывать, – это то, где вы храните данные аутентификации пользователей своего приложения. На рис. 23.2 показано, как сервисы аутентификации загружают данные аутентификации пользователя из базы данных приложения, но это только один из вариантов.

Еще один вариант – делегировать все обязанности по аутентификации сторонним поставщикам идентификационной информации, например Facebook, Okta, Auth0 или Azure Active Directory B2B/B2C. Они управляют пользователями вместо вас, поэтому информация о пользователях и пароли хранятся в их базе данных, а не в вашей. Самое большое преимущество такого подхода состоит в том, что вам не нужно беспокоиться о безопасности данных клиентов; вы можете быть уверены, что эти поставщики защитят их, ведь это их бизнес.

СОВЕТ По возможности я рекомендую использовать этот подход, поскольку он делегирует обязанности по обеспечению безопасности третьей стороне. Вы не потеряете данные своего пользователя, если у вас их никогда не было! Однако убедитесь, что вы понимаете различия между поставщиками. В случае с Auth0 вы будете владеть созданными профилями, а с таким поставщиком, как Facebook, нет!

У каждого поставщика есть инструкции по интеграции со своими службами идентификации, в идеале с использованием спецификации OpenID Connect (OIDC). Обычно это включает в себя настройку сервисов аутентификации в вашем приложении, добавление конфигурации и делегирование самого процесса аутентификации внешнему поставщику. Эти поставщики также могут использоваться с API-приложениями, о чем я расскажу в главе 25.

ПРИМЕЧАНИЕ Подключение приложений и API для использования поставщика идентификационной информации может потребовать изрядной утомительной настройки как в приложении, так и в поставщике, но если вы будете следовать документации поставщика, у вас все будет гладко. Например, можно ознакомиться с документацией по добавлению аутентификации в традиционное веб-приложение с использованием Microsoft Identity Platform: <http://mng.bz/4D9w>.

Хотя по возможности я рекомендую использовать внешнего поставщика удостоверений, иногда вам действительно необходимо хранить

все данные аутентификации пользователей непосредственно в приложении. Именно такой подход и описан в этой главе.

ASP.NET Core Identity (иногда просто Identity) – это система, упрощающая создание системы управления пользователями в приложении. Она обрабатывает все шаблонные задачи по сохранению и загрузке пользователей в базу данных, а также использует передовые методы обеспечения безопасности, среди которых – блокировка пользователей, хеширование паролей и *многофакторная аутентификация*.

ОПРЕДЕЛЕНИЕ *Многофакторная аутентификация* (MFA) и подмножество *двухфакторной аутентификации* требуют пароля и дополнительной информации для входа. Это может быть, например, отправка кода на телефон пользователя с помощью сервиса коротких сообщений (SMS) или использование мобильного приложения для генерации кода.

В следующем разделе я расскажу о системе ASP.NET Core Identity, о задачах, которые она решает, о том, когда у вас может возникнуть желание ее использовать, а когда нет. В разделе 23.3 мы рассмотрим код и увидим ASP.NET Core Identity в действии.

23.2 Что такое ASP.NET Core Identity?

Всякий раз, когда вам нужно добавить в приложение нетривиальное поведение, обычно необходимо добавить пользователей и аутентификацию. Это означает, что вам понадобится способ сохранения сведений о ваших пользователях, среди которых их логины и пароли.

Это может показаться относительно простым требованием, но, учитывая, что это связано с безопасностью и личными данными людей, важно, чтобы вы все сделали правильно. Так же как вы сохраняете утверждения для каждого пользователя, важно хранить пароли, используя сильный алгоритм хеширования, позволять пользователям применять многофакторную аутентификацию там, где это возможно, и защищать их от атак методом полного перебора. И это лишь некоторые из множества требований. Хотя вполне возможно написать весь этот код, чтобы сделать это вручную и создать собственную систему аутентификации и членства, я настоятельно рекомендую вам не делать этого.

Я уже упоминал сторонних поставщиков идентификационной информации, таких как Auth0 или Azure Active Directory. Это решения, использующие модель Software-as-a-Service (SaaS) (программное обеспечение как услуга). Они заботятся об аспектах управления пользователями и аутентификации в приложении. Если вы переносите приложения в облако, то подобные решения могут серьезно облегчить вам жизнь.

Если вы не можете или не хотите использовать эти сторонние решения, рекомендую рассмотреть систему ASP.NET Core Identity для хранения данных пользователя и управления ими в своей базе данных. ASP.NET Core Identity берет на себя большую часть шаблонного кода, связанного с аутентификацией, оставаясь при этом гибкой и при не-

обходимости позволяя контролировать процесс выполнения входа для пользователей.

ПРИМЕЧАНИЕ ASP.NET Core Identity – это следующая версия ASP.NET Identity с улучшениями дизайна и обновленная для работы с ASP.NET Core.

По умолчанию ASP.NET Core Identity использует библиотеку EF Core для хранения сведений о пользователях в базе данных. Если вы уже используете EF Core в своем проекте, она идеально подойдет вам. В качестве альтернативы можно написать собственные хранилища для загрузки и сохранения пользовательских данных другим способом.

Identity заботится о низкоуровневых частях управления пользователями, как показано в табл. 23.1. Как видно из этого списка, Identity дает много разных возможностей, но это далеко не все!

Таблица 23.1 Какие сервисы предоставляет ASP.NET Core Identity, а какие нет

Управляется ASP.NET Core Identity	Требуется реализация со стороны разработчика
Схема базы данных для хранения пользователей и утверждений	Пользовательский интерфейс для выполнения входа, создания и управления пользователями (Razor Pages или контроллеры). Включено в необязательный пакет, который предоставляет пользовательский интерфейс по умолчанию
Создание пользователя в базе данных	Отправка сообщений электронной почты
Проверка пароля и правила для паролей	Настройка утверждений для пользователей
Блокировка учетной записи пользователя (для предотвращения атак методом полного перебора)	Настройка сторонних поставщиков идентификационной информации
Генерация кодов двухфакторной аутентификации и управление ими	Интеграция с методами многофакторной аутентификации: отправка SMS-сообщений, приложения-генераторы одноразовых паролей (TOTP) или поддержка аппаратных ключей
Генерация токенов для сброса пароля	
Сохранение дополнительных утверждений в базе данных	
Управление сторонними поставщиками идентификационной информации (например, Facebook, Google, Twitter)	

Самый большой недостаток состоит в том, что вам необходимо предоставить весь пользовательский интерфейс для приложения, а также объединить воедино все сервисы Identity для создания функционирующего процесса выполнения входа. Это довольно большой недостаток, но он делает систему Identity чрезвычайно гибкой.

К счастью, ASP.NET Core включает в себя вспомогательную библиотеку NuGet, Microsoft.AspNetCore.Identity.UI, которая дает вам

весь шаблонный код пользовательского интерфейса бесплатно. Это свыше 30 страниц Razor Pages с возможностью выполнения входа, регистрации пользователей. Здесь, среди прочего, используются двухфакторная аутентификация и внешние поставщики входа в учетную запись. При необходимости эти страницы можно настроить, но у вас уже будет готовый процесс выполнения входа и не нужно писать никакого кода, а это – существенный выигрыш. Мы рассмотрим эту библиотеку и то, как ее использовать, в разделах 23.3 и 23.4.

По данной причине я настоятельно рекомендую использовать пользовательский интерфейс по умолчанию в качестве отправной точки, независимо от того, создаете ли вы новое приложение или добавляете управление пользователями в уже существующее. Но остается один вопрос: когда следует использовать Identity, а когда нет?

Я большой поклонник Identity, поэтому склонен рекомендовать ее в большинстве ситуаций, поскольку она решает множество вопросов, связанных с безопасностью, в которых легко ошибиться. Мне приходилось слышать аргументы не в пользу Identity. Некоторые из них небезосновательны, а в отношении других можно поспорить:

- *я уже использую аутентификацию пользователей в своем приложении – отлично!* В этом случае вы, вероятно, правы, Identity может и не потребоваться. Но используется ли в вашей реализации двухфакторная аутентификация? У вас есть блокировка учетных записей? Если нет и вам нужно их добавить, тогда, вероятно, имеет смысл рассмотреть возможность использования Identity;
- *я не хочу использовать EF Core* – это разумная позиция. Вы могли бы использовать Dapper, какой-либо иной инструмент объектно-реляционного отображения или даже документоориентированную базу данных для доступа к вашим данным. К счастью, интеграция с базой данных в Identity является подключаемой, поэтому вы можете отказаться от интеграции с EF Core и использовать собственные библиотеки интеграции с базами данных;
- *мой вариант использования слишком сложен для Identity* – Identity предоставляет низкоуровневые сервисы для аутентификации, поэтому вы можете составлять различные фрагменты, как вам нравится. Кроме того, это расширяемая система, поэтому, если вам нужно, например, преобразовать утверждения перед созданием принципала, вы можете это сделать;
- *мне не нравится интерфейс Razor Pages по умолчанию* – пользовательский интерфейс по умолчанию в Identity не является обязательным. Вы по-прежнему можете использовать ее сервисы и управление пользователями и предоставить собственный интерфейс для выполнения входа и регистрации пользователей. Однако имейте в виду, что хотя это и дает большую гибкость, таким образом вы также очень легко можете создать брешь в системе безопасности управления пользователями, а это последнее место, где вам нужны такие проблемы!
- *я не использую Bootstrap для стилизации своего приложения* – пользовательский интерфейс Identity по умолчанию использует Bootstrap в качестве фреймворка для работы со стилями, как и стандарт-

ные шаблоны ASP.NET Core. К сожалению, нельзя так легко это изменить, поэтому если вы используете другой фреймворк или вам нужно настроить сгенерированный HTML, вы все равно можете использовать Identity, но вам нужно будет предоставить собственный пользовательский интерфейс;

- *я не хочу создавать собственную систему идентификации* – рад это слышать. Использование внешнего поставщика идентификационной информации, такого как Azure Active Directory B2C или Auth0, – отличный способ переложить ответственность и риски, связанные с хранением личной информации пользователей, на третью сторону.

Каждый раз, когда вы планируете добавить управление пользователями в свое приложение ASP.NET Core, я бы порекомендовал использовать Identity как отличный вариант для этого. В следующем разделе я продемонстрирую возможности Identity, создав новое приложение Razor Pages с использованием пользовательского интерфейса Identity по умолчанию. В разделе 23.4 мы возьмем этот шаблон и применим его к существующему приложению, а в разделах 23.5 и 23.6 вы увидите, как переопределить страницы по умолчанию.

23.3 Создание проекта, в котором используется ASP.NET Core Identity

Я рассказал об аутентификации и Identity в общих чертах, но лучший способ прочувствовать это в действии – увидеть работающий код. В этом разделе мы рассмотрим код по умолчанию, сгенерированный шаблонами ASP.NET Core с помощью Identity, как работает этот проект и где можно использовать Identity.

23.3.1 Создание проекта из шаблона

Мы начнем с использования шаблонов Visual Studio для генерации простого приложения Razor Pages, которое применяет Identity для хранения отдельных учетных записей пользователей в базе данных.

СОВЕТ Вы можете создать эквивалентный проект с помощью интерфейса командной строки .NET, выполнив команду `dotnet new webapp -au Individual`. Шаблон Visual Studio использует базу данных LocalDB, но шаблон `dotnet new` по умолчанию применяет SQLite. Чтобы использовать LocalDB, выполните команду `dotnet new webapp -au Individual --use-local-db`.

Чтобы создать шаблон с помощью Visual Studio, вы должны использовать VS 2022 или более позднюю версию. Кроме того, у вас должен быть установлен набор средств разработки .NET 7.0. Следуйте приведенным ниже шагам.

- 1 Выберите **File > New > Project** (Файл > Создать > Проект) или **Create a New Project** (Создать новый проект) на экране заставки.

- 2 Из списка шаблонов выберите **ASP.NET Core Web Application** (Веб-приложение ASP.NET Core), убедившись, что выбираете шаблон языка C#.
- 3 В следующем экране введите имя проекта, местоположение и имя решения и щелкните **Create** (Создать).
- 4 На экране **Additional Information** (Дополнительная информация) измените тип аутентификации на **Individual Accounts** (Отдельные учетные записи), показанный на рис. 23.4. Остальные настройки оставьте по умолчанию и выберите **Create** (Создать), чтобы создать приложение.

Visual Studio автоматически выполнит команду `dotnet restore`, чтобы восстановить все необходимые пакеты NuGet для проекта.

- 5 Запустите приложение, чтобы увидеть приложение по умолчанию, как показано на рис. 23.5.

Выберите «Отдельные учетные записи» для хранения учетных записей пользователей посредством ASP.NET Core Identity и EF Core

Выберите «Microsoft Identity Platform» для настройки приложения на использование внешнего поставщика идентификационной информации (такой как Azure Active Directory) для обработки и хранения информации о пользователях

Выберите «Вход Windows» для сайтов во внутренней сети, в которой логин в Windows предоставляет механизм аутентификации

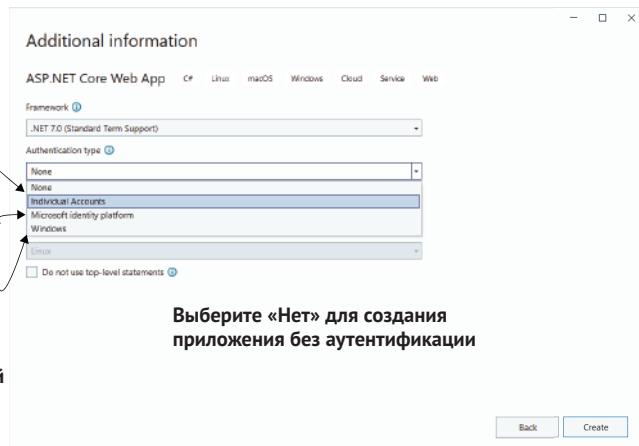


Рис. 23.4 Выбор режима аутентификации для нового шаблона приложения ASP.NET Core в VS 2022

ПРИМЕЧАНИЕ Шаблон Visual Studio настраивает приложение для использования LocalDB и включает миграцию EF Core для SQL Server. Если вы хотите использовать другого поставщика базы данных, можете заменить конфигурацию и миграцию выбранной вами базой данных, как описано в главе 12.

Этот шаблон должен быть вам знаком, но есть одна особенность: теперь у вас есть кнопки **Register** (Регистрация) и **Login** (Вход)! Не стесняйтесь экспериментировать с шаблоном – создайте пользователя, выполните вход и выйдите из приложения, чтобы прочувствовать его. Когда вы увидите, что вас все устраивает, посмотрите на код, сгенерированный шаблоном, который вам не пришлось писать благодаря этому.

СОВЕТ Не забудьте запустить включенные в шаблон миграции EF Core, прежде чем пытаться создавать пользователей. Выполните команду `dotnet ef database update` из папки проекта.

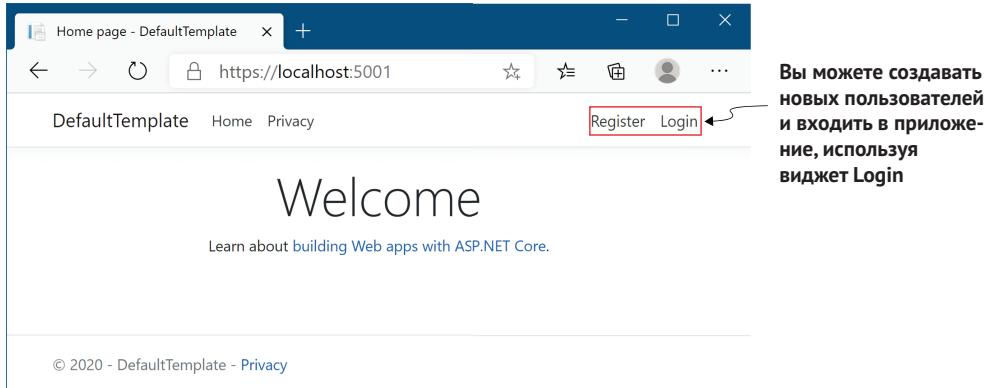


Рис. 23.5 Шаблон по умолчанию с аутентификацией отдельных учетных записей похож на шаблон без аутентификации с добавлением виджета Login в правом верхнем углу страницы

23.3.2 Изучение шаблона в Обозревателе решений

Проект, созданный с помощью шаблона и показанный на рис. 23.6, очень похож на шаблон без аутентификации по умолчанию. Во многом это связано с библиотекой пользовательского интерфейса, которая обеспечивает большую часть функциональности, не заставляя вас вникать в мелочи.

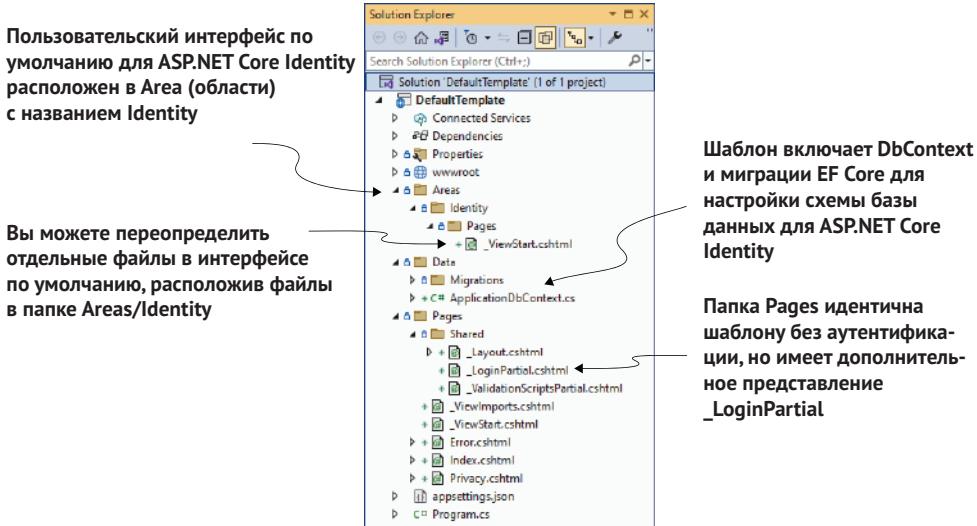


Рис. 23.6 Макет проекта шаблона по умолчанию. В зависимости от версии Visual Studio файлы могут незначительно отличаться

Самым большим дополнением является папка **Areas** (Области) в корне проекта, где содержится папка **Identity**. Иногда области используются для организации разделов функциональности. Каждая область может содержать собственную папку **Pages**, аналогичную основной папке **Pages** в приложении.

ОПРЕДЕЛЕНИЕ Области используются для группировки страниц Razor Pages в отдельные иерархии для организационных целей. Я редко использую области и предпочитаю создавать вложенные папки в основной папке Pages. Единственное исключение – пользовательский интерфейс Identity, который по умолчанию использует отдельную область Identity. Для получения дополнительных сведений об областях см.: <http://mng.bz/7Vw9>.

Пакет Microsoft.AspNetCore.Identity.UI создает Razor Pages в области Identity. Вы можете переопределить любую страницу в этом пользовательском интерфейсе по умолчанию, создав соответствующую страницу в папке Areas/Identity/Pages своего приложения. Например, как показано на рис. 23.6, шаблон по умолчанию добавляет файл _ViewStart.cshtml, переопределяющий версию, включенную как часть пользовательского интерфейса по умолчанию. Этот файл содержит следующий код, который по умолчанию использует файл _Layout.cshtml проекта для пользовательского интерфейса Identity Razor Pages:

```
@{  
    Layout = "/Pages/Shared/_Layout.cshtml";  
}
```

На данном этапе могут возникнуть некоторые очевидные вопросы: «Откуда узнать, что входит в пользовательский интерфейс по умолчанию?» и «Какие файлы можно переопределять?». Ответ на эти вопросы вы увидите в разделе 23.5, но в целом следует по возможности избегать переопределения файлов. В конце концов, цель пользовательского интерфейса по умолчанию – сократить количество кода, который вам нужно писать!

Папка Data в шаблоне нового проекта содержит класс EF Core, ApplicationDbContext, и миграции для конфигурирования схемы базы данных, чтобы использовать Identity. Подробнее об этой схеме я расскажу в разделе 23.3.3.

Последний дополнительный файл, включенный в этот шаблон по сравнению с версией без аутентификации, – это частичное представление Razor Pages/Shared/_LoginPartial.cshtml. Оно предоставляет ссылки «Регистрация» и «Вход», которые вы видели на рис. 23.5, и отображается в макете Razor по умолчанию, _Layout.cshtml.

Если вы заглянете внутрь файла _LoginPartial.cshtml, то увидите, как маршрутизация работает с областями, путем объединения пути к странице Razor с параметром маршрута {area} с помощью тег-хелперов.

Например, ссылка Login указывает на то, что /Account/Login находится в области Identity, используя атрибут asp-area:

```
<a asp-area="Identity" asp-page="/Account/Login">Login</a>
```

СОВЕТ Вы можете ссылаться на страницы Razor Pages в области Identity, задав для Identity значение маршрута area. Вы можете использовать атрибут asp-area в тег-хелперах, которые генерируют ссылки.

Помимо новых файлов, включенных благодаря ASP.NET Core Identity, стоит открыть файл Program.cs и посмотреть, что там изменилось. Наиболее очевидным изменением является дополнительная конфигурация, которая добавляет все сервисы, необходимые Identity.

Листинг 23.1 Добавление сервисов ASP.NET Core Identity в ConfigureServices

ASP.NET Core Identity использует EF Core, поэтому включает стандартную конфигурацию EF Core

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

string connectionString = builder.Configuration
    .GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationContext>(options =>
    options.UseSqlServer(connectionString));

builder.Services.AddDatabaseDeveloperPageExceptionFilter(); <-- Добавляет необязательные сервисы, чтобы улучшить страницу ошибок для разработчика

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
    options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationContext>(); <-- Настраивает Identity для хранения ее данных в EF Core

builder.Services.AddRazorPages();

// Остальная конфигурация не показана;
```

Требует от пользователя подтверждения его учетной записи (обычно по электронной почте), перед тем как они выполнят вход

Добавляет систему Identity, включая пользовательский интерфейс по умолчанию, и настраивает тип пользователя как IdentityUser

Метод расширения AddDefaultIdentity() выполняет несколько функций:

- добавляет основные сервисы ASP.NET Core Identity;
- настраивает тип пользователя приложения как IdentityUser. Это модель сущности, которая хранится в базе данных и представляет «пользователя» в вашем приложении. При необходимости можно расширить этот тип, но это не всегда необходимо, как будет показано в разделе 23.6;
- добавляет страницы Razor Pages пользовательского интерфейса по умолчанию для регистрации, входа и управления пользователями;
- настраивает поставщиков токенов для создания токенов двухфакторной аутентификации и токенов подтверждения по электронной почте.

Теперь, когда у нас есть обзор дополнений, внесенных Identity, мы подробнее рассмотрим схему базы данных и то, как Identity сохраняет пользователей в базе данных.

Где же компонент аутентификации?

Если вы уже знакомы с предыдущими версиями ASP.NET Core, то, возможно, удивитесь, заметив отсутствие компонента аутентификации в шаблоне по умолчанию. Учитывая все, что вы узнали о том, как работает аутентификация, это должно быть удивительно!

Ответ на эту загадку заключается в том, что компонент аутентификации есть в конвейере, хотя вы его и не видите. Как я уже говорил в главе 4, `WebApplication` автоматически добавляет в конвейер множество компонентов промежуточного ПО, включая компоненты маршрутизации, конечной точки и – да – компонент аутентификации. Таким образом, причина, по которой вы не видите его в конвейере, заключается в том, что он уже добавлен.

Фактически `WebApplication` также автоматически добавляет в конвейер компонент авторизации, но в этом случае шаблон все равно вызывает метод `UseAuthorization()`. Почему? По той же причине, по которой шаблон также вызывает метод `UseRouting()`: чтобы точно контролировать, где в конвейере добавляется промежуточное ПО.

Как я упоминал в главе 4, вы можете переопределить автоматически добавляемое промежуточное ПО, добавив его вручную. Крайне важно, чтобы компонент авторизации размещался после компонента маршрутизации, и, как упоминалось в главе 4, обычно нужно размещать компонент маршрутизации после компонента статических файлов. Поскольку компонент маршрутизации должен быть перемещен, то же самое необходимо сделать и с компонентом авторизации!

Традиционно компонент аутентификации также размещается после компонента маршрутизации, перед компонентом авторизации, но это не имеет решающего значения. Единственное требование состоит в том, что оно должно быть размещено перед любым промежуточным ПО, требующим аутентификации пользователя, например перед компонентом авторизации.

При желании можно изменить местоположение компонента аутентификации, вызвав метод `UseAuthentication()` в соответствующем месте. Я предпочитаю быть явным, где это возможно, поэтому обычно использую данный подход, перемещая его между вызовами методов `UseRouting()` и `UseAuthorization()`:

```
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.MapRazorPages();
app.Run();
```

Если вы не разместите компонент аутентификации в нужном месте конвейера, то можете столкнуться со странными ошибками, когда пользователи не аутентифицируются правильно или политики авторизации применяются неправильно. Шаблоны работают «из коробки», но нужно быть осторожным, если вы работаете с существующим приложением или перемещаете промежуточное ПО.

23.3.3 Модель данных ASP.NET Core Identity

Из коробки и в шаблонах по умолчанию Identity использует EF Core для хранения учетных записей пользователей. Он предоставляет базовый класс `DbContext`, от которого можно наследовать, – `IdentityDbContext`. Он использует `IdentityUser` в качестве сущности пользователя для вашего приложения.

В шаблоне класс `DbContext` называется `ApplicationDbContext`. Если вы откроете этот файл, то увидите, что он очень скучный; он наследует от базового класса `IdentityDbContext`, который я описал ранее, и все. Что дает нам этот базовый класс? Самый простой способ увидеть это – обновить базу данных, используя миграции.

Применение миграций – это тот же процесс, что и в главе 12. Убедитесь, что строка подключения указывает на то место, где вы хотите создать базу данных, откройте командную строку в папке проекта и выполните эту команду, чтобы обновить базу данных:

```
dotnet ef database update
```

СОВЕТ Если после выполнения команды `dotnet ef` вы видите ошибку, убедитесь, что установили инструмент командной строки .NET, следуя инструкциям, приведенным в разделе 12.3.1. Также убедитесь, что вы запускаете команду из папки *project*, а не из папки *solution*.

Если базы данных еще нет, интерфейс командной строки создаст ее. На рис. 23.7 показано, как выглядит база данных шаблона по умолчанию.

СОВЕТ Если вы работаете с MS SQL Server (или LocalDB), то можете использовать обозреватель объектов SQL Server в Visual Studio для просмотра таблиц и объектов в своей базе данных. Для получения подробной информации см. <http://mng.bz/mg8r>.

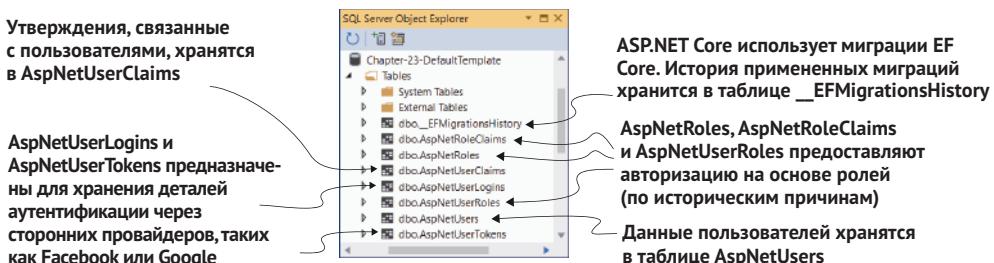


Рис. 23.7 Схема базы данных, используемая ASP.NET Core Identity

Как много таблиц! Вам не нужно напрямую взаимодействовать с этими таблицами – Identity сделает это за вас, но не помешает иметь базовое представление о том, для чего они нужны:

- `_EFMigrationsHistory` – стандартная таблица миграции EF Core, в которой записывается, какие миграции были применены;

- *AspNetUsers* – сама таблица профилей пользователей. Сюда сериализуется *IdentityUser*. Вскоре мы подробнее рассмотрим эту таблицу;
- *AspNetUserClaims* – утверждения, ассоциированные с определенным пользователем. У пользователя может быть много таких объектов, поэтому они моделируются как тип связи «многие к одному»;
- *AspNetUserLogins* и *AspNetUserTokens* – они связаны с выполнением входа в учетную запись, используя сторонние сервисы. После настройки позволяют пользователям выполнять вход с учетной записью Google или Facebook (например), вместо того чтобы создавать пароль в приложении;
- *AspNetUserRoles*, *AspNetRoles* и *AspNetRoleClaims* – эти таблицы позволяют определять роли, к которым могут принадлежать несколько пользователей. Каждой роли можно присвоить некое число утверждений. По сути, эти утверждения наследуются принципом, когда ему присваивается эта роль.

Вы сами можете изучить эти таблицы, но самая интересная из них – это *AspNetUsers*, показанная на рис. 23.8.

The screenshot shows the SQL Server Object Explorer interface. A tree view on the left lists 'dbo.AspNetUsers' under 'Tables'. Under 'Columns', a list of 15 columns is shown, each with its name, data type, and whether it is nullable (null). The columns are: Id (PK, nvarchar(450), not null), UserName (nvarchar(256), null), NormalizedUserName (nvarchar(256), null), Email (nvarchar(256), null), NormalizedEmail (nvarchar(256), null), EmailConfirmed (bit, not null), PasswordHash (nvarchar(max), null), SecurityStamp (nvarchar(max), null), ConcurrencyStamp (nvarchar(max), null), PhoneNumber (nvarchar(max), null), PhoneNumberConfirmed (bit, not null), TwoFactorEnabled (bit, not null), LockoutEnd (datetimeoffset(7), null), LockoutEnabled (bit, not null), and AccessFailedCount (int, not null).

По умолчанию ASP.NET Core Identity использует GUID для идентификатора пользователя, которые хранятся как строки в базе данных

Таблица содержит все необходимые поля для аутентификации пользователя, подтверждения почты и телефона, двухфакторной аутентификации и блокировки учетной записи

Рис. 23.8 Таблица *AspNetUsers* используется для хранения всей информации, необходимой для аутентификации пользователя

Большинство столбцов в таблице *AspNetUsers* связаны с безопасностью – адрес электронной почты пользователя, хеш пароля, подтвердил ли он свою электронную почту, активирована ли двухфакторная аутентификация и т. д. По умолчанию здесь нет столбцов для дополнительной информации, например для имени и фамилии пользователя.

ПРИМЕЧАНИЕ По рис. 23.8 видно, что идентификатор первичного ключа хранится в виде столбца типа string. По умолчанию Identity использует для идентификатора Guid. Чтобы настроить тип данных, см. раздел «Изменение типа первичного ключа» в документации Microsoft: <http://mng.bz/5jdB>.

Все дополнительные свойства пользователя хранятся в виде утверждений в таблице AspNetUserClaims, ассоциированной с этим пользователем. Это позволяет добавлять произвольную дополнительную информацию без необходимости изменять схему базы данных. Хотите сохранить дату рождения пользователя? Можете добавить утверждение к этому пользователю, и не нужно изменять схему базы данных. Вы увидите, как это работает, в разделе 23.6, когда будете добавлять объект Name для каждого нового пользователя.

ПРИМЕЧАНИЕ Часто добавление утверждений является самым простым способом расширить сущность IdentityUser по умолчанию, но вы также можете добавить дополнительные свойства в IdentityUser напрямую. Это требует изменений в базе данных, но тем не менее полезно во многих ситуациях. Вы можете прочитать, как добавить пользовательские данные, используя этот подход, здесь: <http://mng.bz/Xd61>.

Важно понимать разницу между сущностью IdentityUser (хранимой в таблице AspNetUsers) и объектом ClaimsPrincipal, который фактически представляет свойство HttpContext.User. Когда пользователь впервые выполняет вход, из базы данных загружается IdentityUser. Эта сущность сочетается с дополнительными утверждениями для пользователя из таблицы AspNetUserClaims для создания объекта ClaimsPrincipal. Именно этот объект используется для аутентификации и сериализуется в cookie аутентификации, а не IdentityUser.

Полезно иметь ментальную модель базовой схемы базы данных, которую использует Identity, но в повседневной работе вам не нужно взаимодействовать с ней напрямую – в конце концов, для этого и нужна Identity! В следующем разделе мы зайдем с другого конца и рассмотрим пользовательский интерфейс приложения и то, что вы получаете вместе с пользовательским интерфейсом по умолчанию.

23.3.4 Взаимодействие с ASP.NET Core Identity

Вам лучше самостоятельно изучить пользовательский интерфейс по умолчанию, чтобы понять, как сочетаются все элементы вместе, но в этом разделе я выделию то, что вы получаете уже из коробки, а также те области, которые обычно сразу требуют дополнительного внимания.

Точкой входа пользовательского интерфейса по умолчанию является страница регистрации пользователя приложения, как показано на рис. 23.9. Страница регистрации позволяет пользователям зарегистрироваться в приложении путем создания новой сущности IdentityUser с адресом электронной почты и паролем. После создания учетной записи пользователи перенаправляются на страницу, где указано, что они должны подтвердить адрес своей электронной почты. По умолчанию сервис электронной почты не активирован. Все зависит от того, настраиваете ли вы внешний почтовый сервис. Вы можете прочитать, как активировать отправку электронной почты в документации Microsoft «Подтверждение учетной записи и восстановление пароля в ASP.NET Core», на странице <http://mng.bz/6gBo>.



Рис. 23.9 Процесс регистрации для пользователей, применяющих пользовательский интерфейс Identity по умолчанию. Пользователи вводят адрес электронной почты и пароль и перенаправляются на страницу подтверждения адреса своей электронной почты. Это страница-заполнитель, созданная по умолчанию, но если вы активируете подтверждение по электронной почте, эта страница обновится соответствующим образом

Как только вы ее настроите, пользователи будут автоматически получать электронное письмо со ссылкой для подтверждения своей учетной записи.

По умолчанию адреса электронной почты пользователей должны быть уникальными (у вас не может быть двух пользователей с одним и тем же адресом почты), а пароль должен соответствовать разным требованиям к длине и сложности. Можно настроить эти и другие параметры в лямбда-функции конфигурации при вызове метода `AddDefaultIdentity()` в файле `Program.cs`, как показано в следующем листинге.

Листинг 23.2 Настройка параметров Identity в ConfigureServices в файле Startup.cs

Требует от пользователей подтверждения своей учетной записи по электронной почте, прежде чем они смогут выполнить вход

```
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
{
    options.SignIn.RequireConfirmedAccount = true;
    options.Lockout.AllowedForNewUsers = true;
    options.Password.RequiredLength = 12;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireDigit = false;
})
    .AddEntityFrameworkStores<AppDbContext>();
```

Активирует блокировку пользователя, чтобы предотвратить атаки методом перебора, направленные на пароли пользователей

Обновляет требования к паролю. На данный момент предписано требовать длинные пароли

После того как пользователь зарегистрировался в вашем приложении, ему необходимо выполнить вход, как показано на рис. 23.10. В правой части страницы входа шаблоны пользовательского интер-

файса по умолчанию описывают, как вы, будучи разработчиком, можете сконфигурировать внешних поставщиков входа в учетную запись, таких как Facebook и Google. Это полезная информация для вас, но это одна из причин, по которой вам может потребоваться настроить шаблоны пользовательского интерфейса по умолчанию, как будет показано в разделе 23.5.

Интерфейс пользователя по умолчанию содержит ссылки на документацию страницы входа и включения двухфакторной аутентификации

После входа вы можете перейти к страницам управления, нажав на ссылку с email в заголовке

Страницы управления позволяют пользователям обновлять email и пароль, включать двухфакторную аутентификацию и удалять учетную запись

Рис. 23.10 Вход с существующим пользователем и управление учетной записью пользователя. На странице входа описано, как настроить внешних поставщиков входа, таких как Facebook и Google. Страницы управления пользователями позволяют пользователям изменять адрес своей электронной почты и пароль и настраивать двухфакторную аутентификацию

После того как пользователь выполнил вход, он может получить доступ к страницам управления в пользовательском интерфейсе Identity. Они позволяют изменять адрес электронной почты, менять пароль, настраивать двухфакторную аутентификацию с помощью приложения-аутентификатора или удалять все свои личные данные.

Большинство этих функций работают без каких-либо усилий с вашей стороны, если вы уже настроили отправку электронной почты.

Я рассказал обо всем, что вы получаете в шаблонах пользовательского интерфейса по умолчанию. Вам может показаться, что информации мало, но здесь изложено множество требований, которые являются общими почти для всех приложений. Тем не менее есть несколько вещей, которые вы почти всегда захотите настроить:

- сконфигурируйте сервис отправки электронной почты, чтобы активировать подтверждение учетной записи и восстановление пароля, как описано в документации Microsoft «Подтверждение учетной записи и восстановление пароля в ASP.NET Core»: <http://mng.bz/vzy7>;
- добавьте генератор QR-кода для страницы активации двухфакторной аутентификации, как описано в документации Microsoft «Активация генерации QR-кода для приложений, использующих TOTP в ASP.NET Core»: <http://mng.bz/4Zmw>;

- настройте страницы регистрации и входа, чтобы удалить ссылку на документацию для активации внешних сервисов. Вы увидите, как это сделать, в разделе 23.5. В качестве альтернативы вы можете полностью отключить регистрацию пользователей, как описано здесь: <http://mng.bz/mMG>;
- соберите дополнительную информацию о пользователях на странице регистрации. Вы увидите, как это сделать, в разделе 23.6.

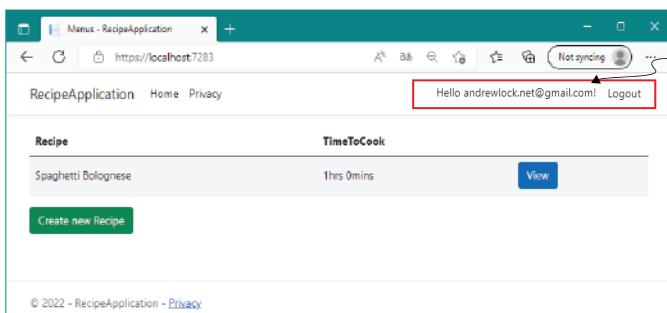
Есть еще много способов расширить или обновить систему Identity и множество доступных вариантов, поэтому я рекомендую изучить «Обзор аутентификации в ASP.NET Core» на странице <http://mng.bz/XdgV>, чтобы увидеть варианты, которые вам предлагаются. В следующем разделе вы увидите, как выполнить еще одно распространенное требование: добавить пользователей в существующее приложение.

23.4 Добавляем ASP.NET Core Identity в существующий проект

В этом разделе мы добавим пользователей в существующее приложение. Исходное приложение – это приложение Razor Pages, основанное на приложении с рецептами из главы 12. Сюда мы хотим добавить возможность работы с пользователями. В главе 24 мы продолжим работу с ним, чтобы ограничить контроль над тем, кому разрешено редактировать рецепты в приложении.

К концу этого раздела у нас будет приложение со страницей регистрации, экраном входа и экраном управления учетной записью, аналогичными шаблонам по умолчанию. У нас также будет постоянный виджет в правом верхнем углу экрана, показывающий статус входа текущего пользователя, как показано на рис. 23.11.

Как и в разделе 23.3, на данном этапе я не буду изменять какие-либо значения по умолчанию, поэтому мы не будем настраивать внешних поставщиков входа в учетную запись, подтверждение по электронной почте или двухфакторную аутентификацию. Моя задача – только добавить ASP.NET Core Identity в существующее приложение, которое уже использует EF Core.



Виджет Login отображает текущего вошедшего пользователя (email) и ссылку для выхода из приложения

Рис. 23.11 Приложение рецептов после добавления аутентификации с виджетом Login

СОВЕТ Прежде чем приступить к добавлению Identity в существующий проект, стоит убедиться, что вы знакомы с новыми шаблонами проектов. Создайте тестовое приложение и рассмотрите возможность настройки внешнего поставщика входа в учетную запись, настройки подтверждения адреса электронной почты и активации двухфакторной аутентификации. Это займет немного времени, но будет бесценным опытом для расшифровки ошибок при добавлении Identity в существующие приложения.

Чтобы добавить Identity в приложение, необходимо сделать следующее:

- 1 добавьте пакеты NuGet для ASP.NET Core Identity;
- 2 добавьте необходимые сервисы Identity в контейнер внедрения зависимостей;
- 3 обновите модель данных EF Core, добавив сущности Identity;
- 4 обновите страницы Razor Pages и макеты, чтобы предоставить ссылки на пользовательский интерфейс Identity.

В этом разделе мы рассмотрим каждый из этих шагов по очереди. В конце раздела 23.4 вы успешно добавите учетные записи пользователей в приложение с рецептами.

23.4.1 Настройка сервисов ASP.NET Core Identity и промежуточного ПО

Можно добавить ASP.NET Core Identity с пользовательским интерфейсом по умолчанию в существующее приложение, добавив ссылки на два пакета NuGet:

- *Microsoft.AspNetCore.Identity.EntityFrameworkCore* – предоставляет все основные сервисы Identity и интеграцию с EF Core;
- *Microsoft.AspNetCore.Identity.UI* – предоставляет страницы Razor Pages пользовательского интерфейса по умолчанию.

Обновите файл проекта .csproj, включив в него эти два пакета:

```
<PackageReference  
    Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore"  
    Version="7.0.0" />  
<PackageReference  
    Include="Microsoft.AspNetCore.Identity.UI" Version="7.0.0" />
```

Они содержат все дополнительные необходимые зависимости, чтобы добавить Identity с пользовательским интерфейсом по умолчанию. Обязательно выполните команду `dotnet restore` после добавления их в свой проект.

После добавления пакетов можно обновить файл Program.cs, чтобы включить сервисы Identity, как показано ниже. Это похоже на настройку шаблона по умолчанию, которая была показана в листинге 23.1, но обязательно сохраните ссылку на существующий `AppDbContext`.

Листинг 23.3 Добавление сервисов ASP.NET Core Identity в приложение с рецептами

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlite(builder.Configuration
        .GetConnectionString("DefaultConnection")!));
```

Существующая конфигурация сервиса не изменилась

```
builder.Services.AddDefaultIdentity<ApplicationUser>(options =>
    options.SignIn.RequireConfirmedAccount = true)
```

```
    .AddEntityFrameworkStores<AppDbContext>();
```

Добавляет сервисы Identity в контейнер внедрения зависимостей и использует специальный тип пользователя, ApplicationUser

```
builder.Services.AddRazorPages();
builder.Services.AddScoped<RecipeService>();
```

Убеждаемся, что используется имя ApplicationUser

Так мы добавляем все необходимые сервисы и настраиваем Identity для использования EF Core. Я представил здесь новый тип `ApplicationUser`, который мы будем использовать для настройки нашей пользовательской сущности позже. Вы увидите, как добавить этот тип, в разделе 23.4.2.

Следующий шаг не является обязательным: добавьте `AuthenticationMiddleware` после вызова метода `UseRouting()` в `WebApplication`, как показано в следующем листинге. Как я упоминал ранее, `WebApplication` автоматически добавляет компонент аутентификации, поэтому данный шаг не является обязательным.

Листинг 23.4 Добавляем компонент AuthenticationMiddleware в приложение рецептов

```
app.UseStaticFiles();
```

StaticFileMiddleware никогда не будет воспринимать запросы как аутентифицированные, даже после того, как вы выполните вход

```
app.UseRouting();
```

Добавляем AuthenticationMiddleware после метода UseRouting() и до метода UseAuthorization

```
app.UseAuthentication();
app.UseAuthorization();
```

Промежуточное ПО, которое идет после компонента AuthenticationMiddleware, может читать принципала из `HttpContext.User`

```
app.MapRazorPages();
app.Run
```

Мы настроили приложение для использования Identity, поэтому следующим шагом будет обновление модели данных EF Core. Мы уже используем EF Core в этом приложении, поэтому нам необходимо обновить свою схему базы данных для включения таблиц, которые требуются Identity.

23.4.2 Обновление модели данных EF Core для поддержки Identity

Код, приведенный в листинге 23.3, не компилируется, поскольку ссылается на тип `ApplicationUser`, которого еще не существует. Создайте его в папке `Data`, используя следующую строку:

```
public class ApplicationUser : IdentityUser { }
```

В данном случае не обязательно создавать специальный пользовательский тип (например, шаблоны по умолчанию используют `IdentityUser`), но я считаю, что проще добавить производный тип на раннем этапе, а не пытаться модернизировать его позже, если вам нужно будет добавить дополнительные свойства в пользовательский тип.

В разделе 23.3.3 показано, что `Identity` предоставляет класс `DbContext` с именем `IdentityDbContext`, от которого можно наследовать. Базовый класс `IdentityDbContext` включает в себя необходимый тип `DbSet<T>` для хранения пользовательских сущностей с помощью EF Core.

Обновить существующий класс `DbContext` для `Identity` просто: обновите класс `DbContext` своего приложения, добавив наследование от `IdentityDbContext`, как показано в следующем листинге. В данном случае мы используем обобщенную версию базового контекста `Identity` с использованием типа `ApplicationUser`.

Листинг 23.5 Обновляем класс `AppDbContext`, чтобы использовать `IdentityDbContext`

```
Обновляем для наследования из контекста
Identity, а не напрямую из DbContext
public class AppDbContext : IdentityDbContext<ApplicationUser> {
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {
        public DbSet<Recipe> Recipes { get; set; }
    }
}
```

Остальная часть
класса остается
прежней

Фактически, обновив таким образом базовый класс контекста, мы добавили всю загрузку новых сущностей в модель данных EF Core. Как было показано в главе 12, всякий раз, когда модель данных EF Core изменяется, необходимо создавать новую миграцию и применять эти изменения к базе данных.

На данном этапе приложение должно скомпилироваться, поэтому можно добавить новую миграцию, `AddIdentitySchema`:

```
dotnet ef migrations add AddIdentitySchema
```

Последний шаг – обновить страницы Razor Pages и макеты приложения, чтобы они ссылались на пользовательский интерфейс `Identity` по умолчанию. Обычно добавление 30 новых страниц Razor Pages в приложение – довольно трудоемкий процесс, но использование пользовательского интерфейса `Identity` по умолчанию делает его проще некуда.

23.4.3 Обновление представлений Razor для связи с пользовательским интерфейсом `Identity`

Технически не нужно обновлять страницы Razor, чтобы ссылаться на страницы, включенные в пользовательский интерфейс по умолчанию,

но вы, как минимум, вероятно, захотите добавить виджет Login в свой макет приложения, а также убедиться, что ваши страницы используют тот же базовый файл Layout.cshtml, что и остальная часть приложения.

Мы начнем с исправления макета страниц Identity. Создайте файл по «волшебному» пути Areas/Identity/Pages/_ViewStart.cshtml и добавьте туда следующее содержимое:

```
@{ Layout = "/Pages/Shared/_Layout.cshtml"; }
```

Таким образом, макет вашего приложения по умолчанию становится макетом по умолчанию для страниц Identity. Затем добавьте файл _LoginPartial.cshtml в Pages/Shared, чтобы определить виджет Login, как показано в следующем листинге. Этот код в значительной степени идентичен шаблону, сгенерированному по умолчанию, но здесь мы используем наш пользовательский ApplicationUser вместо стандартного IdentityUser.

Листинг 23.6 Добавляем файл _LoginPartial.cshtml в существующее приложение

```
using Microsoft.AspNetCore.Identity  
using RecipeApplication.Data; ← Обновляем пространство имен нашего проекта, которое содержит ApplicationUser  
@inject SignInManager<ApplicationUser> SignInManager |  
@inject UserManager<ApplicationUser> UserManager |  
  
<ul class="navbar-nav">  
@if (SignInManager.IsSignedIn(User)) {  
    <li class="nav-item">  
        <a class="nav-link text-dark" asp-area="Identity"  
            asp-page="/Account/Manage/Index" title="Manage">  
            Hello @User.Identity.Name!</a>  
    </li>  
    <li class="nav-item">  
        <form class="form-inline" asp-page="/Account/Logout"  
            asp-route-returnUrl="@Url.Page("/", new { area = "" })"  
            asp-area="Identity" method="post" >  
            <button class="nav-link btn-link text-dark"  
                type="submit">Logout</button>  
        </form>  
    </li>  
}  
else {  
    <li class="nav-item">  
        <a class="nav-link text-dark" asp-area="Identity"  
            asp-page="/Account/Register">Register</a>  
    </li>  
    <li class="nav-item">  
        <a class="nav-link text-dark" asp-area="Identity"  
            asp-page="/Account/Login">Login</a>  
    </li>  
}  
</ul>
```

В шаблоне по умолчанию используется IdentityUser. Выполним обновление, чтобы использовать ApplicationUser

Этот фрагмент показывает текущий статус входа пользователя и предоставляет ссылки для регистрации или входа в учетную запись.

Все, что осталось сделать, – визуализировать частичное представление, вызвав

```
<partial name="_LoginPartial" />
```

в основном файле макета нашего приложения `_Layout.cshtml`.

Вот и все: вы добавили Identity в существующее приложение. Пользовательский интерфейс по умолчанию делает этот процесс относительно простым, и вы можете быть уверены, что не создали никаких брешей в системе безопасности, создавая собственный пользовательский интерфейс!

Как я писал в разделе 23.3.4, есть некоторые функции, недоступные в пользовательском интерфейсе по умолчанию. Их нужно реализовать самостоятельно, например подтверждение по электронной почте и генерация QR-кода для двухфакторной аутентификации. Также часто бывает, что вам нужно изменить что-то в каких-то местах. В следующем разделе я покажу, как заменить страницу в пользовательском интерфейсе по умолчанию, без необходимости самостоятельно перестраивать весь пользовательский интерфейс.

23.5 Настройка страницы по умолчанию в пользовательском интерфейсе ASP.NET *Core Identity*

В этом разделе вы узнаете, как использовать скаффолдинг для замены отдельных страниц по умолчанию в пользовательском интерфейсе Identity. Вы научитесь заменять страницу таким образом, чтобы она переопределяла пользовательский интерфейс по умолчанию, позволяя настраивать и шаблон Razor, и обработчики страниц `PageModel`.

Теоретически, когда Identity предоставляет весь пользовательский интерфейс целиком для вашего приложения, это здорово, но на практике появляются шероховатости, как было показано в разделе 23.3.4. По умолчанию пользовательский интерфейс предоставляет столько, сколько может, но есть некоторые вещи, которые вы, возможно, захотите отрегулировать. Например, на страницах входа и регистрации описано, как настроить внешних поставщиков входа в учетную запись для приложений ASP.NET Core, как вы видели на рис. 23.12 и 23.13. Это полезная информация для вас как для разработчика, но это не то, что вы хотите показывать своим пользователям. Еще одно часто упоминаемое требование – это желание изменить внешний вид и поведение одной или нескольких страниц.

К счастью, пользовательский интерфейс Identity по умолчанию разработан с возможностью поэтапной замены, поэтому вы можете переопределить одну страницу без необходимости самостоятельно перестраивать весь пользовательский интерфейс. Кроме того, и в Visual Studio, и в интерфейсе командной строки .NET есть функции, позво-

ляющие заменять любую (или все) страницу в пользовательском интерфейсе по умолчанию, чтобы не приходилось начинать все сначала, когда вы хотите настроить страницу.

ОПРЕДЕЛЕНИЕ Скаффолдинг – это процесс генерации файлов в проекте, которые служат основой для настройки. В Identity существует для этого специальный инструмент, который добавляет страницы Razor Pages в нужные места, чтобы они переопределяли эквивалентные страницы пользовательского интерфейса по умолчанию. Первоначально код в таких страницах совпадает с кодом пользовательского интерфейса Identity по умолчанию, но его можно настроить.

В качестве примера изменений, которые легко можно внести, мы изменим страницу регистрации и удалим раздел дополнительной информации о внешних поставщиках. Следующие шаги описывают, как изменить страницу Register.cshtml в Visual Studio.

- 1 Добавьте NuGet-пакеты Microsoft.VisualStudio.Web.CodeGeneration.Design и Microsoft.EntityFrameworkCore.Tools в файл проекта, если они еще не добавлены. Visual Studio использует их, чтобы правильно добавить заготовки в ваше приложение, и без них вы можете получить ошибку при запуске инструмента скаффолдинга:

```
<PackageReference Version="7.0.0"
    Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" />
<PackageReference Version="7.0.0"
    Include="Microsoft.EntityFrameworkCore.Tools" />
```

- 2 Убедитесь, что ваш проект собирается, – если это не так, инструмент скаффолдинга прекратит работу, прежде чем будут добавлены новые страницы.
- 3 Щелкните по проекту правой кнопкой мыши и выберите **Add > New Scaffolded Item**.
- 4 В диалоговом окне выбора выберите **Identity** из категории и нажмите **Add**.
- 5 В диалоговом окне **Add Identity** выберите страницу **Account/Register** и выберите **AppDbContext** приложения в качестве класса Data-Context, как показано на рис. 23.12. Щелкните **Add**, чтобы изменить страницу.

СОВЕТ Для скаффолдинга страницы регистрации с помощью .NET CLI установите необходимые инструменты и пакеты, как описано в документации Microsoft «Scaffold Identity в проектах ASP.NET Core»: <http://mng.bz/QPRv>. Затем выполните команду `dotnet aspnet-codegenerator identity -dc RecipeApplication.Data.AppDbContext --files "Account.Register"`.

Visual Studio соберет ваше приложение, а затем сгенерирует страницу Register.cshtml за вас, поместив ее в папку Areas/Identity/Pages/Account.

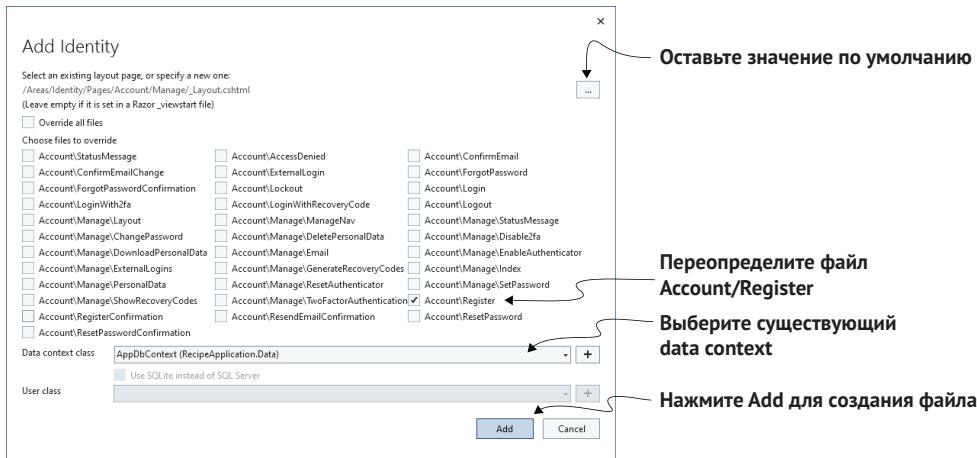


Рис. 23.12 Использование Visual Studio для изменения страницы Identity. Сгенерированные страницы Razor Pages будут переопределять версии, предоставляемые пользовательским интерфейсом по умолчанию

Он также сгенерирует несколько вспомогательных файлов, как показано на рис. 23.13. В основном они требуются для того, чтобы гарантировать, что ваша новая страница Register.cshtml может ссылаться на остальные страницы в пользовательском интерфейсе Identity по умолчанию.

Нас интересует страница Register.cshtml, поскольку мы хотим настроить пользовательский интерфейс на странице регистрации, но если вы заглянете внутрь страницы, где находится код программной части, Register.cshtml.cs, то увидите, сколько сложностей скрывает от вас пользовательский интерфейс Identity по умолчанию. Нельзя сказать, что они непреодолимы (мы настроим обработчик страницы в разделе 23.6), но всегда полезно избегать самостоятельного написания кода, если у вас есть средства, которые могут сделать это за вас.

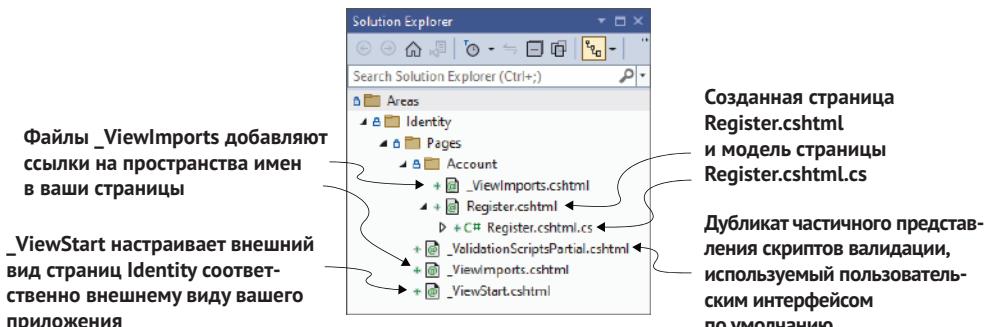


Рис. 23.13 Использование Visual Studio для изменения страницы Identity. Сгенерированные страницы Razor Pages будут переопределять версии, предоставляемые пользовательским интерфейсом по умолчанию

Теперь, когда у вас есть шаблон Razor в приложении, можно настроить его по своему усмотрению. Обратной стороной является то, что теперь вы сопровождаете больше кода, по сравнению с тем, что у вас было, когда вы использовали пользовательский интерфейс по умолчанию. Вам не пришлось его писать, но, возможно, вам все равно придется *обновить* его при выходе новой версии ASP.NET Core.

Мне нравится использовать небольшой трюк, когда дело доходит до подобного переопределения пользовательского интерфейса Identity по умолчанию. Во многих случаях на самом деле не нужно менять *обработчики страниц* для страниц Razor, а только *представление*. Этого можно добиться, удалив файл модели страницы Register.cshtml.cs и указав новому файлу с расширением .cshtml на *исходную* модель страницы, которая является частью пакета NuGet пользовательского интерфейса по умолчанию.

Другое преимущество этого подхода заключается в том, что вы можете удалить и другие файлы, которые были заменены автоматически.

В целом можно внести следующие изменения:

- обновите директиву @model в файле Register.cshtml, чтобы она указывала на PageModel пользовательского интерфейса по умолчанию.

```
@model  
↳ Microsoft.AspNetCore.Identity.UI.V5.Pages.Account.Internal.RegisterModel  
  
■ обновите файл Areas/Identity/Pages/_ViewImports.cshtml:
```

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelper
```

- удалите файл Areas/Identity/Pages/_ValidationScriptsPartial.cshtml;
- удалите файл Areas/Identity/Pages/Account/Register.cshtml.cs;
- удалите файл Areas/Identity/Pages/Account/_ViewImports.cshtml.

После внесения всех этих изменений вы получите лучшее из обоих миров – вы сможете обновить HTML-код страниц пользовательского интерфейса по умолчанию, не беря на себя ответственность за сопровождение кода программной части пользовательского интерфейса по умолчанию.

СОВЕТ В исходном коде для этой книги можно увидеть эти изменения в действии, где представление страницы регистрации было настроено таким образом, чтобы удалить ссылки на внешние поставщики идентификационной информации.

К сожалению, не всегда можно использовать модель страницы пользовательского интерфейса по умолчанию. Иногда *необходимо* обновить обработчики страниц, например если вы хотите изменить функциональность области Identity, а не просто внешний вид. Существует распространенное требование, согласно которому необходимо сохранять дополнительную информацию о пользователе, как будет показано в следующем разделе.

23.6 Управление пользователями: добавление специальных данных для пользователей

В этом разделе вы увидите, как настроить объект `ClaimsPrincipal`, назначенный вашим пользователям, добавив дополнительные утверждения в таблицу `AspNetUserClaims` при создании пользователя. Вы также увидите, как получить доступ к этим объектам на страницах Razor Pages и шаблонах.

Очень часто следующим шагом после добавления Identity в приложение является его настройка.

Шаблоны по умолчанию требуют для регистрации только адрес электронной почты и пароль. Что, если вам нужна дополнительная информация, например понятное имя для пользователя? Кроме того, я упоминал, что мы используем утверждения для безопасности, а что, если вы хотите добавить утверждение с именем `IsAdmin`, определенным пользователем?

Вы знаете, что у каждого принципала есть набор утверждений, поэтому концептуально для добавления любого такого утверждения требуется лишь добавить его в коллекцию пользователя. Есть два основных случая, когда нужно предоставить пользователю утверждение:

- *каждому пользователю при первой регистрации в приложении.* Например, вы можете добавить поле «Имя» в форме регистрации и добавить его в качестве утверждения для пользователя, когда он регистрируется;
- *вручную, после того как пользователь уже зарегистрировался.* Это обычное дело для утверждений, используемых в качестве полномочий, когда существующий пользователь может захотеть добавить утверждение `IsAdmin` конкретному пользователю после регистрации в приложении.

В этом разделе я покажу вам первый подход, добавляя новые утверждения к пользователю при их создании. Последний подход более гибкий, и в конечном итоге он потребуется многим приложениям, особенно бизнес-приложениям. К счастью, в нем нет ничего концептуально сложного; для этого требуется простой пользовательский интерфейс, который позволяет просматривать пользователей и добавлять утверждения через тот же механизм, который я покажу здесь.

СОВЕТ Еще один распространенный подход – настроить сущность `IdentityUser`, добавив, например, свойство `Name`. С таким подходом иногда проще работать, если вы хотите предоставить пользователям возможность редактировать это свойство. В статье Microsoft «Добавление, загрузка и удаление специальных пользовательских данных в Identity в проекте ASP.NET Core» описаны шаги, необходимые для этого: <http://mng.bz/aoe7>.

Допустим, вы хотите добавить пользователю новое утверждение `FullName`. Типичный подход будет выглядеть так.

- 1 Выполнить скафолдинг страницы `Register.cshtml`, как мы делали в разделе 23.5.

- 2 Добавить поле «Имя» в InputModel в модель страницы PageModel файла Register.cshtml.cs.
- 3 Добавить поле ввода «Имя» в шаблон представления Razor, Register.cshtml.
- 4 Создать новую сущность ApplicationUser, как и раньше, в обработчике страницы OnPost() путем вызова метода CreateAsync для UserManager<ApplicationUser>.
- 5 Добавить пользователю новое утверждение, вызвав метод UserManager.AddClaimAsync().
- 6 Продолжить выполнение метода, как и раньше, отправив электронное письмо с подтверждением или выполнив вход, если подтверждение по электронной почте не требуется.

Шаги с 1 по 3 говорят сами за себя и требуют лишь добавления в существующие шаблоны нового поля. Шаги с 4 по 6 выполняются в файле Register.cshtml.cs в обработчике страницы OnPost(), который кратко описан в следующем листинге. На практике у обработчика страниц больше проверок ошибок и шаблонного кода; мы сосредоточимся на дополнительных строках, которые добавляют дополнительное утверждение в ApplicationUser; полный код вы можете найти в примере кода для этой главы.

Листинг 23.7 Добавляем специальное утверждение к новому пользователю на странице Register.cshtml.cs

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser {
            UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(
            user, Input.Password);
        if (result.Succeeded)
        {
            var claim = new Claim("FullName", Input.Name); ← Проверяет
            await _userManager.AddClaimAsync(user, claim); правильность
            var code = await _userManager → предоставлена-
                .GenerateEmailConfirmationTokenAsync(user); ного пароля и
            await _emailSender.SendEmailAsync( создает поль-
                Input.Email, "Confirm your email", code ); зователя в
            await _signInManager.SignInAsync(user); базе данных
            return LocalRedirect(returnUrl); → Создает ут-
        }
        foreach (var error in result.Errors) верждение
        {
            ModelState.AddModelError(
                string.Empty, error.Description); со строко-
        }
    }
    return Page(); вым именем
}
```

Создает экземпляр сущности ApplicationUser

Добавляет новое утверждение в коллекцию ApplicationUser

Отправляет пользователю электронное письмо с подтверждением, если вы настроили отправителя электронной почты

Проверяет правильность предоставленного пароля и создает пользователя в базе данных

Создает утверждение со строковым именем «FullName» и предоставленным значением

Регистрирует пользователя, задав HttpContext.User; принципал будет включать специальное утверждение

При создании пользователя возникла проблема. Добавляет ошибки в ModelState и повторно отображает страницу

СОВЕТ В листинге 23.7 показано, как добавлять дополнительные утверждения во время регистрации, но вам часто потребуется добавить дополнительные данные позже, например утверждения, связанные с полномочиями, или другую информацию. Вам нужно будет создать дополнительные конечные точки и страницы для добавления этих данных, при необходимости обеспечивая безопасность страниц (например, чтобы пользователи не могли обновлять свои полномочия).

Это все, что требуется для *добавления* нового утверждения, но в настоящее время вы нигде его не *используете*. Что, если вы хотите отобразить его? Итак, вы добавили утверждение в `ClaimsPrincipal`, который был назначен свойству `HttpContext.User` при вызове `SignInAsync`. Это означает, что вы можете получить утверждения везде, где у вас есть доступ к `ClaimsPrincipal`, в том числе в обработчиках страниц и в шаблонах представлений. Например, можно было бы отобразить утверждение пользователя `FullName` в любом месте шаблона Razor с помощью следующей инструкции:

```
@User.Claims.FirstOrDefault(x=>x.Type == "FullName")?.Value
```

Будет найдено первое утверждение текущего принципала с типом `"FullName"` и выведено присвоенное значение (или, если утверждение не найдено, ничего выведено не будет). Система Identity даже включает в себя удобный метод расширения, который сокращает это выражение LINQ (он находится в пространстве имен `System.Security.Claims`):

```
@User.FindFirstValue("FullName")
```

Рассказав об этой любопытной детали, мы подошли к концу главы, посвященной ASP.NET Core Identity. Надеюсь, вы оценили, сколько усилий можно сэкономить с помощью Identity, особенно когда вы используете пакет Identity UI по умолчанию.

Добавление учетных записей пользователей и аутентификации в приложение обычно является первым шагом к дальнейшей настройке приложения. После аутентификации вы можете заняться авторизацией, которая позволяет заблокировать определенные действия в вашем приложении в зависимости от текущего пользователя. В следующей главе вы узнаете о системе авторизации ASP.NET Core и о том, как использовать ее для настройки своих приложений; в частности, приложения с рецептами, которое неплохо продвигается!

Резюме

- Аутентификация – это процесс определения того, кем вы являетесь, а авторизация – процесс определения того, что вам разрешено делать. Необходимо провести аутентификацию пользователей, прежде чем выполнить авторизацию;

- каждый запрос в ASP.NET Core ассоциирован с пользователем, также известным как принципал. По умолчанию без аутентификации это анонимный пользователь. Вы можете использовать объект `ClaimsPrincipal`, чтобы обеспечить разное поведение в зависимости от того, кто сделал запрос;
- текущий принципал для запроса отображается в свойстве `HttpContext.User`. Вы можете получить доступ к этому значению на страницах Razor и представлениях, чтобы узнать свойства пользователя, например его ID, имя или адрес электронной почты;
- у каждого пользователя есть набор утверждений (claims). Они представляют собой отдельные фрагменты информации о пользователе. Эти объекты могут быть свойствами физического пользователя, например `Name` и `Email`, или могут быть связаны с вещами, которые есть у пользователя, например `HasAdminAccess` или `IsVipCustomer`;
- в более ранних версиях ASP.NET вместо утверждений использовались роли. При необходимости их по-прежнему можно использовать, но по возможности следует работать с утверждениями;
- аутентификация в ASP.NET Core обеспечивается компонентом `AuthenticationMiddleware` и рядом сервисов аутентификации. Эти сервисы отвечают за настройку текущего принципала, когда пользователь выполняет вход, сохраняя его в файле cookie и загружая принципала из этого файла при последующих запросах;
- `AuthenticationMiddleware` добавляется путем вызова метода `UseAuthentication()` в конвейере промежуточного ПО. Он должен размещаться после вызова метода `UseRouting()` и перед методами `UseAuthorization()` и `UseEndpoints()`;
- ASP.NET Core Identity предоставляет низкоуровневые сервисы, необходимые для сохранения пользователей в базе данных, гарантируя безопасное хранение их паролей, а также для входа пользователей в учетную запись и выхода из нее. Вы должны сами предоставить пользовательский интерфейс для функциональности и подключить его к подсистеме Identity;
- пакет `Microsoft.AspNetCore.Identity.UI` предоставляет пользовательский интерфейс по умолчанию для системы Identity и включает в себя поддержку подтверждения по электронной почте, двухфакторную аутентификацию и внешних поставщиков входа в учетную запись. Необходимо выполнить дополнительную настройку, чтобы активировать эти функции;
- шаблон по умолчанию для веб-приложения с аутентификацией отдельной учетной записи использует ASP.NET Core Identity для хранения пользователей в базе данных с помощью EF Core. Он включает в себя весь шаблонный код, необходимый для подключения пользовательского интерфейса к системе Identity;
- можно использовать класс `UserManager<T>` для создания новых учетных записей пользователей, загрузки их из базы данных и изменения паролей. `SignInManager<T>` используется для входа и выхода пользователя из приложения, назначая принципала для запроса

и устанавливая cookie-файл аутентификации. Пользовательский интерфейс по умолчанию использует эти классы за вас, чтобы облегчить регистрацию пользователя и вход в учетную запись;

- можно обновить класс EF Core `DbContext` для поддержки `Identity`, наследуя от `IdentityDbContext<TUser>`, где `TUser` – это класс, который наследует от `IdentityUser`;
- вы можете добавить пользователю дополнительные утверждения с помощью метода `UserManager<TUser>.AddClaimAsync(TUser user, Claim claim)`. Они добавляются в объект `HttpContext.User`, когда пользователь входит в приложение;
- утверждения состоят из типа и значения. Оба значения являются строками. Вы можете использовать стандартные значения для типов, предоставляемых в классе `ClaimTypes`, например `ClaimTypes.GivenName` и `ClaimTypes.FirstName`, или специальную строку, например `"FullName"`.

24

Авторизация: обеспечиваем защиту приложения

В этой главе:

- использование авторизации для контроля над тем, кто может использовать ваше приложение;
- использование авторизации на основе утверждений и политик;
- создание специальных политик для работы со сложными требованиями;
- авторизация запроса в зависимости от ресурса, к которому осуществляется доступ;
- скрытие элементов из шаблона Razor, к которым у пользователя нет прав доступа.

В предыдущей главе я показал, как добавлять пользователей в приложение ASP.NET Core, используя аутентификацию. С помощью аутентификации пользователи могут регистрироваться и входить в приложение, используя адрес электронной почты и пароль. Каждый раз, когда вы добавляете аутентификацию в приложение, вы неизбежно сталкиваетесь с желанием ограничить действия некоторых пользователей. Процесс определения того, может ли пользователь выполнить данное действие в вашем приложении, называется *авторизацией*.

Например, на сайте для онлайн-торговли у вас могут быть администраторы, которым разрешено добавлять новые продукты и менять цены, продавцы, которым разрешено просматривать оформленные заказы, и клиенты, которым разрешено только размещать заказы и покупать продукты.

В этой главе я покажу вам, как использовать авторизацию в приложении, чтобы контролировать действия своих пользователей. В разделе 24.1 я познакомлю вас с авторизацией в контексте сценария из реальной жизни, с которым вы, вероятно, сталкивались: аэропорт. Я опишу последовательность событий, от регистрации и прохождения контроля безопасности до входа в зал ожидания аэропорта, и вы увидите, как все это соотносится с концепциями авторизации в этой главе.

В разделе 24.2 я покажу, как авторизация вписывается в веб-приложение ASP.NET Core и как это связано с классом `ClaimsPrincipal`, который вы видели в предыдущей главе. Вы увидите, как обеспечить простейший уровень авторизации в приложении ASP.NET Core, гарантируя, что только проверенные пользователи могут открывать страницу Razor Page или выполнять действие MVC.

Мы расширим этот подход в разделе 24.3, добавив понятие *политик*, которые позволяют устанавливать определенные требования для определенного проверенного пользователя, проверяя, чтобы у него были права для выполнения действия или страницы Razor. В этой главе основное внимание уделяется авторизации в Razor Pages и контроллерах Model-View-Controller (MVC); в главе 25 вы узнаете, как те же принципы применяются к минимальным API.

Вы будете широко использовать политики в системе авторизации ASP.NET Core, поэтому в разделе 24.4 мы рассмотрим, как работать с более сложными сценариями. Вы узнаете о требованиях к авторизации и обработчиках, как сочетать их для создания особых политик, которые можно применять к страницам Razor и действиям.

Иногда авторизация пользователя зависит от того, к какому ресурсу или документу он пытается получить доступ. Ресурс – это все то, что вы пытаетесь защитить, поэтому это может быть документ или сообщение в социальных сетях. Например, можно разрешить пользователям создавать свои документы или читать документы, полученные от других пользователей, но редактировать они смогут только те документы, которые создали сами. Такой тип авторизации, при котором вам необходимы данные документа, чтобы определить, авторизован ли пользователь, называется *авторизацией на основе ресурсов*. Этой теме посвящен раздел 24.5.

В последнем разделе данной главы я покажу, как расширить подход с использованием авторизации на основе ресурсов в шаблонах представлений Razor. Это позволяет изменять пользовательский интерфейс, чтобы скрыть элементы, с которыми пользователи не имеют права взаимодействовать. В частности, вы узнаете, как скрыть кнопку «Изменить», если у пользователя нет прав на редактирование сущности.

Мы начнем с более внимательного изучения концепции авторизации, разберем, чем она отличается от аутентификации и как это связано с концепциями из реальной жизни, которые вы можете увидеть в аэропорту.

24.1 Знакомство с авторизацией

В этом разделе вы познакомитесь с авторизацией, и мы сравним ее с аутентификацией.

Я использую пример из реальной жизни – пребывание в аэропорту, чтобы проиллюстрировать, как работает авторизация на основе утверждений.

Для тех, кто плохо знаком с веб-приложениями и безопасностью, термины «аутентификация» и «авторизация» иногда могут выглядеть несколько пугающими. Конечно, тот факт, что эти слова так похожи, не очень помогает! Эти два понятия часто используются вместе, но между ними определенно есть различия:

- **аутентификация** – процесс определения того, кто сделал запрос;
- **авторизация** – процесс определения того, разрешено ли запрошенное действие.

Как правило, сначала выполняется аутентификация, чтобы вы знали, кто выполняет запрос к вашему приложению. Если это традиционное веб-приложение, то оно проверяет запрос, обращаясь к зашифрованному файлу cookie, который был установлен при входе пользователя в учетную запись (как было показано в предыдущей главе). Веб-API обычно используют заголовок вместо файла cookie для аутентификации, но в основном процесс тот же, как будет показано в главе 25.

Как только запрос будет проверен и вы узнаете, кто его отправляет, можно определить, разрешено ли этому лицу выполнять действия на вашем сервере. Данный процесс называется *авторизацией*, которой и посвящена эта глава.

Прежде чем углубиться в код и приступить к изучению авторизации в ASP.NET Core, я продемонстрирую эти концепции в сценарии из реальной жизни, с которым вы, надеюсь, знакомы: проверка в аэропорту. Чтобы попасть в аэропорт и сесть в самолет, необходимо пройти несколько этапов. Сначала вы должны доказать, что вы – это вы (аутентификация).

После этого нужно пройти еще ряд этапов, в ходе которых будет установлено, можно ли вам двигаться дальше (авторизация). В упрощенном виде это может выглядеть так:

- 1 Вы предъявляете паспорт на стойке регистрации и получаете посадочный талон.
- 2 Вы предъявляете свой посадочный талон, чтобы пройти контроль службы безопасности, и проходите ее.
- 3 Вы предъявляете карту часто летающего пассажира, чтобы войти в бизнес-зал. Входите туда.
- 4 Вы предъявляете свой посадочный талон, чтобы сесть на самолет. Садитесь в самолет.

Очевидно, что эти шаги, также показанные на рис. 24.1, будут несколько отличаться в реальной жизни (у меня нет карты часто летающего пассажира!), но пока мы будем использовать их. Давайте подробнее рассмотрим каждый шаг.



Рис. 24.1 При посадке в самолет в аэропорту вы проходите несколько этапов авторизации. На каждом этапе необходимо предъявить утверждение в виде посадочного талона или карты часто летающего пассажира. Если вы не авторизованы, то доступ будет запрещен

По прибытии в аэропорт первое, что вы делаете, – идете к стойке регистрации. Здесь вы можете приобрести билет на самолет, но для этого вам нужно дать понять, кто вы, предъявив паспорт; вы *аутентифицируете* себя. Если вы забыли паспорт, то не можете пройти аутентификацию, и дальнейшие действия невозможны.

После покупки билета вам выдается посадочный талон, на котором указано, каким рейсом вы летите. Предположим, он также содержит BoardingPassNumber. Можно воспринимать этот номер как дополнительное утверждение, ассоциированное с вашей личностью.

ОПРЕДЕЛЕНИЕ Утверждение – это фрагмент информации о пользователе, состоящий из типа и необязательного значения.

Следующий шаг – безопасность. Сотрудники службы безопасности попросят вас предъявить посадочный талон, чтобы пройти проверку. Они должны удостовериться, что вы действительно куда-то летите и вам разрешено пройти дальше на территорию аэропорта. Это процесс авторизации: у вас должно быть необходимое утверждение (`BoardingPassNumber`), чтобы двигаться дальше.

Если у вас его нет, то далее может произойти одно из двух:

- если вы еще не купили билет – вас направят обратно к стойке регистрации, где вы сможете пройти аутентификацию и приобрести его. После этого вы можете снова попробовать пройти службу безопасности;
- если у вас недействительный билет – вас не пропустят через службу безопасности, а тут ничего не поделаешь. Если, например, вы предъявили посадочный талон, а улететь должны были неделю назад, то, скорее всего, вас не пропустят. (Не спрашивайте, откуда мне это известно!)

После прохождения проверки вам нужно дождаться посадки на рейс, но, к сожалению, свободных мест нет. Типичный случай! К счастью, вы регулярно летаете и накопили достаточно миль, чтобы получить золотой статус часто летающего пассажира, поэтому можете воспользоваться бизнес-залом.

Вы отправляетесь в зал ожидания, где вас просят предъявить золотую карту часто летающего пассажира дежурному, и вас впустят. Это еще один пример авторизации. Чтобы продолжить, у вас должно быть утверждение `FrequentFlyerClass` со значением `Gold`.

ПРИМЕЧАНИЕ До сих пор в этом сценарии вы использовали авторизацию дважды и каждый раз предъявляли утверждение. В первом случае было достаточно наличия любого `BoardingPassNumber`, тогда как для утверждения `FrequentFlyerClass` вам потребовалось определенное значение `Gold`.

Когда вы садитесь в самолет, наступает последний этап авторизации, на котором вы должны снова предъявить утверждение `BoardingPassNumber`. Вы уже делали это ранее, но посадка в самолет и проход через службу безопасности – разные действия, поэтому нужно предъявить его снова.

Весь этот сценарий имеет множество параллелей с запросами к веб-приложению:

- оба процесса начинаются с аутентификации;
- вы должны доказать, кто вы, чтобы получить утверждения, которые вам нужны для авторизации;
- вы используете авторизацию для защиты конфиденциальных действий, таких как проход через службу безопасности и вход в бизнес-зал.

Я буду использовать этот сценарий на протяжении всей главы, чтобы создать простое веб-приложение, имитирующее этапы, которые вы проходите в аэропорту. Мы рассмотрели концепцию авторизации в общих чертах, поэтому в следующем разделе разберем, как работает

авторизация в ASP.NET Core. Начнем с самого базового уровня авторизации, гарантируя, что только аутентифицированные пользователи могут выполнить действие, и посмотрим, что происходит, когда вы пытаетесь выполнить такое действие.

24.2 Авторизация в ASP.NET Core

В этом разделе вы увидите, как принципы авторизации, описанные в предыдущем разделе, применяются к приложению ASP.NET Core. Вы узнаете о роли атрибута [Authorize] и компонента AuthorizationMiddleware при авторизации запросов к страницам Razor Pages и действиям MVC. Наконец, вы узнаете, как не дать пользователям, не прошедшим аутентификацию, выполнять конечные точки, и что происходит, когда пользователи не проходят проверку.

В ASP.NET Core есть встроенная авторизация, поэтому вы можете использовать ее в любом месте своего приложения, но чаще всего применяется авторизация с компонентом AuthorizationMiddleware, который должен быть размещен после компонентов маршрутизации и аутентификации, но перед компонентом конечной точки, как показано на рис. 24.2.

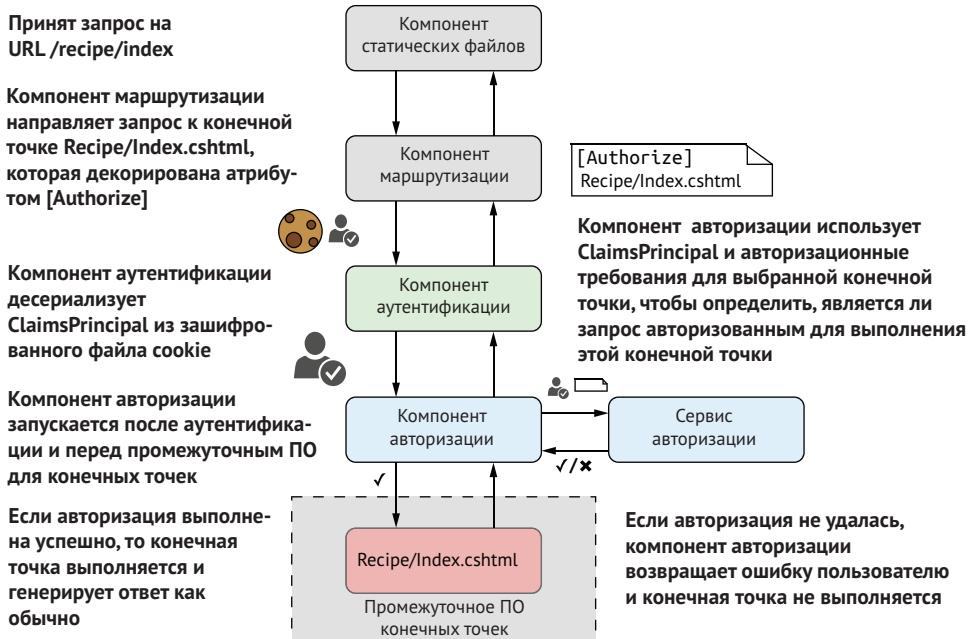


Рис. 24.2 Авторизация происходит после выбора конечной точки и после того, как запрос будет проверен, но до выполнения метода действия или конечной точки Razor Page

ПРИМЕЧАНИЕ Помните, что в ASP.NET Core конечная точка означает обработчик, выбранный компонентом маршрутизации, который будет генерировать ответ при выполнении. Обычно это страница Razor, или метод действия веб-API, или обработчик конечной точки минимального API.

Используя эту конфигурацию, `RoutingMiddleware` выбирает конечную точку для выполнения на основе URL-адреса запроса, например страницу Razor, как было показано в главе 14. Метаданные о выбранной конечной точке доступны всему промежуточному ПО, которое идет после компонента маршрутизации. Они содержат сведения обо всех требованиях к авторизации для конечной точки и обычно определяются путем декорирования действия или страницы Razor атрибутом `[Authorize]`.

`AuthenticationMiddleware` десериализует зашифрованный файл cookie (или токен носителя для API), ассоциированный с запросом, чтобы создать объект `ClaimsPrincipal`. Этот объект задается как свойство `HttpContext.User` для запроса, поэтому все последующие компоненты промежуточного ПО могут получить доступ к данному значению. Он содержит все утверждения, которые были добавлены в файл cookie, когда пользователь прошел аутентификацию.

Теперь переходим к компоненту `AuthorizationMiddleware`. Этот компонент проверяет, есть ли у выбранной конечной точки какие-либо требования к авторизации, на основе метаданных, предоставленных `RoutingMiddleware`. Если таковые имеются, `AuthorizationMiddleware` использует `HttpContext.User`, чтобы определить, прошел ли текущий запрос аутентификацию и может ли он выполнить конечную точку.

Если запрос прошел аутентификацию, следующий компонент в конвейере выполняется как обычно. В противном случае `AuthorizationMiddleware` прерывает выполнение конвейера, а компонент конечной точки не выполняется.

ПРИМЕЧАНИЕ Вызов `UseAuthorization()` всегда должен размещаться после `UseRouting()` и `UseAuthentication()`, но перед `UseEndpoints()`. `WebApplication` автоматически добавляет все это промежуточное программное обеспечение в правильном порядке, но если вы переопределяете позицию в конвейере, например вызывая `UseRouting()`, вы должны обязательно поддерживать этот общий порядок.

`AuthorizationMiddleware` отвечает за реализацию требований авторизации и гарантию того, что только проверенные пользователи могут выполнять защищенные конечные точки. В разделе 24.2.1 вы узнаете, как применить простейшее требование авторизации, а в разделе 24.2.2 увидите, как реагирует фреймворк, когда пользователю не разрешается выполнить конечную точку.

24.2.1 Предотвращение доступа анонимных пользователей к приложению

Думая об авторизации, вы обычно думаете о том, как проверить, что у конкретного пользователя есть полномочия на выполнение конечной точки. В ASP.NET Core обычно это можно сделать, проверив, есть ли у пользователя определенное утверждение.

Есть еще более простой уровень авторизации, который мы еще не рассматривали, – когда вы позволяете только проверенным пользова-

телям выполнять конечную точку. Это даже проще, чем сценарий с утверждениями (к которому мы вернемся позже), поскольку здесь есть только две возможности:

- *пользователь прошел аутентификацию* – действие выполняется в обычном режиме;
- *пользователь не прошел аутентификацию* – пользователь не может выполнить конечную точку.

Такого базового уровня авторизации можно достичь с помощью атрибута `[Authorize]`, который вы видели в главе 22, когда мы обсуждали фильтры авторизации. Вы можете применить его к своим действиям и страницам Razor, как показано в следующем листинге, чтобы ограничить их только проверенными (выполнившими вход) пользователями. Если не прошедший аутентификацию пользователь пытается выполнить действие или страницу Razor, защищенные атрибутом `[Authorize]`, он будет перенаправлен на страницу входа.

Листинг 24.1 Применение атрибута `[Authorize]` к действию

```
public class RecipeApiController : ControllerBase
{
    public IActionResult List() <-- Это действие может выполнить кто
    {
        return Ok();
    } <-- Применяет атрибут [Authorize] к отдельным действиям,
    [Authorize] <-- целым контроллерам или страницам Razor
    public IActionResult View() <-- Это действие могут выполнять
    {
        return Ok();
    } <-- только пользователи, прошедшие
} <-- аутентификацию
```

Применяя атрибут `[Authorize]` к конечной точке, вы добавляете к ней метаданные, указывая на то, что только проверенные пользователи могут получить к ней доступ. Как было показано на рис. 24.2, эти метаданные становятся доступными для `AuthorizationMiddleware`, когда `RoutingMiddleware` выбирает конечную точку.

Можно применять атрибут `[Authorize]` в области метода действия, контроллера, страницы Razor или глобально, как вы видели в главе 13.

Любое действие или страница Razor, имеющие этот атрибут, примененный таким образом, могут быть выполнены только аутентифицированным пользователем. Пользователи, не прошедшие аутентификацию, будут перенаправлены на страницу входа.

СОВЕТ Существует несколько различных способов глобального применения атрибута `[Authorize]`. Об этих вариантах и о том, что и когда выбирать, можно прочитать в моем блоге: <http://mng.bz/opQp>.

Иногда, особенно когда вы применяете атрибут `[Authorize]` глобально, вам, возможно, понадобится добавить исключения для этого требования.

Если применить его глобально, то любой запрос, не прошедший аутентификацию, будет перенаправлен на страницу входа вашего приложения.

Но если этот атрибут глобальный, то, когда страница входа попытается загрузиться, вы не пройдете аутентификацию и снова будете перенаправлены на страницу входа. Получается, вы застряли в бесконечном цикле переадресации.

Чтобы избежать этого, можно назначить определенным конечным точкам игнорировать этот атрибут, применив атрибут `[AllowAnonymous]` к действию или странице Razor, как показано ниже. Это позволит пользователям, не прошедшим аутентификацию, выполнить действие и может помочь вам избежать цикла переадресации, который мог бы возникнуть в противном случае.

Листинг 24.2 Применение атрибута `[AllowAnonymous]`, чтобы разрешить доступ пользователям, не прошедшим аутентификацию

```
[Authorize]
public class AccountController : ControllerBase
{
    public IActionResult ManageAccount()
    {
        return Ok();
    }
    [AllowAnonymous]
    public IActionResult Login()
    {
        return Ok();
    }
}
```

Только пользователи, прошедшие аутентификацию, могут выполнять `ManageAccount`

Применяется в области действия контроллера, поэтому пользователь должен быть аутентифицирован для всех действий контроллера

Атрибут `[AllowAnonymous]` переопределяет атрибут `[Authorize]`, чтобы допустить пользователей, не прошедших аутентификацию

Вход в приложение может быть выполнен анонимными пользователями

ВНИМАНИЕ! Если вы применяете атрибут `[Authorize]` глобально, обязательно добавьте атрибут `[AllowAnonymous]` к действиям входа, ошибок и сброса пароля, а также любым другим действиям, которые должны выполнять пользователи, не прошедшие аутентификацию. Если вы используете пользовательский интерфейс Identity по умолчанию, описанный в главе 23, то все уже настроено за вас.

Если пользователь, не прошедший аутентификацию, пытается выполнить действие, защищенное атрибутом `[Authorize]`, традиционные веб-приложения перенаправляют его на страницу входа. А что насчет веб-API или более сложных сценариев, когда пользователь выполнил вход, но у него нет необходимых утверждений для выполнения действия? В разделе 24.2.2 мы рассмотрим, как сервисы аутентификации ASP.NET Core решают все эти вопросы за вас.

24.2.2 Обработка запросов, не прошедших аутентификацию

В предыдущем разделе вы видели, как применить атрибут `[Authorize]` к действию, чтобы гарантировать, что его могут выполнять только

аутентифицированные пользователи. В разделе 24.3 мы рассмотрим более сложные примеры, которые требуют от вас наличия конкретного утверждения.

В обоих случаях вы должны отвечать одному или нескольким требованиям авторизации (например, должны быть аутентифицированы), чтобы выполнить действие.

Если пользователь отвечает требованиям авторизации, то запрос проходит беспрепятственно через компонент `AuthorizationMiddleware`, а конечная точка выполняется в `EndpointMiddleware`. Если он не отвечает требованиям выбранной конечной точки, `AuthorizationMiddleware` завершит запрос. В зависимости от причины, по которой запрос не прошел проверку, `AuthorizationMiddleware` генерирует один из двух типов ответов, как показано на рис. 24.3:

- *вызов (challenge)* – этот ответ указывает на то, что пользователь не был авторизован для выполнения действия, потому что еще не выполнил вход;
- *запрет (forbid)* – этот ответ указывает на то, что пользователь выполнил вход, но не отвечает требованиям для выполнения действия. Например, у него не было требуемого утверждения.

ПРИМЕЧАНИЕ Если вы примените атрибут `[Authorize]` в базовой форме, как делали это в разделе 24.2.1, то будете генерировать только ответы типа «вызов». В данном случае «вызов» будет сгенерирован для пользователей, не прошедших аутентификацию, но аутентифицированные пользователи всегда будут считаться авторизованными.

Точный ответ HTTP, сгенерированный «вызовом» или «запретом», обычно зависит от типа приложения, которое вы создаете, а следовательно, от типа аутентификации, которое оно использует: традиционное ли это веб-приложение с Razor Pages или приложение API.

Для традиционных веб-приложений, использующих аутентификацию с файлами cookie, например при использовании ASP.NET Core Identity, как и в главе 23, вызовы и запреты генерируют HTTP-перенаправление на страницу в вашем приложении. «Вызов» указывает на то, что пользователь еще не прошел аутентификацию, поэтому он перенаправляется на страницу входа в приложение. После входа он может попытаться получить доступ к защищенному ресурсу снова. «Запрет» означает, что запрос исходил от пользователя, который уже выполнил вход, но ему по-прежнему не разрешено выполнять действие. Следовательно, пользователь перенаправляется на страницу «запрещено» или «доступ запрещен», как показано на рис. 24.4, которая информирует его о том, что он не может выполнить действие, или страницу Razor Page.

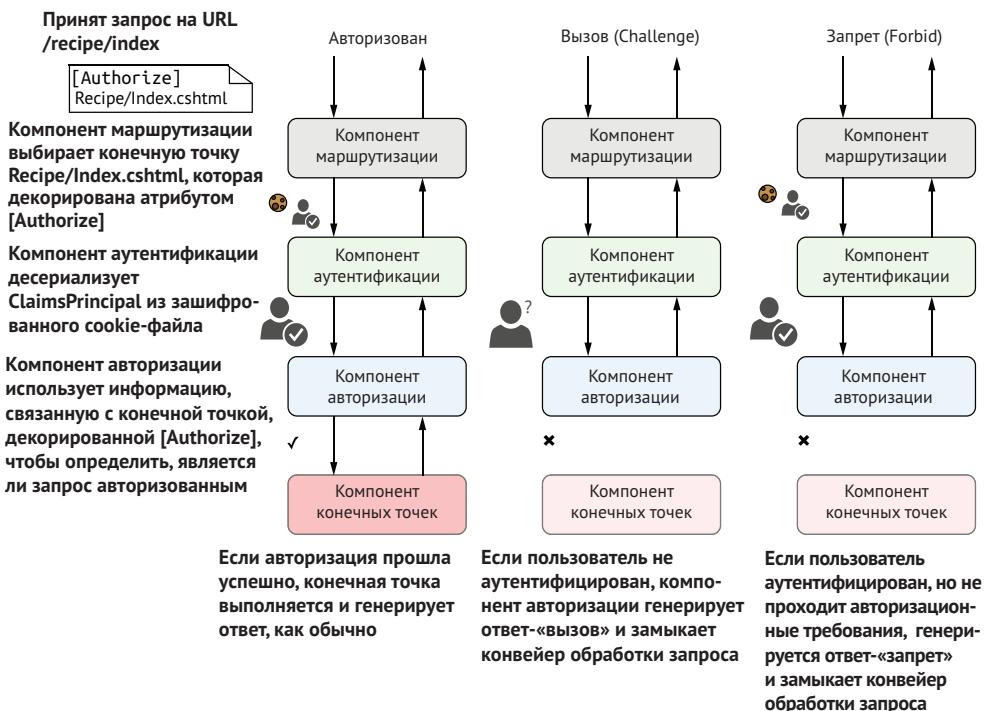


Рис. 24.3 Три типа ответа на попытку авторизации. В примере слева запрос содержит cookie-файл аутентификации, поэтому пользователь проходит аутентификацию в AuthenticationMiddleware. AuthorizationMiddleware подтверждает, что проверенный пользователь может получить доступ к выбранной конечной точке, поэтому конечная точка выполняется. В центральном примере запрос не прошел проверку, поэтому AuthorizationMiddleware генерирует «вызов». В примере справа запрос прошел проверку, но у пользователя нет полномочий на выполнение конечной точки, поэтому генерируется «запрет»

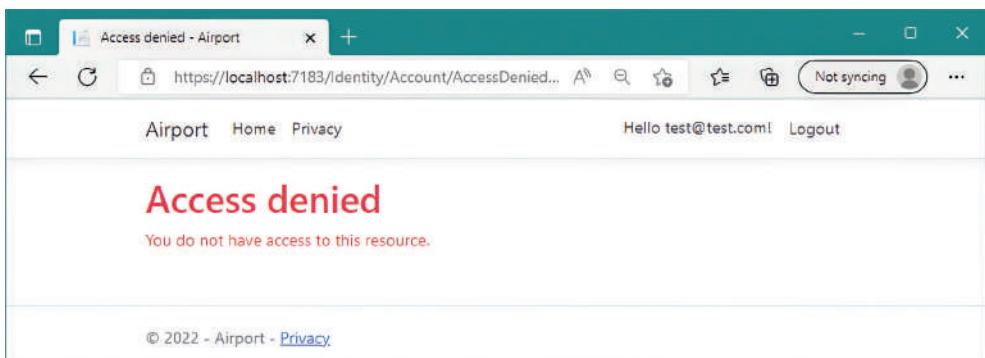


Рис. 24.4 Ответ, сообщающий об отказе в доступе, в традиционных веб-приложениях, использующих аутентификацию с файлами cookie. Если у вас нет полномочий, чтобы выполнить страницу Razor, и вы уже выполнили вход, то вы будете перенаправлены на страницу с отказом в доступе

Предыдущее поведение является стандартным для традиционных веб-приложений, но веб-API обычно используют другой подход к аутентификации, как вы увидите в главе 25. Вместо того чтобы выполнить вход и использовать API напрямую, вы обычно входите в стороннее приложение, предоставляющее токен клиентскому одностороничному приложению или приложению для мобильных устройств.

Клиентское приложение отправляет его, когда делает запрос к вашему веб-API.

Аутентификация запроса для приложения API, по сути, идентична традиционному веб-приложению, использующему файлы cookie, как будет показано в главе 25. `AuthenticationMiddleware` десериализует файл cookie или токен для создания `ClaimsPrincipal`. Разница в том, как веб-API обрабатывает отказы в авторизации.

Когда веб-API генерирует «вызов», то возвращает ошибку `401 Unauthorized` вызывающей стороне. Аналогично, когда приложение генерирует «запрет», оно возвращает ответ `403 Forbidden`. Традиционное веб-приложение, по сути, обрабатывает эти ошибки, автоматически перенаправляя пользователей, не прошедших аутентификацию, на страницу входа или страницу с надписью «доступ запрещен», но веб-API так не делает.

Клиентское одностороничное приложение или приложение для мобильных устройств должно самостоятельно обнаруживать эти ошибки и обрабатывать их соответствующим образом.

СОВЕТ Такая разница в поведении – одна из причин, по которой я обычно рекомендую создавать отдельные приложения для ваших API и приложений Razor Pages – можно использовать и то, и другое в одном приложении, но конфигурация будет более сложной.

Различное поведение традиционных и одностороничных веб-приложений поначалу может сбивать с толку, но на практике обычно не нужно об этом слишком беспокоиться. Независимо от того, создаете ли вы веб-API или традиционное MVC-приложение, код авторизации будет выглядеть одинаково в обоих случаях. Примените атрибуты `[Authorize]` к своим конечным точкам и позвольте фреймворку самому позаботиться об этих различиях.

ПРИМЕЧАНИЕ В главе 23 вы увидели, как настроить ASP.NET Core Identity в приложении Razor Pages. В этой главе предполагается, что вы также создаете приложение Razor Pages, но она в равной степени применима и к веб-API. Политики авторизации применяются одинаково, независимо от того, какое приложение вы создаете. Отличается только ответ на неавторизованные запросы.

Вы узнали, как применить самое основное требование авторизации – ограничить выполнение конечной точки только аутентифицированными пользователями, – но большинству приложений нужно что-то более тонкое, нежели подход «все или ничего».

Рассмотрим сценарий с аэропортом из раздела 24.1. Пройти аутентификацию (наличие паспорта) недостаточно, чтобы пройти через службу безопасности. Вам также понадобится конкретное утверждение: `BoardingPassNumber`.

В следующем разделе мы разберем, как реализовать аналогичное требование в ASP.NET Core.

24.3 Использование политик для авторизации на основе утверждений

В предыдущем разделе вы узнали, как сделать так, чтобы пользователи выполняли вход для доступа к конечной точке. В этом разделе вы увидите, как применять дополнительные требования. Вы узнаете, как использовать политики авторизации для выполнения авторизации на основе утверждений, чтобы у выполнившего вход пользователя были необходимые утверждения для выполнения определенной конечной точки.

В главе 23 вы видели, что аутентификация в ASP.NET Core сосредоточена вокруг объекта `ClaimsPrincipal`, представляющего пользователя. У этого объекта имеется коллекция утверждений, содержащих информацию о пользователе, такую как его имя, адрес электронной почты и дату рождения.

Вы можете использовать их, чтобы настроить приложение под каждого пользователя, например отображая приветственное сообщение и обращаясь к пользователю по имени, но также можете использовать утверждения для авторизации. Например, можно авторизовать пользователя, только если у него есть конкретное утверждение (например, `BoardingPassNumber`) или если утверждения есть определенное значение (утверждение `FrequentFlyerClass` со значением `Gold`). В ASP.NET Core правила, определяющие, авторизован ли пользователь, инкапсулированы в *политике*.

ОПРЕДЕЛЕНИЕ Политика определяет требования, которым вы должны отвечать, чтобы запрос был авторизован.

Политики можно применять к действию с помощью атрибута `[Authorize]`, как было показано в разделе 24.2.1. В этом листинге показана модель страницы Razor, представляющая первый этап авторизации в сценарии с аэропортом. Страница Razor, `AirportSecurity.cshtml`, защищена атрибутом `[Authorize]`, но мы также предоставили имя политики: "CanEnterSecurity".

Листинг 24.3 Применение политики авторизации к странице Razor

```
[Authorize("CanEnterSecurity")]
public class AirportSecurityModel : PageModel
{
    public void OnGet()
    {
    }
}
```

← Применение политики CanEnterSecurity с помощью атрибута [Authorize]

← Только пользователи, удовлетворяющие политике CanEnterSecurity, могут выполнять страницу Razor

Если пользователь пытается выполнить страницу `AirportSecurity.cshtml`, компонент авторизации проверит, отвечает ли пользователь требованиям политики (мы вскоре рассмотрим саму политику). Это дает один из трех возможных результатов:

пользователь отвечает требованиям политики – конвейер промежуточного ПО продолжает работу, и `EndpointMiddleware` выполняет страницу `Razor`, как обычно;

пользователь не прошел аутентификацию – пользователь перенаправляется на страницу входа;

пользователь прошел аутентификацию, но не отвечает требованиям политики – пользователь перенаправляется на страницу с надписями «Запрещено» или «Отказано в доступе».

Эти три результата коррелируют с реальными результатами, которых вы можете ожидать, когда пытаетесь пройти службу безопасности в аэропорту:

- *у вас есть действующий посадочный талон* – вы можете пройти службу безопасности, как обычно;
- *у вас нет посадочного талона* – вас отправят покупать билет;
- *ваши посадочные талоны недействительны* (*например, вы опоздали на день*) – вас не пропустят.

В листинге 24.3 показано, как применить политику к странице `Razor` с помощью атрибута `[Authorize]`, но вам еще нужно определить политику `CanEnterSecurity`.

Сначала вы добавляете сервисы авторизации и возвращаете объект `AuthorizationBuilder` с помощью `AddAuthorizationBuilder()`. Затем вы можете добавить политики в построитель, вызвав `AddPolicy()`. Вы определяете саму политику, вызывая методы в лямбда-методе в `AuthorizationPolicyBuilder` (здесь он называется `policyBuilder`).

Листинг 24.4 Добавление политики авторизации с помощью `AuthorizationPolicyBuilder`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorizationBuilder() <-- Вызывает метод
    .AddPolicy( <-- AddAuthorization для до-
        "CanEnterSecurity", <-- Добавляет новую
        policyBuilder => policyBuilder <-- бавления нужных серви-
            .RequireClaim("BoardingPassNumber")); <-- ксов авторизации
    );
    // Дополнительная конфигурация; <-- Определяет требования
                                                политики с помощью
                                                AuthorizationPolicyBuilder
```

Задает имя политики

политики

Когда вы вызываете метод `AddPolicy`, то указываете имя политики, соответствующее значению, которое вы используете в своих атрибутах `[Authorize]`, и определяете требования политики. В этом примере у вас есть одно простое требование: у пользователя должно быть утверждение типа `BoardingPassNumber`. Если у пользователя оно есть, независимо от его значения, это будет отвечать требованиям политики, и пользователь будет авторизован.

ПРИМЕЧАНИЕ Утверждение – это информация о пользователе, имеющая вид пары «ключ–значение». Политика определяет требования для успешной авторизации. Она может потребовать, чтобы у пользователя было утверждение, а также указать более сложные требования, как вы вскоре увидите.

Класс `AuthorizationPolicyBuilder` содержит несколько методов для создания простых политик, таких как эта, как показано в табл. 24.1. Например, перегруженный вариант метода `RequireClaim()` позволяет указать конкретное значение, которое должно иметь утверждение. Следующая строка кода позволяет создать политику, в которой утверждение `BoardingPassNumber` должно иметь значение "A1234":

```
policyBuilder => policyBuilder.RequireClaim("BoardingPassNumber", "A1234");
```

Таблица 24.1 Простые методы политик в классе `AuthorizationPolicyBuilder`

Метод	Поведение политики
<code>RequireAuthenticatedUser()</code>	Пользователь должен быть аутентифицирован. Создает политику, аналогичную атрибуту <code>[Authorize]</code> по умолчанию, где вы не устанавливаете политику
<code>RequireClaim(утверждение, значения)</code>	У пользователя должно быть указанное утверждение. Если это предусмотрено, то оно должно быть одним из указанных значений
<code>RequireUsername(имя пользователя)</code>	У пользователя должно быть указанное имя пользователя
<code>RequireAssertion(функция)</code>	Выполняет предоставленную лямбда-функцию, которая возвращает логическое значение, указывая на то, отвечает ли вы требованиям политики

Авторизация на основе ролей или авторизация на основе утверждений

Если вы посмотрите на все методы, доступные в классе `AuthorizationPolicyBuilder` с помощью `IntelliSense`, то заметите, что здесь есть метод, о котором я не упоминал в табл. 24.1, `RequireRole()`. Он применялся в подходе на основе ролей, который использовался в предыдущих версиях ASP.NET, и я не рекомендую его использовать.

До того, как Microsoft приняла за стандарт авторизацию на основе утверждений, используемую ASP.NET Core и в последних версиях ASP.NET, авторизация на основе ролей была нормой. Пользователи назначались одной или нескольким ролям, таким как `Administrator` или `Manager`, и авторизация включала в себя проверку того, находился ли текущий пользователь в требуемой роли.

Такой подход к авторизации на основе ролей возможен и в ASP.NET Core, но в первую очередь он используется по причинам совместимости с прежними версиями. Рекомендуется применять авторизацию на основе утверждений. Если вы не переносите устаревшее приложение, использующее роли, то рекомендую использовать авторизацию на основе утверждений.

Можно применять эти методы для создания простых политик, которые могут справляться с базовыми ситуациями, но часто вам нужно что-то посложнее. Что, если вы хотите создать политику, обеспечивающую выполнение конечной точки только пользователями старше 18 лет?

Утверждение `DateOfBirth` предоставляет необходимую информацию, но у него нет единственного правильного значения, поэтому вы не сможете использовать метод `RequireClaim()`. Вы могли бы использовать метод `RequireAssertion()` и реализовать функцию, которая вычисляет возраст из утверждения `DateOfBirth`, но это может сделать код более запутанным.

Для более сложных политик, которые не так просто определить с помощью метода `RequireClaim()`, я рекомендую использовать другой подход и создавать специальную политику. Об этом в следующем разделе.

24.4 Создание пользовательских политик авторизации

Вы уже видели, как создать политику, проверяющую конкретное утверждение или требующую утверждения с определенным значением, но часто требования будут более сложными. В этом разделе вы узнаете, как создавать специализированные требования авторизации и обработчики, а также как настроить требования для авторизации, где есть несколько способов, позволяющих отвечать требованиям политики, при этом каждый из которых допустим.

Вернемся к примеру с аэропортом. Вы уже настроили политику для прохождения через службу безопасности и теперь собираетесь настроить политику, которая будет контролировать, имеете ли вы право входить в бизнес-зал.

Как видно на рис. 24.1, вам разрешено входить в бизнес-зал, если у вас есть утверждение `FlyerClass` со значением `Gold`. Если бы это было единственное утверждение, то для создания такой политики можно было бы использовать класс `AuthorizationPolicyBuilder`:

```
options.AddPolicy("CanAccessLounge", policyBuilder =>
    policyBuilder.RequireClaim("FrequentFlyerClass", "Gold");
```

Но что, если требования гораздо сложнее? Например, предположим, что вы можете войти в бизнес-зал, если вам исполнилось 18 лет (на основе расчетов из утверждения `DateOfBirth`), и вы являетесь представителем одной из следующих категорий:

- вы часто летаете, и у вас есть золотая карта (у вас есть утверждение `FlyerClass` со значением `Gold`);
- вы сотрудник авиакомпании (у вас есть утверждение `EmployeeNumber`).

Если вам когда-либо запрещали доступ в бизнес-зал (у вас есть утверждение `IsBannedFromLounge`), вас не пустят, даже если вы отвечаете остальным требованиям.

Невозможно выполнить этот сложный набор требований с помощью базового использования класса `AuthorizationPolicyBuilder`, как уже

было показано. К счастью, эти методы представляют собой обертку для набора строительных блоков, которые можно комбинировать, чтобы получить желаемую политику.

24.4.1 Требования и обработчики: строительные блоки политики

Каждая политика в ASP.NET Core состоит из одного или нескольких требований, и у каждого требования может быть один или несколько обработчиков. Например, в зале ожидания аэропорта у вас есть одна политика ("CanAccessLounge"), два требования (MinimumAgeRequirement и AllowedInLoungeRequirement) и несколько обработчиков, как показано на рис. 24.5.

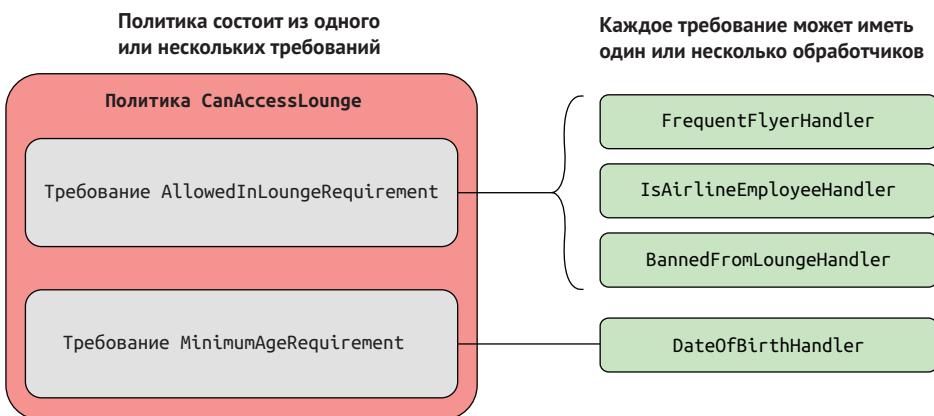


Рис. 24.5 У политики может быть много требований, и у каждого требования может быть много обработчиков. Объединив несколько требований в политику и предоставив несколько реализаций обработчиков, можно создавать сложные политики авторизации, которые соответствуют вашим потребностям

Чтобы соответствовать политике, пользователь должен выполнить *все* требования. Если пользователь терпит неудачу с *каким-либо* из этих требований, компонент авторизации не разрешит выполнить защищенную конечную точку. В этом примере пользователю должен быть разрешен доступ в бизнес-зал, и он должен быть старше 18 лет.

Каждое требование может иметь один или несколько обработчиков, которые подтверждают, что требование было выполнено. Например, как показано на рис. 24.5, у `AllowedInLoungeRequirement` есть два обработчика, которые могут удовлетворить это требование:

- `FrequentFlyerHandler`;
- `IsAirlineEmployeeHandler`.

Если пользователь удовлетворяет любому из этих обработчиков, это значит, что он удовлетворяет `AllowedInLoungeRequirement`. Вам не нужны все обработчики, чтобы требование было удовлетворено, нужен только один.

ПРИМЕЧАНИЕ На рис. 24.5 показан третий обработчик, `BannedFromLoungeHandler`, о котором я расскажу в разделе 24.4.2. Он немного отличается в том смысле, что может использоваться только для проверки несоответствия требованию.

Вы можете использовать требования и обработчики для достижения практически любой комбинации поведения, необходимой вам для политики. Сочетая обработчики для требования, вы можете проверять условия, используя логический оператор OR: если какой-либо из обработчиков удовлетворен, то требование удовлетворено. Комбинируя требования, вы создаете логический оператор AND: все требования должны быть удовлетвореными, чтобы отвечать политике, как показано на рис. 24.6.

Чтобы политика была удовлетворена, необходимо, чтобы каждое требование было удовлетворено

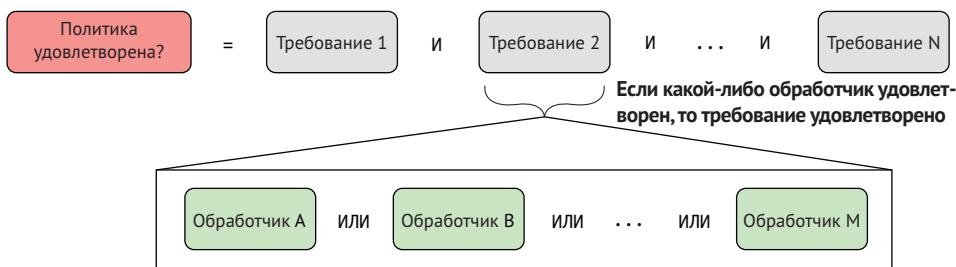


Рис. 24.6 Чтобы удовлетворять политике, необходимо отвечать всем требованиям. Вы отвечаете требованию, если отвечаете какому-либо обработчику

СОВЕТ Вы также можете добавить несколько политик на страницу Razor Page или метод действия, многократно применяя атрибут [Authorize], например [Authorize("Policy1"), Authorize("Policy2")]. Чтобы запрос был авторизован, должны быть удовлетворены все политики.

Я выделил требования и обработчики, из которых состоит ваша политика "CanAccessLounge", поэтому в следующем разделе вы создадите каждый из компонентов и примените их в приложении для аэропорта, которое мы используем в качестве примера.

24.4.2 Создание политики со специальным требованием и обработчиком

Вы видели все составляющие специальной политики авторизации, поэтому в этом разделе мы рассмотрим реализацию политики "CanAccessLounge".

Создание IAuthorizationRequirement для представления требования

Как вы видели, у специальной политики может быть несколько требований, но что такое требование на языке кода? Требование к авторизации в ASP.NET Core – это любой класс, реализующий интерфейс `IAuthorizationRequirement`. Это пустой интерфейс-маркер, который можно применить к любому классу, чтобы указать, что он предстает-

ляет собой требование. Если у интерфейса нет членов, то как должен выглядеть класс требований? Обычно это простые РОСО-классы. В следующем листинге показан класс `AllowedInLoungeRequirement`, который примерно так же прост, как и требование. У него нет свойств или методов; он реализует необходимый интерфейс `IAuthorizationRequirement`.

Листинг 24.5 Класс AllowedInLoungeRequirement

```
public class AllowedInLoungeRequirement : IAuthorizationRequirement { } Интерфейс идентифицирует класс как требование авторизации
```

Это простейшая форма требования, но часто у них может быть одно или два свойства, которые делают требование более универсальным. Например, вместо того чтобы создавать очень специфичное требование `MustBe18YearsOldRequirement`, можно было бы создать параметризованное требование `MinimumAgeRequirement`, как показано в следующем листинге. Указав минимальный возраст в качестве параметра требования, вы можете повторно использовать его для других политик с иными требованиями к минимальному возрасту.

Листинг 24.6 Параметризованное требование MinimumAgeRequirement

```
public class MinimumAgeRequirement : IAuthorizationRequirement { Интерфейс идентифицирует класс как требование авторизации
    public MinimumAgeRequirement(int minimumAge) { При создании требования указывается минимальный возраст
        MinimumAge = minimumAge;
    }
    public int MinimumAge { get; } Обработчики могут использовать установленный минимальный возраст, чтобы определить, выполнено ли требование
}
```

Требования – самая легкая часть. Они представляют каждый из компонентов политики, которым вы должны соответствовать, чтобы отвечать политике в целом.

Создание политики с несколькими требованиями

Вы создали два требования, поэтому теперь можете настроить политику "CanAccessLounge", чтобы использовать их. Делается это, как и раньше, в методе `ConfigureServices` файла `Startup.cs`. В листинге 24.7 показано, как это сделать, создав экземпляр каждого требования и передав их классу `AuthorizationPolicyBuilder`. Обработчики авторизации будут использовать эти объекты требований при попытке авторизовать политику.

Листинг 24.7 Создание политики авторизации со множеством требований

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.services.AddAuthorization(options =>
{
    options.AddPolicy(
        "CanEnterSecurity",
        policyBuilder => policyBuilder
            .RequireClaim(Claims.BoardingPassNumber)); | Добавляет преды-
                                                    | дущую простую
                                                    | политику для про-
                                                    | хождения службы
                                                    | безопасности

    options.AddPolicy(
        "CanAccessLounge",
        policyBuilder => policyBuilder.AddRequirements( | Добавляет экзем-
                                                    | пляр каждого объекта
                                                    | IAuthorizationRequirement
            new MinimumAgeRequirement(18),
            new AllowedInLoungeRequirement()
        ));
}); | Добавляет новую политику для бизнес-
      | зала аэропорта, CanAccessLounge

// Дополнительная конфигурация сервиса;
```

Добавляет новую политику для бизнес-
зала аэропорта, CanAccessLounge

Теперь у вас есть политика `CanAccessLounge` с двумя требованиями, поэтому вы можете применить ее к странице Razor или методу действия с помощью атрибута `[Authorize]` точно так же, как и для политики `"CanEnterSecurity"`:

```
[Authorize("CanAccessLounge")]
public class AirportLoungeModel : PageModel
{
    public void OnGet() { }
```

Когда запрос маршрутизируется на страницу `AirportLounge.cshtml`, компонент авторизации выполняет политику авторизации и проверяет каждое из требований. Но, как вы могли заметить, требования – это просто данные; они указывают, что нужно выполнить, но не описывают, как именно. Для этого необходимо написать несколько обработчиков.

Создаем обработчики авторизации, чтобы отвечать требованиям

Обработчики авторизации содержат логику того, как соответствовать конкретному интерфейсу `IAuthorizationRequirement`. При выполнении обработчик может делать одно из трех:

- отметить обработку требований как успешную;
- ничего не делать;
- явно отметить несоответствие требованию.

Обработчики должны реализовать `AuthorizationHandler<T>`, где `T` – тип требования, которое они обрабатывают. Например, в следующем листинге показан обработчик для `AllowedInLoungeRequirement`, который проверяет, есть ли у пользователя утверждение `FrequentFlyerClass` со значением `Gold`.

Листинг 24.8 Обработчик FrequentFlyerHandler для требования AllowedInLoungeRequirement

```

public class FrequentFlyerHandler : 
    AuthorizationHandler<AllowedInLoungeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, 
        AllowedInLoungeRequirement requirement)
    {
        if(context.User.HasClaim("FrequentFlyerClass", "Gold"))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}

```

Обработчик реализует AuthorizationHandler<T>

Вы должны переопределить абстрактный метод HandleRequirementAsync

Контекст содержит такие сведения, как пользовательский объект ClaimsPrincipal

Проверяет, есть ли у пользователя утверждение FrequentFlyerClass со значением Gold

Если у пользователя было необходимо утверждение, отмечаем его как удовлетворенное, вызывая метод Succeed

Экземпляр требования для обработки

Если требование не было выполнено, ничего не предпринимаем

Этот обработчик функционально эквивалентен простому обработчику `RequireClaim()`, который вы видели в начале раздела 24.4, но он использует подход «требование + обработчик». Когда запрос маршрутизируется на страницу `AirportLounge.cshtml`, компонент авторизации видит атрибут `[Authorize]` на конечной точке с политикой `"CanAccessLounge"`. Он перебирает все требования в политике и всех обработчиков для каждого требования, вызывая для каждого метод `HandleRequirementAsync`. Компонент авторизации передает текущий `AuthorizationHandlerContext` и требование для проверки каждому обработчику. Текущий объект `ClaimsPrincipal`, который авторизуется, предоставляется в контексте как свойство `User`. В листинге 24.8 обработчик `FrequentFlyerHandler` использует контекст для проверки наличия утверждения `FrequentFlyerClass` со значением `Gold` и, если оно существует, указывает на то, что пользователю разрешено войти в зал ожидания авиакомпании, вызывая метод `Succeed()`.

ПРИМЕЧАНИЕ Обработчики помечают требование как успешно выполненное, вызывая метод `context.Succeed()` и передавая требование в качестве аргумента.

Важно отметить поведение, при котором у пользователя *нет* утверждения. Утверждение `FrequentFlyerHandler` ничего не делает в этом случае (оно возвращает завершенную задачу, чтобы соответствовать сигнатуре метода).

ПРИМЕЧАНИЕ Помните, что если какой-либо из обработчиков, ассоциированных с требованием, проходит, то требование считается удовлетворенным. Достаточно одного успешно выполненного обработчика, чтобы отвечать требованию.

Такое поведение, при котором вы либо вызываете метод `context.Succeed()`, либо ничего не делаете, типично для обработчиков авторизации. В следующем листинге показана реализация обработчика `IsAirlineEmployeeHandler`, который использует аналогичную проверку утверждения, чтобы определить, отвечает ли вы требованию.

Листинг 24.9 Обработчик `IsAirlineEmployeeHandler`

```
public class IsAirlineEmployeeHandler : AuthorizationHandler<AllowedInLoungeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        AllowedInLoungeRequirement requirement)
    {
        if(context.User.HasClaim(c => c.Type == "EmployeeNumber"))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}
```

Проверяет, есть ли у пользователя утверждение EmployeeNumber

Обработчик реализует AuthorizationHandler<T>

Вы должны переопределить абстрактный метод HandleRequirementAsync

Если у пользователя было необходимо утверждение, помечаем его как удовлетворенное, вызывая метод Succeed

Если требование не было выполнено, ничего не предпринимаем

Я оставил реализацию обработчика `MinimumAgeHandler` для требования `MinimumAgeRequirement` вам в качестве упражнения. Вы можете найти реализацию в примерах кода для этой главы.

СОВЕТ Можно писать очень обобщенные обработчики, которые могут быть использованы с несколькими требованиями, но я предлагаю придерживаться только одного требования. Если вам нужно извлечь какую-то общую функциональность, переместите ее во внешний сервис и выполните вызов из обоих обработчиков.

Это распространенный шаблон обработчика авторизации, но в некоторых случаях, вместо того чтобы проверить, насколько все было успешно, у вас может возникнуть желание проверить, насколько все было неудачно. Например, если вернуться к аэропорту, вы не хотите авторизовать человека, которому ранее запретили вход в зал ожидания, даже если в противном случае ему разрешили бы войти. Для этого можно использовать метод `context.Fail()`, предоставляемый в контексте, как показано в следующем листинге. Вызов метода `Fail()` в обработчике всегда будет приводить к тому, что требование, а следовательно, и вся политика не будут удовлетворены. Вы должны использовать его только тогда, когда хотите гарантировать сбой, даже если другие обработчики выполняются успешно.

Листинг 24.10 Вызов метода context.Fail() в обработчике, чтобы не выполнять требование

```

public class BannedFromLoungeHandler : AuthorizationHandler<AllowedInLoungeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        AllowedInLoungeRequirement requirement)
    {
        if(context.User.HasClaim(c => c.Type == "IsBannedFromLounge"))
        {
            context.Fail();
        }
        return Task.CompletedTask;
    }
}

```

Обработчик реализует AuthorizationHandler<T>

Вы должны переопределить абстрактный метод HandleRequirementAsync

Проверяет, есть ли у пользователя утверждение IsBannedFromLounge

Если утверждение не найдено, ничего не предпринимаем

Если у пользователя есть утверждение, отмечаем требование как неудовлетворенное, вызывая метод Fail. Вся политика не будет удовлетворена

В большинстве случаев обработчики либо вызывают метод `Succeed()`, либо ничего не делают, но метод `Fail()` полезен, когда вам нужен аварийный выключатель, чтобы гарантировать, что требование не будет удовлетворено.

ПРИМЕЧАНИЕ Независимо от того, вызывает обработчик методы `Succeed()` или `Fail()` либо ни то, ни другое, система авторизации всегда будет выполнять все обработчики для требования и все требования к политике, поэтому можете быть уверены, что обработчики всегда будут вызываться.

Последний шаг для завершения реализации авторизации в приложении – это регистрация обработчиков авторизации в контейнере внедрения зависимостей, как показано в следующем листинге.

Листинг 24.11 Регистрация обработчиков авторизации в контейнере внедрения зависимостей

```

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization(options =>

    options.AddPolicy(
        "CanEnterSecurity",
        policyBuilder => policyBuilder
            .RequireClaim(Claims.BookingPassNumber));
    options.AddPolicy(
        "CanAccessLounge",
        policyBuilder => policyBuilder.AddRequirements(
            new MinimumAgeRequirement(18),
            new AllowedInLoungeRequirement()
        ));
});

```

```
services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
services.AddSingleton<IAuthorizationHandler, FrequentFlyerHandler>();
services.AddSingleton<IAuthorizationHandler, BannedFromLoungeHandler>();
services.AddSingleton<IAuthorizationHandler,
IsAirlineEmployeeHandler>();
// Дополнительная конфигурация сервиса;
```

Для этого приложения у обработчиков нет зависимостей, внедряемых в конструктор, поэтому я зарегистрировал их в контейнере как объекты-одиночки. Если у ваших обработчиков есть зависимости с жизненным циклом Scoped или Transient (например, класс EF Core, DbContext), то вы можете зарегистрировать их как Scoped.

ПРИМЕЧАНИЕ Сервисы регистрируются с жизненным циклом трех типов: transient, scoped или singleton, как описано в главе 9.

Вы можете комбинировать концепции политик, требований и обработчиков разными способами для достижения своих целей по авторизации в приложении. Пример, приведенный в этом разделе, хотя и надуманный, демонстрирует каждый из необходимых вам компонентов, чтобы декларативно применить авторизацию на уровне метода действия или страницы Razor, создавая политики и при необходимости применения атрибут [Authorize]. Помимо явного применения атрибута [Authorize] к действиям и страницам Razor, вы также можете сконфигурировать его глобально, чтобы политика применялась к каждой странице Razor или контроллеру в приложении. Кроме того, для Razor Pages можно применять разные политики авторизации к разным папкам. Дополнительную информацию о применении политик авторизации с использованием соглашений см. в статье Microsoft: <http://mng.bz/nMm2>. Однако есть одна область, где атрибут [Authorize] не работает: авторизация на основе ресурсов. Атрибут [Authorize] добавляет метаданные к конечной точке, поэтому компонент авторизации может авторизовать пользователя *до* выполнения конечной точки, но что, если вам нужно авторизовать действие *во время* метода действия или обработчика страницы Razor? Применение авторизации на уровне документа или ресурса – обычное явление. Если пользователям разрешено редактировать только те документы, которые они создали, то вам необходимо загрузить документ, прежде чем вы узнаете, разрешено ли им редактировать его! С декларативным подходом, в котором используется атрибут [Authorize], сделать это непросто, поэтому нужно воспользоваться алтернативой – императивным подходом. В следующем разделе вы увидите, как применить авторизацию на основе ресурсов в обработчике страниц Razor.

24.5 Управление доступом с авторизацией на основе ресурсов

В этом разделе вы узнаете об авторизации на основе ресурсов. Она используется, когда вам необходимо узнать подробности о защищаемом ресурсе, чтобы определить, авторизован ли пользователь. Вы узнаете,

как применять политики авторизации вручную с помощью интерфейса `IAuthorizationService` и создавать обработчики `AuthorizationHandler` на основе ресурсов.

Авторизация на основе ресурсов – распространенная задача для приложений, особенно когда у вас есть пользователи, которые могут создавать или редактировать какие-то документы. Рассмотрим приложение с рецептами, которое мы создали в предыдущих трех главах. Оно позволяет пользователям создавать, просматривать и редактировать рецепты.

До настоящего момента каждый может создавать новые рецепты и отредактировать любой рецепт, даже если не выполнил вход в приложение. Теперь вы хотите добавить дополнительное поведение:

- только аутентифицированные пользователи должны иметь возможность создавать новые рецепты;
- вы можете редактировать лишь созданные вами рецепты.

Вы уже видели, как выполнить первое из этих требований: декорировать страницу `Create.cshtml` атрибутом `[Authorize]` без указания политики, как показано в этом листинге. Это заставит пользователя пройти аутентификацию, прежде чем он сможет создать новый рецепт.

Листинг 24.12 Добавление атрибута `Authorize` на страницу `Create.cshtml`

```
[Authorize]
public class CreateModel : PageModel
{
    [BindProperty]
    public CreateRecipeCommand Input { get; set; }

    public void OnGet()
    {
        Input = new CreateRecipeCommand();
    }

    public async Task<IActionResult> OnPost()
    {
        // Тело метода не показано для краткости;
    }
}
```

Пользователи должны пройти аутентификацию для выполнения страницы Razor, `Create.cshtml`

Все обработчики страницы защищены. Вы можете применить атрибут `[Authorize]` только к `PageModel`, но не к обработчикам

СОВЕТ Как и в случае со всеми фильтрами, атрибут `[Authorize]` можно применить только к странице Razor Page, а не к отдельным обработчикам страниц. Атрибут применяется ко всем обработчикам страницы Razor.

Добавление атрибута `[Authorize]` отвечает вашему первому требованию, но, к сожалению, используя техники, которые были показаны до сих пор, у вас нет возможности удовлетворить второе требование. Можно было бы применить политику, которая разрешает или запрещает пользователю редактировать *все* рецепты, но в настоящее время нельзя просто сделать так, чтобы пользователь мог редактировать только *собственные* рецепты. Чтобы узнать, кто создал рецепт, сначала

нужно загрузить его из базы данных. Только после этого можно попытаться авторизовать пользователя, принимая во внимание конкретный рецепт (ресурс). В следующем листинге показан частично реализованный обработчик страницы, демонстрирующий, как может выглядеть авторизация, которая происходит после загрузки объекта Recipe.

Листинг 24.13 Страница Edit.cshtml должна загрузить объект Recipe перед авторизацией запроса

```
public IActionResult OnGet(int id)
{
    var recipe = _service.GetRecipe(id);
    var createdById = recipe.CreatedById;
    // Авторизуем пользователя на основе createdById; <|
    if(isAuthorized)
    {
        return View(recipe); | Метод действия может
    } | продолжить работу
} | только в том случае,
    | если пользователь ав-
    | торизован
    | Вы должны загрузить сущность
    | Recipe из базы данных, прежде
    | чем узнаете, кто ее создал
    | Вы должны автори-
    | зовать текуще-
    | го пользователя,
    | чтобы убедиться,
    | что ему разреше-
    | но редактировать
    | этот конкретный
    | рецепт
```

Идентификатор редактируемого рецепта предоставляется привязкой модели

Вам необходим доступ к ресурсу (в данном случае сущности Recipe) для выполнения авторизации, поэтому декларативный атрибут [Authorize] вам не поможет. В разделе 24.5.1 вы увидите подход, которым нужно воспользоваться, чтобы справиться с этими ситуациями и применить авторизацию внутри метода действия или страницы Razor.

ВНИМАНИЕ! Будьте осторожны при предоставлении целочисленного идентификатора сущностей в URL-адресе, как показано в листинге 24.13. Пользователи смогут редактировать любую сущность, изменяя идентификатор в URL-адресе для доступа к другой сущности. Обязательно примените проверку авторизации, или вы рискуете столкнуться с уязвимостью под названием *небезопасные прямые ссылки на объекты* (IDOR). Подробнее об IDOR можно прочитать на странице <http://mng.bz/QPnG>.

24.5.1 Ручная авторизация запросов с помощью *IAuthService*

До сих пор все подходы к авторизации были *декларативными*. Вы применяете атрибут [Authorize] с именем политики или без него и позволяете фреймворку брать на себя заботу о выполнении самой авторизации. В примере с редактированием рецепта необходимо использовать *императивную* авторизацию, чтобы авторизовать пользователя после загрузки объекта Recipe из базы данных. Вместо того чтобы применять маркер, гласящий «Авторизуйтесь для выполнения этого метода», нужно самостоятельно написать код авторизации.

ОПРЕДЕЛЕНИЕ *Декларативное и императивное программирование* – два разных стиля программирования. *Декларативное программирова-*

ние описывает то, чего вы пытаетесь достичь, и позволяет фреймворку выяснить, как этого добиться. *Императивное программирование* описывает, как сделать это, предоставляя все необходимые шаги.

ASP.NET Core предоставляет интерфейс `IAuthorizationService`, который вы можете внедрить в страницы Razor и контроллеры для императивной авторизации. В следующем листинге показано, как обновить страницу `Edit.cshtml` (она частично показана в листинге 24.13), чтобы использовать интерфейс `IAuthorizationService` и проверить, разрешено ли действию продолжить выполнение.

Листинг 24.14 Использование интерфейса `IAuthorizationService` для авторизации на основе ресурсов

```
[Authorize]
public class EditModel : PageModel
{
    [BindProperty]
    public Recipe Recipe { get; set; }

    private readonly RecipeService _service;
    private readonly IAuthorizationService _authService;

    public EditModel(
        RecipeService service,
        IAuthorizationService authService)
    {
        _service = service;
        _authService = authService;
    }

    public async Task<IActionResult> OnGet(int id)
    {
        Recipe = _service.GetRecipe(id);
        AuthorizationResult authResult = await _authService
            .AuthorizeAsync(User, Recipe, "CanManageRecipe");
        if (!authResult.Succeeded)
        {
            return new ForbidResult();
        }
        return Page();
    }
}
```

Загружает рецепт из базы данных

Только авторизованные пользователи должны иметь право редактировать рецепты

IAuthorizationService внедряется в конструктор класса, используя DI

Вызывает `IAuthorizationService`, предоставляя `ClaimsPrincipal`, ресурс и имя политики

Если авторизация прошла неудачно, возвращается результат `Forbidden`

Если авторизация прошла успешно, продолжает отображение страницы Razor

`IAuthorizationService` предоставляет метод `AuthorizeAsync`, которому требуются три вещи для авторизации запроса:

- пользовательский объект `ClaimsPrincipal`, предоставленный в `PageModel` как `User`;
- авторизуемый ресурс: `Recipe`;
- политика, которую нужно проверить: `"CanManageRecipe"`.

Попытка авторизации возвращает объект `AuthorizationResult`, который указывает на то, была ли попытка успешной, через свойство `Succeeded`. Если попытка была неудачной, вы должны вернуть новый объект `ForbidResult`, который будет либо преобразован в ответ `403 Forbidden`, либо перенаправит пользователя на страницу с надписью «Отказано в доступе», в зависимости от того, создаете вы традиционное веб-приложение или веб-API.

ПРИМЕЧАНИЕ Как упоминалось в разделе 24.2.2, генерируемый тип ответа зависит от того, какие сервисы аутентификации настроены. Конфигурация Identity по умолчанию, используемая Razor Pages, генерирует перенаправления. Веб-API приложения обычно генерируют ответы с кодами состояния `401` и `403`.

Вы настроили императивную авторизацию на самой странице `Edit.cshtml`, но вам все равно нужно определить политику `"CanManageRecipe"`, которую вы используете для авторизации пользователя. Этот процесс выглядит так же, как и для декларативной авторизации, поэтому необходимо сделать следующее:

- создайте политику в `ConfigureServices`, вызвав метод `AddAuthorization()`;
- определите одно или несколько требований политики;
- определите один или несколько *обработчиков* для каждого требования;
- зарегистрируйте обработчики в контейнере внедрения зависимостей.

За исключением обработчика, все эти шаги идентичны подходу с декларативной авторизацией с атрибутом `[Authorize]`, поэтому я только быстро пробегусь по ним. Во-первых, вы можете создать простое требование `IAuthorizationRequirement`. Как и многие другие требования, оно не содержит данных и просто реализует интерфейс-маркер.

```
public class IsRecipeOwnerRequirement : IAuthorizationRequirement { }
```

Определить политику в `ConfigureServices` так же просто, поскольку у вас есть только одно требование. Обратите внимание, что в этом коде пока нет ничего конкретного для ресурсов:

```
builder.Services.AddAuthorization(options => {
    options.AddPolicy("CanManageRecipe", policyBuilder =>
        policyBuilder.AddRequirements(new IsRecipeOwnerRequirement());
});
```

Мы уже прошли половину пути; все, что вам нужно сделать сейчас, – это создать обработчик авторизации для `IsRecipeOwnerRequirement` и зарегистрировать его в контейнере внедрения зависимостей.

24.5.2 Создание обработчика AuthorizationHandler на основе ресурсов

Обработчики авторизации на основе ресурсов, по сути, такие же, как и реализации обработчиков, которые вы видели в разделе 24.4.2. Единственная разница состоит в том, что обработчик также имеет доступ к авторизуемому ресурсу. Чтобы создать обработчик на основе ресурсов, вы должны унаследовать его от базового класса `AuthorizationHandler<TRequirement, TResource>`, где `TRequirement` – это тип требования для обработки, а `TResource` – это тип ресурса, который вы предоставляете, когда вызываете `IAuthorizationService`. Сравните его с классом `AuthorizationHandler<T>`, реализованным ранее, где вы указывали только тип требования. В этом листинге показана реализация обработчика для приложения с рецептами. Как видите, мы указали требование как `IsRecipeOwnerRequirement`, а ресурс как `Recipe` и реализовали метод `HandleRequirementAsync`.

Листинг 24.15 Класс IsRecipeOwnerHandler для авторизации на основе ресурсов

```
public class IsRecipeOwnerHandler : AuthorizationHandler<IsRecipeOwnerRequirement, Recipe>
{
    private readonly UserManager<ApplicationUser> _userManager;
    public IsRecipeOwnerHandler(
        UserManager<ApplicationUser> userManager)
    {
        _userManager = userManager;
    }
    protected override async Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        IsRecipeOwnerRequirement requirement,
        Recipe resource)
    {
        var appUser = await _userManager.GetUserAsync(context.User);
        if(appUser == null) ←
        {
            return; ←
        }
        if(resource.CreatedById == appUser.Id) ←
        {
            context.Succeed(requirement); ←
        }
    }
}
```

Реализует необходимый базовый класс с указанием требования и типа ресурса

Внедряет экземпляр класса `UserManager<T>` с помощью DI

Помимо контекста и требования, вам также предоставляется экземпляр ресурса

Если вы не прошли аутентификацию, `appUser` будет иметь значение `null`

Проверяет, создал ли текущий пользователь рецепт, проверяя свойство `CreatedById`

Если пользователь создал рецепт, использует метод `Succeed`; в противном случае ничего не делает

Этот обработчик немного сложнее, чем те примеры, которые вы видели ранее, прежде всего потому, что вы используете дополнительный сервис `UserManager<>` для загрузки сущности `ApplicationUser` на основе объекта `ClaimsPrincipal` из запроса.

ПРИМЕЧАНИЕ На практике в ClaimsPrincipal, скорее всего, уже будет добавлен идентификатор в качестве утверждения, что делает лишним дополнительный шаг в данном случае. Этот пример показывает общий шаблон, если вам нужно использовать сервисы с внедрением зависимостей.

Другое существенное отличие состоит в том, что метод HandleRequirementAsync предоставляет ресурс Recipe в качестве аргумента метода. Это тот же объект, который вы предоставляли при вызове метода AuthorizeAsync в IAuthorizationService. Вы можете использовать этот ресурс, чтобы проверить, был ли он создан текущим пользователем. Если это так, то используется метод Succeed(); в противном случае вы ничего не делаете.

Последняя задача – добавить обработчик IsRecipeOwnerHandler в контейнер внедрения зависимостей. Ваш обработчик использует дополнительную зависимость UserManager<>, которая применяет EF Core, поэтому следует зарегистрировать обработчик как сервис с жизненным циклом Scoped:

```
services.AddScoped<IAuthorizationHandler, IsRecipeOwnerHandler>();
```

СОВЕТ Если вам интересно, как узнать, с каким жизненным циклом нужно зарегистрировать обработчик: Scoped или Singleton, – вспомните главу 9. По сути, если у вас есть зависимости с жизненным циклом Scoped, то вы должны зарегистрировать обработчик как scoped; в противном случае будет singleton.

Теперь, когда все части на своих местах, можно опробовать приложение. Если вы попытаетесь отредактировать рецепт, который не создавали, нажав кнопку «Изменить», то либо будете перенаправлены на страницу входа (если еще не прошли аутентификацию), либо увидите страницу «Отказано в доступе», как показано на рис. 24.7. Используя авторизацию на основе ресурсов, вы можете вводить более детальные требования авторизации, которые можно применять на уровне отдельного документа или ресурса. Вместо того чтобы разрешать пользователю редактировать либо *любой* рецепт, либо никакой, можно разрешить пользователю редактировать *определенный* рецепт.

Все техники авторизации, которые вы видели до сих пор, были сосредоточены на проверке на стороне сервера. И атрибут [Authorize], и авторизация на основе ресурсов фокусируются на запрете пользователям выполнять защищенное действие на сервере. Это важно с точки зрения безопасности, но есть еще один аспект, который следует учитывать: опыт взаимодействия пользователя с сайтом, когда у этого пользователя нет полномочий. Вы защитили код, выполняющийся на сервере, но, возможно, кнопку «Изменить» не стоит показывать пользователям, если вы не собираетесь разрешать им редактировать рецепт! В следующем разделе мы рассмотрим, как можно условно скрыть ее с помощью авторизации на основе ресурсов в моделях представления.

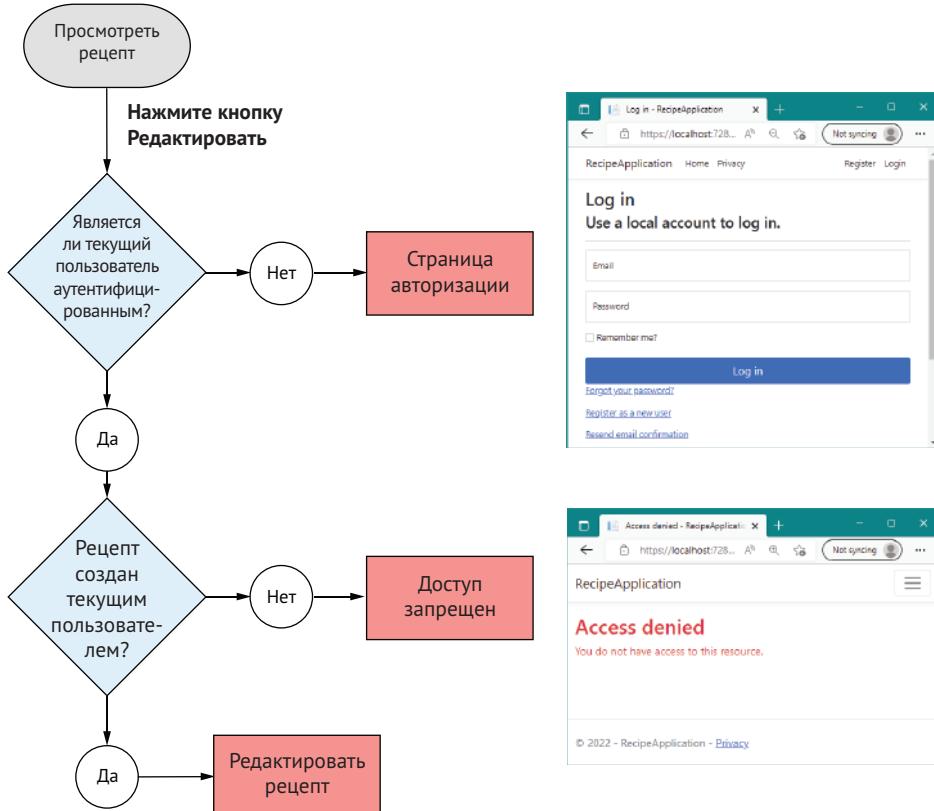


Рис. 24.7 Если вы выполнили вход в приложение, но не авторизованы для редактирования рецепта, то будете перенаправлены на страницу «Отказано в доступе». Если вы не выполнили вход, то будете перенаправлены на страницу входа

Авторизация на основе ресурсов или проверка бизнес-логики

Ценность использования подхода с авторизацией на основе ресурсов в ASP.NET Core не всегда очевидна по сравнению с использованием простой проверки бизнес-логики, осуществляющейся вручную (как в листинге 24.13). Использование интерфейса `IAuthorizationService` и инфраструктуры авторизации добавляет явную зависимость от фреймворка ASP.NET Core, которую вы, возможно, не захотите использовать, если выполняете проверку авторизации в сервисах модели предметной области.

Это серьезная проблема, для которой нет простого решения. Я предлагаю простую проверку в бизнес-логике внутри предметной области, не полагаясь на инфраструктуру авторизации фреймворка, чтобы сделать свою предметную область более простой для тестирования и независимой от фреймворка. Но при этом теряются некоторые преимущества такого фреймворка:

- `IAuthorizationService` использует декларативные политики, даже если вы императивно вызываете фреймворк авторизации;
- вы можете отделить необходимость авторизации действия от фактических требований;
- вы можете с легкостью полагаться на периферийные сервисы и свойства запроса, что может быть сложнее (или нежелательно) при проверке в бизнес-логике.

Эти преимущества можно получить с помощью проверок в бизнес-логике, но обычно для этого требуется создание большой инфраструктуры, поэтому вы теряете много преимуществ простоты реализации. Какой подход лучше? Это зависит от специфики дизайна вашего приложения, и вполне могут быть случаи, когда можно будет использовать оба варианта.

Например, одним из возможных подходов является использование атрибута `[Authorize]`, как описано в разделе 24.2.1 для предотвращения анонимного доступа к вашим API, возможно с простыми политиками, примененными к API. А все прочие проверки доступа, требуемые в вашей предметной области, выносятся в проверки `ClaimsPrincipal` на уровне бизнес-логики. Это может существенно уменьшить сложность работы с подсистемой авторизации ASP.NET Core.

24.6 Скрытие HTML-элементов от неавторизованных пользователей

Весь код авторизации, который вы видели до сих пор, был связан с защитой методов действий или страниц Razor Pages на стороне сервера, а не изменением пользовательского интерфейса для пользователей. Это важно и должно быть отправной точкой всякий раз, когда вы добавляете авторизацию в приложение.

ВНИМАНИЕ! Злоумышленники могут с легкостью обойти ваш пользовательский интерфейс, поэтому всегда важно авторизовывать методы действия и страницы Razor Pages на сервере, а не только на стороне клиента.

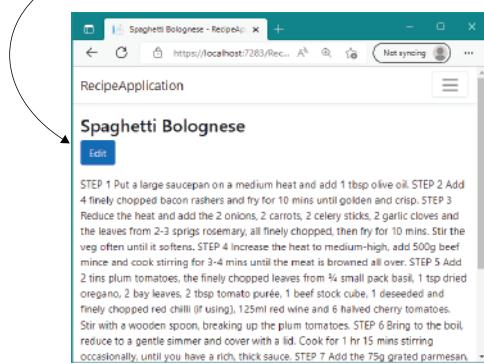
Однако с точки зрения опыта взаимодействия пользователя с сайтом наличие кнопок или ссылок, которые выглядят так, будто они доступны, но отображают сообщение «Отказ в доступе», когда на них нажимают, не совсем удобно. Лучше отключать эти ссылки или сделать так, чтобы их вообще не было видно.

Этого можно добиться несколькими способами в ваших шаблонах Razor. В данном разделе я покажу вам, как добавить дополнительное свойство в модель страницы: `CanEditRecipe`. Шаблон представления Razor будет использовать его для изменения визуализированного HTML.

СОВЕТ В качестве альтернативы можно было бы внедрить интерфейс `IAuthorizationService` непосредственно в шаблон представления с помощью директивы `@inject`, как было показано в главе 9, но предпочтительнее хранить подобную логику в обработчике страницы.

Когда вы закончите, визуализированный HTML созданных вами рецептов останется без изменений, но кнопка «Изменить» будет скрыта при просмотре рецепта, созданного кем-то другим, как показано на рис. 24.8.

Если пользователь создавал рецепт, он может видеть кнопку Редактировать



Если рецепт создан другим пользователем, кнопка Редактировать скрыта



Рис. 24.8 Несмотря на то что HTML созданных вами рецептов будет отображаться без изменений, кнопка «Изменить» будет скрыта при просмотре рецептов, созданных другим пользователем

В следующем листинге показана модель страницы RazorView.cshtml, которая используется для отрисовки страницы рецептов, показанной на рис. 24.8. Как вы уже видели в случае с авторизацией на основе ресурсов, вы можете использовать интерфейс IAuthorizationService, чтобы определить, есть ли у текущего пользователя полномочия на редактирование рецепта путем вызова метода AuthorizeAsync. Затем вы можете задать это значение в качестве дополнительного свойства модели страницы CanEditRecipe.

Листинг 24.16 Задаем свойство CanEditRecipe на странице View.cshtml

```
public class ViewModel : PageModel
{
    public Recipe Recipe { get; set; }
    public bool CanEditRecipe { get; set; } ← Свойство CanEditRecipe будет
                                              использоваться для
                                              управления отрисов-
                                              кой кнопки «Изменить»

    private readonly RecipeService _service;
    private readonly IAuthorizationService _authService;
    public ViewModel(
        RecipeService service,
        IAuthorizationService authService)
    {
        _service = service;
        _authService = authService;
    }
}
```

Свойство CanEditRecipe будет использоваться для управления отрисовкой кнопки «Изменить»

```

public async Task<IActionResult> OnGetAsync(int id)
{
    Recipe = _service.GetRecipe(id); ← Загружает ресурс Recipe
    AuthorizationResult isAuthorised = await _authService ← для использования с
        .AuthorizeAsync(User, recipe, "CanManageRecipe");
    CanEditRecipe = isAuthorised.Succeeded; ← IAuthorizationService
    return Page();   Задает свойство CanEditRecipe
}                                модели страницы соответствую-
}                                щим образом
}

```

Проверяет, имеет ли пользователь право редактировать рецепт

Вместо того чтобы блокировать выполнение страницы Razor (как вы это делали ранее в обработчике страницы Edit.cshtml), используйте результат вызова метода `AuthorizeAsync`, чтобы задать значение `CanEditRecipe` в модели страницы. Затем можно просто изменить шаблон View.cshtml: добавьте предложение `if` вокруг отрисовки ссылки «Изменить».

```

@if(Model.CanEditRecipe)
{
    <a asp-page="Edit" asp-route-id="@Model.Id"
        class="btn btn-primary">Edit</a>
}

```

Это гарантирует, что только пользователи, которые могут выполнить страницу Edit.cshtml, смогут увидеть ссылку на эту страницу.

ВНИМАНИЕ! Предложение `if` означает, что ссылка «Изменить» не будет отображаться, если только пользователь не создавал этот рецепт, но злоумышленник по-прежнему может обойти пользовательский интерфейс. Важно сохранить проверку авторизации на стороне сервера в вашем обработчике страницы Edit.cshtml, чтобы защитить себя от попыток взлома.

Выполнив это последнее изменение, мы завершили добавление авторизации в приложение рецептов. Анонимные пользователи могут просматривать рецепты, созданные другими лицами, но они должны выполнить вход, чтобы создавать новые рецепты. Кроме того, аутентифицированные пользователи могут редактировать только те рецепты, которые они создали, и они не увидят кнопку «Изменить» в рецептах других пользователей. Авторизация – ключевой аспект большинства приложений, поэтому важно помнить о ней с самого начала. Хотя авторизацию можно добавить позже, как мы это сделали в приложении с рецептами, обычно предпочтительнее заняться данным вопросом до разработки приложения.

В главах 23 и 24 мы сосредоточились на аутентификации и авторизации для традиционных веб-приложений с использованием Razor. В главе 25 мы рассмотрим приложения API, как работает аутентификация с помощью токенов и как добавить политики авторизации в минимальные API.

Резюме

- Аутентификация – это процесс определения пользователя. Он отличается от авторизации, когда вы определяете, что может делать пользователь. Обычно аутентификация происходит до авторизации;
- вы можете использовать сервисы авторизации в любой части приложения, но обычно применяется компонент `AuthorizationMiddleware` путем вызова метода `UseAuthorization()`. Для правильной работы его следует размещать после вызова методов `UseRouting()` и `UseAuthentication()` и перед вызовом метода `UseEndpoints()`;
- вы можете защитить страницы Razor Pages и действия MVC, применяя атрибут `[Authorize]`. Компонент маршрутизации записывает присутствие атрибута в качестве метаданных с выбранной конечной точкой. Компонент авторизации использует эти метаданные, чтобы определить, как авторизовать запрос;
- самая простая форма авторизации требует, чтобы пользователь прошел аутентификацию до того, как выполнит действие. Для этого можно применить атрибут `[Authorize]` на странице Razor к действию, контроллеру или глобально. Вы также можете применять атрибуты обычным образом к подмножеству страниц Razor Pages;
- авторизация на основе утверждений использует утверждения текущего пользователя, чтобы определить, имеет ли он право выполнить действие. Утверждения, необходимые для выполнения действия, определяются в *политике*;
- у политик есть имена. Они настраиваются в файле `Startup.cs` как часть вызова метода `AddAuthorization()` в методе `ConfigureServices`. Политика определяется с помощью метода `AddPolicy()`, путем передачи имени и лямбда-функции, определяющей необходимые утверждения;
- можно применить политику к действию или странице Razor, указав политику в атрибуте `[Authorize]`, например `[Authorize("CanAccessLounge")]`. Эта политика будет использоваться `AuthorizationMiddleware`, чтобы определить, позволено ли пользователю выполнить выбранную конечную точку;
- в приложении Razor Pages, если не прошедший аутентификацию пользователь пытается выполнить защищенное действие, он будет перенаправлен на страницу входа. Если он уже прошел аутентификацию, но у него нет необходимых утверждений, то он увидит страницу с надписью «Отказано в доступе»;
- для сложных политик авторизации можно создать специальную политику. Такая политика состоит из одного или нескольких требований, а у требования может быть один или несколько обработчиков. Можно сочетать требования и обработчики для создания политик произвольной сложности;
- чтобы политика была удовлетворена, необходимо отвечать всем

требованиям. Для этого один или несколько ассоциированных обработчиков должны указывать на успех, и ни один из них не должен явно указывать на сбой;

- `AuthorizationHandler<T>` содержит логику, определяющую, удовлетворено ли требование. Например, если требуется, чтобы пользователи были старше 18 лет, обработчик может найти утверждение `DateOfBirth` и вычислить возраст пользователя;
- обработчики могут пометить требование как удовлетворенное, вызвав метод `context.Succeed(requirement)`. Если обработчик не может удовлетворить требование, он не должен ничего вызывать в контексте, поскольку другой обработчик может вызвать метод `Succeed()` и удовлетворить требование;
- если обработчик вызывает метод `context.Fail()`, требование не удовлетворяется, даже если другой обработчик пометил его как успешное с помощью метода `Succeed()`. Используйте этот метод, только если хотите переопределить все вызовы `Succeed()` от других обработчиков, чтобы гарантировать, что политика не будет удовлетворена;
- авторизация на основе ресурсов использует сведения о защищаемом ресурсе, чтобы определить, авторизован ли текущий пользователь. Например, если пользователю разрешено редактировать только собственные документы, необходимо знать автора документа, прежде чем вы сможете определить, авторизован ли он;
- авторизация на основе ресурсов использует ту же политику «требование + обработчик», что и раньше. Вместо того чтобы применять авторизацию с атрибутом `[Authorize]`, вы должны вручную вызвать `IAuthorizationService` и предоставить ресурс, который вы защищаете;
- вы можете изменить пользовательский интерфейс с учетом прав пользователя, добавив дополнительные свойства в модель страницы. Если пользователь не авторизован для выполнения действия, вы можете удалить или отключить ссылку на этот метод действия в пользовательском интерфейсе. Всегда следует проходить авторизацию на сервере, даже если вы удалили ссылки из пользовательского интерфейса.

25

Аутентификация и авторизация для API

В этой главе:

- знакомство с работой аутентификации для API в ASP.NET Core;
- использование токенов на предъявителя для аутентификации;
- локальное тестирование API с помощью веб-токенов JSON;
- применение политик авторизации к минимальным API.

В главе 23 вы узнали, как работает аутентификация с традиционными веб-приложениями, например с теми, которые создаются с помощью Razor Pages или контроллеров MVC. Традиционные веб-приложения обычно используют зашифрованные файлы cookie для хранения личности пользователя для запроса, которые затем декодируются в `AuthenticationMiddleware`. В этой главе вы узнаете, как работает аутентификация для приложений API, чем она отличается от традиционных веб-приложений и какие опции доступны.

Мы начнем с общего рассмотрения того, как работает аутентификация для API, как изолированно, так и когда они являются частью более крупного приложения или распределенной системы. Вы узнаете о некоторых задействованных протоколах, таких как OAuth 2.0 и OpenID Connect; шаблонах, которые вы можете использовать для защиты своих API; и токенах, используемых для управления доступом (обычно это веб-токены JSON, называемые JWT).

В разделе 25.3 вы узнаете, как применить эти знания на практике, добавив аутентификацию в приложение минимальных API с использованием JWT-токенов. В разделе 25.4 вы узнаете, как использовать интерфейс командной строки .NET для создания JWT для локального тестирования API.

.NET CLI хорошо подходит для генерации токенов, но нам нужен способ добавить этот токен в запрос. В частности, если мы используем определения OpenAPI и пользовательский интерфейс Swagger, как описано в главе 11, нам нужен способ сообщить Swagger о наших требованиях к аутентификации. В разделе 25.5 вы узнаете о параметрах конфигурации аутентификации для документов OpenAPI и способах использования пользовательского интерфейса Swagger для отправки аутентифицированных запросов к API.

Наконец, в разделе 25.6 я покажу, как применять политики авторизации к конечным точкам минимальных API, чтобы ограничить круг пользователей, которые могут вызывать ваши API. Концепции авторизации, о которых вы узнали в главе 24 для Razor Pages, такие же и для API, поэтому мы по-прежнему будем использовать утверждения, требования, обработчики и политики.

Мы начнем с рассмотрения того, как работает аутентификация, когда у вас есть приложение API. Многие концепции аутентификации аналогичны традиционным приложениям, но необходимость поддержки нескольких типов пользователей, традиционных, клиентских и мобильных приложений привела к несколько различным решениям.

25.1 Аутентификация для API и распределенных приложений

В этом разделе вы узнаете о процессе аутентификации для приложений API, о том, почему он обычно отличается от аутентификации для традиционных веб-приложений, а также о некоторых использующихся при этом общих шаблонах и протоколах.

25.1.1 Распространение аутентификации на несколько приложений

Я описал процесс аутентификации для традиционных веб-приложений в главе 23. Когда пользователь входит в приложение, вы устанавливаете зашифрованный файл cookie. Этот файл содержит сериализованную версию `ClaimsPrincipal` пользователя, включая его идентификатор и все связанные с ним утверждения. Когда вы выполняете второй запрос, браузер автоматически отправляет этот файл cookie. Затем `AuthenticationMiddleware` декодирует его, десериализует `ClaimsPrincipal` и задает текущего пользователя для запроса, как показано ранее на рис. 23.3 и воспроизведено на рис. 25.1.

Этот процесс особенно хорошо работает, когда у вас есть одно традиционное веб-приложение, которое выполняет всю работу. Прило-

жение отвечает за аутентификацию пользователей и управление ими, а также за обслуживание данных вашего приложения и выполнение бизнес-логики, как показано на рис. 25.2.

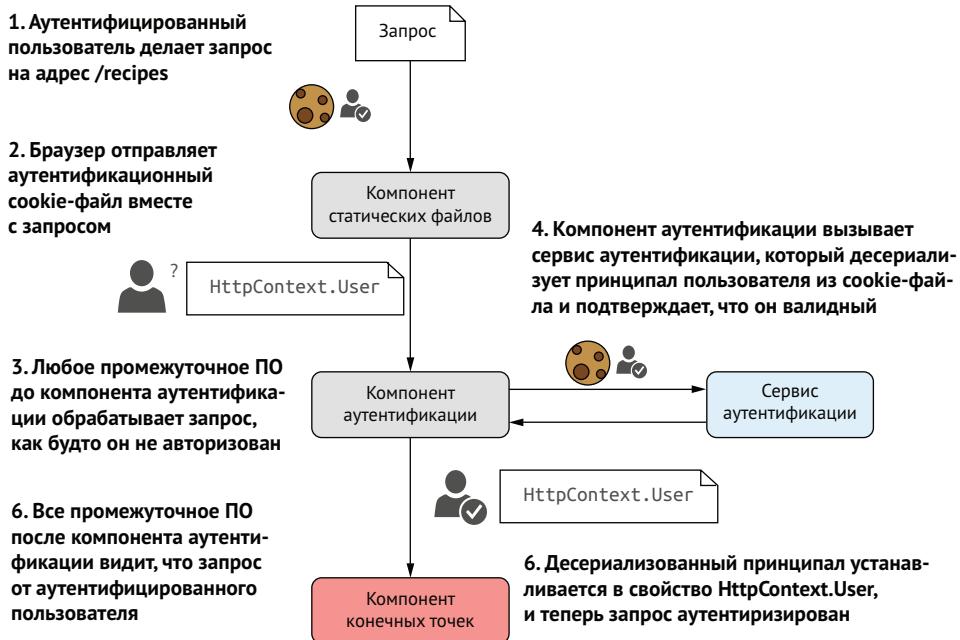


Рис. 25.1 Когда пользователь впервые входит в приложение, приложение устанавливает зашифрованный файл cookie, содержащий ClaimsPrincipal. При последующих запросах файл cookie, отправляемый вместе с запросом, содержит субъект пользователя, который десериализуется, проверяется и используется для аутентификации запроса

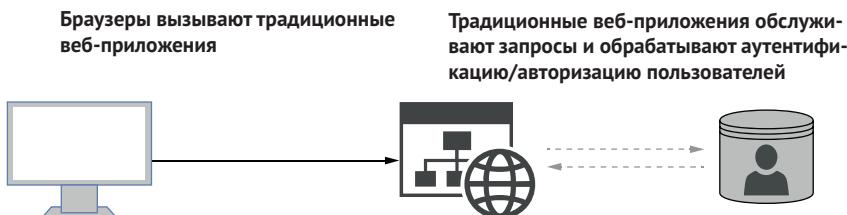


Рис. 25.2 Традиционные приложения обычно обрабатывают всю функциональность приложения: бизнес-логику, генерацию пользовательского интерфейса, аутентификацию и управление пользователями

Помимо традиционных веб-приложений, ASP.NET Core обычно используется в качестве API для обслуживания данных для мобильных и клиентских одностраничных приложений. Аналогичным образом даже традиционным веб-приложениям, использующим Razor Pages, часто приходится вызывать приложения API «за кулисами», как показано на рис. 25.3.

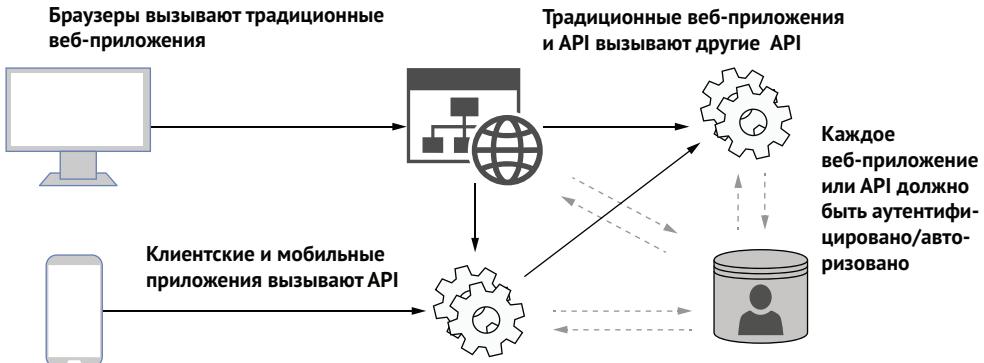


Рис. 25.3 Современные приложения обычно должны предоставлять веб-API для клиентских приложений и приложений для мобильных устройств, а также потенциально вызывать API для серверной части. Когда все эти сервисы нуждаются в аутентификации и управлении пользователями, это становится сложным с логистической точки зрения

В этой ситуации у вас есть несколько приложений и API, и все они должны понимать, что один и тот же пользователь отправляет запрос во все приложения и API. Если вы будете придерживаться того же подхода, что и раньше, когда каждое приложение управляет своими пользователями, все может быстро выйти из-под контроля! Вам также потребуется продублировать всю логику входа между приложениями и API, а еще понадобится иметь некую централизованную базу данных, содержащую данные пользователей. Пользователям может понадобиться выполнить вход несколько раз, чтобы получить доступ к разным частям приложения. Более того, использование файлов cookie становится проблемой, в частности для некоторых мобильных клиентов или в тех случаях, когда вы делаете запросы к нескольким доменам (поскольку файлы cookie принадлежат только одному домену). Как это улучшить? Перенести обязанности по аутентификации в отдельный сервис.

25.1.2 Централизация аутентификации в поставщике идентификационной информации

Современные системы часто имеют множество движущихся частей, каждая из которых требует определенного уровня аутентификации и авторизации для защиты каждого приложения от несанкционированного использования. Вместо того чтобы встраивать обязанности по аутентификации в каждое приложение, общий подход состоит в том, чтобы извлекать код, общий для всех приложений и API, а затем передавать его поставщику идентификационной информации, как показано на рис. 25.4.

Вместо того чтобы обеспечить вход в приложение напрямую, приложение перенаправляет пользователя к поставщику идентификационной информации. Пользователь выполняет вход в учетную запись этого поставщика, который передает токены носителя обратно клиенту (например, браузеру или мобильному приложению), что указывает, кто является пользователем и к чему ему разрешен доступ.

Клиенты и приложения могут передавать эти токены в API, чтобы предоставить информацию о выполнившем вход пользователе, без необходимости повторной аутентификации или управления пользователями непосредственно в API.

ОПРЕДЕЛЕНИЕ Токены на предъявителя (токены носителя, *bearer tokens*) – это строки, содержащие сведения об аутентификации пользователя или приложения. Они могут быть или не быть зашифрованы, но обычно подписываются, чтобы избежать подделки. JWT (JSON Web Token) – наиболее распространенный формат. Мы подробнее рассмотрим JWT в разделе 25.2.

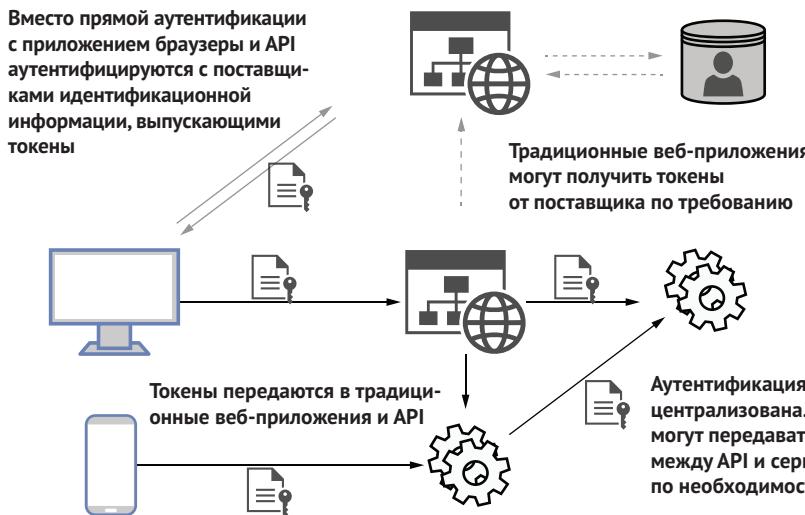


Рис. 25.4 Альтернативная архитектура предполагает использование центрального поставщика идентификационной информации для аутентификации и управления пользователями системы. Токены передаются туда и обратно между поставщиком, приложениями и API

Использование отдельного поставщика идентификационной информации на первый взгляд явно сложнее, поскольку вы добавляете совершенно новый сервис, но в долгосрочной перспективе это дает ряд преимуществ:

- *пользователи могут совместно использовать свою личность в нескольких сервисах.* Когда вы выполняете вход в централизованном поставщике идентификационной информации, то, по сути, входите во все приложения, использующие этот сервис. Это дает вам возможность входить только один раз, и не нужно совершать вход в несколько сервисов;
- *не нужно дублировать логику входа между несколькими сервисами.* Вся логика входа инкапсулирована в поставщике идентификационной информации, поэтому не нужно добавлять экраны входа во все свои приложения;

- *поставщик идентификационной информации имеет единственную ответственность.* Поставщик идентификационной информации отвечает только за аутентификацию и управление пользователями. Во многих случаях он довольно универсальный, поэтому вы можете (и должны!) использовать сторонний сервис идентификации, например Auth0 или Azure Active Directory, вместо того чтобы создавать собственную;
- *можно легко добавлять новых поставщиков.* Независимо от того, используете ли вы поставщика идентификационной информации или традиционный подход, можно использовать внешние сервисы для аутентификации пользователей. Вы будете встречать это в приложениях, которые позволяют «выполнить вход с помощью Facebook» или «с помощью Google», например. Если вы используете централизованного поставщика идентификационной информации, добавление поддержки дополнительных поставщиков может быть выполнено в одном месте. Не нужно явно конфигурировать каждое приложение и API.

Из коробки ASP.NET Core поддерживает такие архитектуры, а также потребление выпущенных токенов на предъявителя, но .NET 7 не включает в себя поддержку выпуска этих токенов. Это означает, что вам потребуется использовать другую библиотеку или сервис для поставщика идентификационной информации.

Как я упоминал в главе 23, отличным вариантом является использование стороннего поставщика идентификационной информации, например Facebook, Google, Okta, Auth0 или Azure Active Directory. Эти поставщики заботятся о хранении паролей пользователей, предоставляют возможности аутентификации с использованием современных стандартов, таких как WebAuthn (<https://webauthn.guide>), и выявляют злонамеренные попытки выдать себя за пользователя.

Используя поставщика, вы оставляете сложные вопросы безопасности экспертам и можете сосредоточиться на основной цели вашего бизнеса. Однако не все поставщики равны: профили некоторых из них (например, Auth0) принадлежат вам, а других (Facebook или Google) – нет. Обязательно выберите поставщика, который соответствует вашим требованиям.

СОВЕТ По возможности я рекомендую использовать стороннего поставщика идентификационной информации. У уважаемых поставщиков есть множество экспертов, которые работают исключительно над защитой данных ваших клиентов, активно предотвращают атаки и обеспечивают безопасность данных. Доверив эту сложную работу экспертам, вы можете сосредоточиться на основной деятельности приложения, какой бы она ни была.

Еще один распространенный вариант – создать собственного поставщика идентификационной информации. Может показаться, что это трудоемкий процесс (и так оно и есть!), но благодаря таким отличным биб-

лиотекам, как OpenIddict (<https://github.com/openiddict>) и IdentityServer от компании Duende Software (<https://duendesoftware.com>), вполне возможно написать собственного поставщика идентификационной информации для выпуска токенов на предъявителя, которые будут потребляться вашими приложениями и API.

ВНИМАНИЕ! Следует тщательно обдумать, оправданы ли усилия и риски, связанные с созданием собственного поставщика. Ошибки – это жизненный факт, и ошибка при выборе поставщика идентификационной информации может легко привести к уязвимостям в системе безопасности. Тем не менее если у вас особые требования к идентификационной информации, создание собственного поставщика может быть разумным или необходимым вариантом.

Есть один аспект, который часто упускают из виду те, кто начинает работать с OpenIddict и IdentityServer. Он заключается в том, что это не готовые решения. Они состоят из набора сервисов и промежуточного ПО, которые вы добавляете в стандартное приложение ASP.NET Core, обеспечивая реализацию соответствующих стандартов идентификации в соответствии со спецификацией. Вам как разработчику все равно необходимо писать код управления профилями, который знает, как создать нового пользователя (обычно в базе данных), загрузить данные пользователя, проверить его пароль и управлять связанными с ним утверждениями. Кроме того, необходимо предоставить пользователю весь код пользовательского интерфейса для входа в систему, управления его паролями и настройки многофакторной аутентификации. Это не для слабонервных!

Во многих отношениях можно рассматривать поставщика идентификационной информации как традиционное веб-приложение, имеющее только страницы управления учетными записями. Если вы хотите заняться созданием собственного поставщика, то ASP.NET Core Identity, о которой идет речь в главе 23, обеспечивает неплохую основу для управления пользователями. Добавление IdentityServer или OpenIddict дает вам возможность генерировать токены для других сервисов, используя стандарт OpenID Connect, для максимальной совместимости с другими сервисами.

25.1.3 OpenID Connect и OAuth 2.0

OpenID Connect (OIDC) (<http://openid.net/connect>) – это протокол аутентификации, созданный на основе спецификации OAuth 2.0 (<https://oauth.net/2>). Он разработан для упрощения подходов, описанных в разделе 25.1.2. Здесь вы перекладываете ответственность за хранение учетных данных пользователя кому-то другому (поставщику идентификационной информации). Он дает ответ на вопрос «Какой пользователь отправил этот запрос?» без необходимости самостоятельно управлять пользователем.

ПРИМЕЧАНИЕ Не обязательно понимать эти протоколы, чтобы добавить аутентификацию в свои API, но я думаю, что лучше иметь базовое представление о них, чтобы вы понимали, как ваши API вписываются в среду безопасности. Если вы хотите узнать больше об OpenID Connect, то в книге Прабата Сиривардены «OpenID Connect в действии» (Manning, 2023) вы найдете гораздо больше подробностей.

Open ID Connect построен на основе протокола OAuth 2.0, поэтому полезно сначала немного понять, что это за протокол. OAuth 2.0 – это протокол авторизации. Он позволяет пользователю контролируемым образом делегировать доступ к ресурсу другому сервису, не раскрывая каких-либо дополнительных данных, таких как ваша личность или любая другая информация.

Все это выглядит немного абстрактно, поэтому рассмотрим один пример. Вам нужно распечатать фотографии своей собаки с помощью сервиса печати фотографий, dogphotos.com. Вы регистрируетесь на сайте dogphotos.com, и вам предоставляется два варианта загрузки фото:

- загрузка со своего компьютера;
- загрузка напрямую с Facebook, используя протокол OAuth 2.0.

Поскольку вы используете новый ноутбук, вы не загрузили все фотографии своей собаки на свой компьютер, поэтому решили использовать OAuth 2.0, как показано на рис. 25.5. Это вызывает следующую последовательность действий:

- 1 Сайт dogphotos.com перенаправит вас на Facebook, где вам необходимо войти в учетную запись (если вы еще этого не сделали).
- 2 После того как вы пройдете аутентификацию, Facebook покажет вам экран согласия, на котором описаны данные, к которым хочет получить доступ сайт dogphotos.com. В данном случае это должны быть только ваши фотографии.
- 3 Когда вы нажмете «OK», Facebook автоматически перенаправит вас на сайт dogphotos.com и включит в URL-адрес код авторизации.
- 4 dogphotos.com использует этот код в сочетании с секретом, известным только Facebook и dogphotos.com, для получения токена доступа от Facebook.
- 5 Наконец, сайт dogphotos.com использует токен для вызова API Facebook и получения фотографий вашей собаки!

В этом примере происходит много всего, но зато вы получаете ряд приятных преимуществ:

- вам не нужно было предоставлять свои учетные данные Facebook на сайте dogphotos.com. Вы просто вошли в Facebook как обычно;
- вы могли контролировать, к каким данным сайт dogphotos.com может получить доступ от вашего имени через API фотографий на Facebook;
- вам не нужно было предоставлять сайту dogphotos.com какую-либо информацию о себе (хотя на практике это часто требуется).

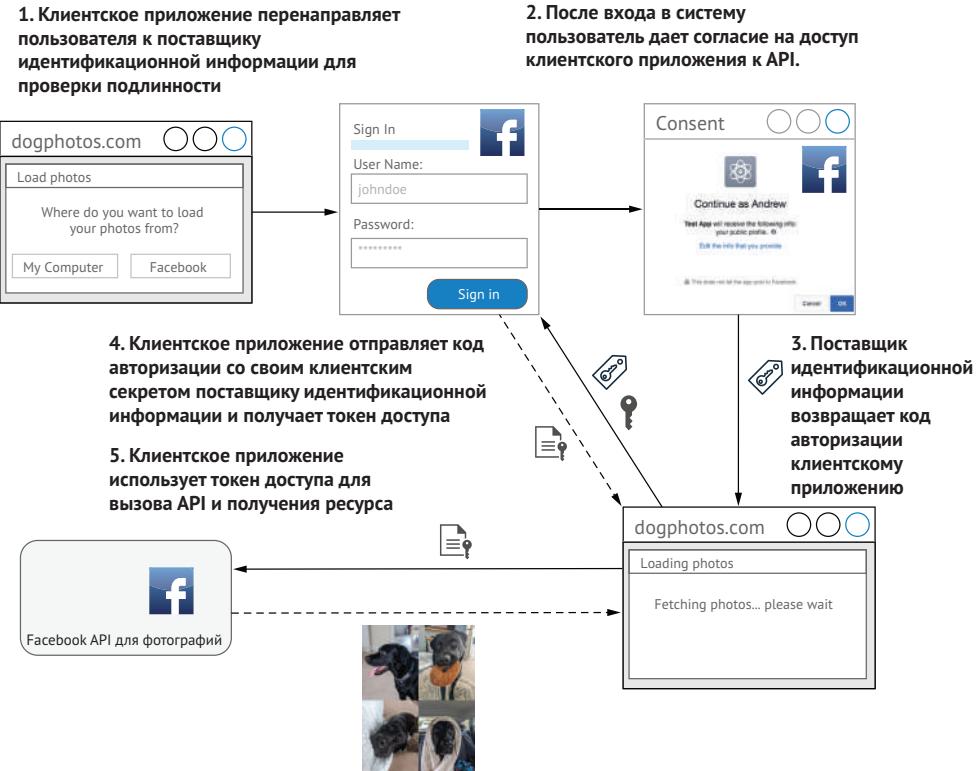


Рис. 25.5 Использование OAuth 2.0 для авторизации сайта dogphotos.com для доступа к вашим фотографиям на Facebook

По сути, вы делегировали доступ к API фотографий Facebook компании dogphotos.com. Именно по причине такого подхода OAuth 2.0 описывается как протокол авторизации, а не протокол аутентификации. Dogphotos.com не знает вашу личность на Facebook; ему разрешен доступ к API фотографий только от чьего-либо имени.

Потоки авторизации OAuth 2.0 и типы доступа

В примере с OAuth 2.0, показанном в этом разделе, используется общий поток или *тип доступа*, как он называется в OAuth 2.0, для получения токена от поставщика идентификационной информации. OAuth 2.0 определяет несколько типов доступа и расширений, каждое из которых предназначено для разных сценариев:

- код авторизации – это процесс, описанный на рис. 25.5, в котором приложение использует комбинацию кода авторизации и секрета для получения токена;
- PKCE (Proof Key for Code Exchange) – это расширение кода авторизации, которому по возможности всегда следует отдавать предпочтение, поскольку оно обеспечивает дополнительную защиту от

определенных атак, как описано на странице <https://www.rfc-editor.org/rfc/rfc7636>;

- учетные данные клиента – они используются, когда пользователь не участвует, например когда у вас есть API, взаимодействующий с другим API.

Доступно и гораздо больше типов доступа (см. <https://oauth.net/2/grant-types>), и каждый из них подходит для конкретной ситуации. В этих примерах представлены наиболее распространенные типы, но если ваш сценарий не соответствует им, то стоит изучить другие доступные типы, прежде чем думать над необходимостью изобретать свой собственный! А поскольку скоро выйдет OAuth 2.1 (<http://mng.bz/XNav>), то вполне возможно, что появятся обновленные рекомендации, о которых следует знать.

OAuth 2.0 отлично подходит для описанного мной сценария, в котором вам нужно делегировать доступ к ресурсу (вашим фотографиям) кому-то другому (dogphotos.com). Но приложения также часто хотят знать вашу личность, помимо доступа к API. Например, у сайта dogphotos.com может возникнуть желание связаться с вами через Facebook, если с вашими фотографиями будут проблемы.

Именно здесь на помощь приходит OpenID Connect. OpenID Connect использует те же основные потоки, что и OAuth 2.0, но добавляет ряд соглашений, возможность обнаружения и аутентификации. OpenID Connect рассматривает вашу личность (например, идентификатор или адрес электронной почты) как ресурс, который защищен так же, как и любой другой API. Вам по-прежнему необходимо дать согласие на предоставление dogphotos.com доступа к вашим идентификационным данным, но как только вы это сделаете, сайту потребуется дополнительный вызов API для получения этих данных, как показано на рис. 25.6.

OpenID Connect – важнейший компонент аутентификации во многих системах, но если вы создаете только API (например, API фотографий Facebook на рис. 25.5 и 25.6), все, что вас действительно волнует, – это токены в запросах; то, как был получен этот токен, менее важно с технической точки зрения. В следующем разделе мы подробнее рассмотрим эти токены и как они работают.

25.2 Аутентификация с токеном на предъявителя

В этом разделе вы познакомитесь с токенами на предъявителя: что это такое, как их можно использовать для обеспечения безопасности в API, а также об общем формате JWT для токенов. Вы узнаете о некоторых ограничениях токенов, подходах к обходу этих ограничений и некоторых общих понятиях, таких как аудитории и области действия.

Словосочетание «токен на предъявителя» состоит из двух частей, описывающих его использование:

- *токен (token)* – токен безопасности – это строка, обеспечивающая доступ к защищенному ресурсу;

- предъявитель, носитель (*bearer*) – токен на предъявителя – это токен, в котором любой, у кого есть токен (предъявитель), может использовать его, как и любой другой. Вам не нужно доказывать, что именно вы получили этот токен изначально или имеете доступ к какому-либо дополнительному ключу. Можно рассматривать токен на предъявителя как деньги: если они у вас есть, вы можете их потратить!

1. Клиентское приложение перенаправляет пользователя к поставщику идентификационной информации для проверки подлинности



2. После входа в систему пользователь дает согласие на доступ клиентского приложения к API



4. Клиентское приложение отправляет код авторизации со своим клиентским секретом поставщику идентификационной информации и получает токен доступа



5. Клиентское приложение обычно использует токен доступа для вызова конечной точки userInfo и получения утверждений пользователя



3. Поставщик идентификационной информации возвращает клиентскому приложению код авторизации вместе с идентификационным токеном, описывающим событие аутентификации. Идентификационный токен также можно использовать для выхода из системы поставщика идентификационной информации



6. Клиентское приложение может использовать токен доступа для вызова API и получения доступа к ресурсам, как в OAuth 2.0



Рис. 25.6 Использование OpenID Connect для аутентификации в Facebook и получения идентификационной информации. Общий поток такой же, как в OAuth 2.0, как показано на рис. 25.5, но с дополнительным токеном удостоверения, описывающим событие аутентификации, и вызовом API для получения сведений об удостоверении

Если второй пункт вас немного смущает, то это хорошо. Следует воспринимать токены на предъявителя как пароли: вы должны защищать их любой ценой! Например, нужно избегать включения этих токенов в строки запроса URL-адреса, поскольку они могут автоматически записываться в логи, что приведет к случайному раскрытию токена.

Снова в моде: файлы Cookie для API

Аутентификация с токеном на предъявителя чрезвычайно распространена для API, но, как и во всем, что касается сферы технологий, ситуация постоянно меняется. Одной из областей, в которой произошли большие изменения, является процесс обеспечения безопасности одностороничных приложений, создаваемых с использованием фреймворков React, Angular и Blazor WASM. В течение нескольких лет нам советовали использовать код авторизации с РКСЕ (<https://www.rfc-editor.org/rfc/rfc8252#section-6>), но есть серьезная проблема с этим паттерном. Она заключается в том, что токены на предъявителя для вызова API в конечном итоге хранятся в браузере.

Недавно появился альтернативный паттерн: Backend for Frontend (BFF). При таком подходе у вас есть традиционное приложение ASP.NET Core (бэкенд), на котором размещается приложение Blazor WASM или другое одностороничное приложение (фронтенд). Основная задача приложения ASP.NET Core – обработка аутентификации с OpenID Connect, безопасное хранение токенов на предъявителя и установка cookie-файла аутентификации точно так же, как в традиционном веб-приложении.

Фронтенд-приложение в браузере отправляет запросы бэкенд-приложению, которые автоматически включают файл cookie. Бэкенд заменяет файл аутентификации на соответствующий токен и перенаправляет запрос реальному API.

Большим преимуществом этого подхода является тот факт, что токены на предъявителя никогда не отправляются в браузер, а большая часть кода фронтенда значительно упрощается. Основной недостаток заключается в том, что необходимо запускать дополнительный бэкенд-сервис для поддержки внешнего приложения. Тем не менее этот подход быстро становится рекомендуемым. Подробнее об этом паттерне можно прочитать на странице <http://mng.bz/yQdB>. Кроме того, вы можете найти шаблон проекта для паттерна BFF от Дэмиена Боудена на странице <http://mng.bz/MBIW>.

Токены на предъявителя не обязательно должны иметь какую-либо особую ценность; например, это может быть совершенно случайная строка. Однако наиболее распространенным форматом и форматом, используемым OpenID Connect, является JWT. JWT-токены (определен в <https://www.rfc-editor.org/rfc/rfc7519.html>) состоят из трех частей:

- заголовок JSON, описывающий токен;
- полезные данные JSON, содержащие утверждения;
- двоичная подпись, созданная из заголовка и полезной нагрузки.

Каждая часть кодируется в формате Base64 и объединяется с помощью символа '.' в одну строку, которую можно безопасно передавать в HTTP-заголовках, например как показано на рис. 25.7. Сигнатура создается с использованием ключевого материала, который должен быть доступен поставщику, создавшему токен, и любому API, который его использует. Это гарантирует, что JWT-токен нельзя будет подделать, например добавить дополнительные утверждения.

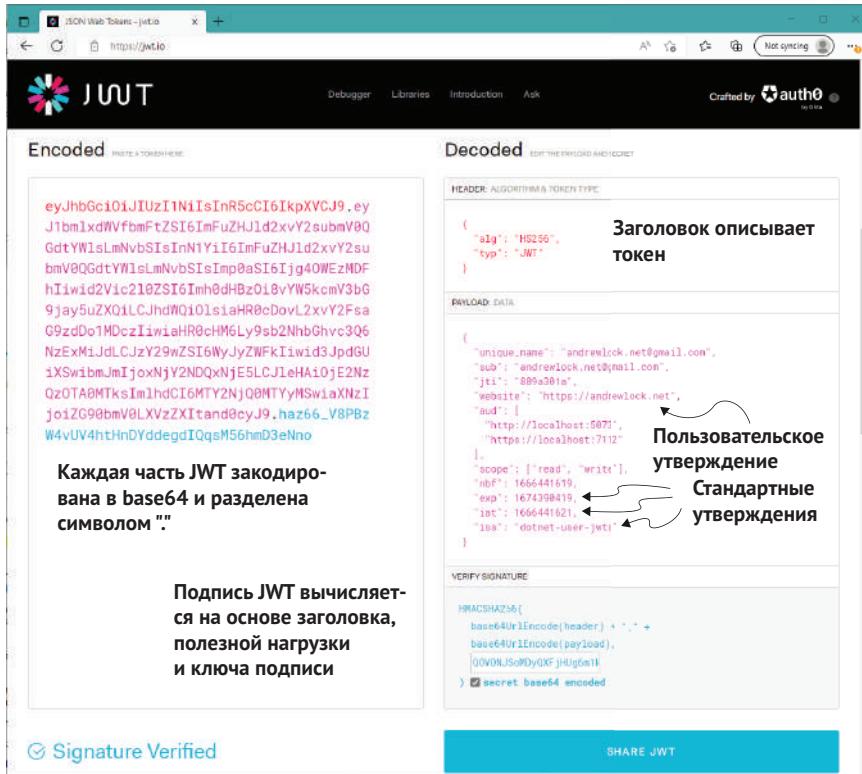


Рис. 25.7 Пример JWT-токена, расшифрованный с помощью сайта <https://jwt.io>. Он состоит из трех частей: заголовка, полезной нагрузки и сигнатуры. Всегда нужно проверять сигнатуры всех получаемых вами JWT-токенов

ВНИМАНИЕ! Всегда проверяйте сигнатуру всех используемых вами JWT-токенов, как описано на странице (<https://www.rfc-editor.org/rfc/rfc8725>). ASP.NET Core делает это по умолчанию.

На рис. 25.7 показаны утверждения, включенные в JWT-токен. У некоторых из них загадочные имена, например `iss` и `iat`. Это стандартные имена утверждений, используемые в OpenID Connect (они означают «Issuer» и «Issued at» соответственно). Обычно не нужно об этом беспокоиться, поскольку они автоматически обрабатываются ASP.NET Core при декодировании токена. Тем не менее полезно понять, что означают некоторые из этих утверждений, поскольку это поможет, когда что-то пойдет не так:

- `sub` – тема токена, уникальный идентификатор субъекта, который он описывает. Часто это будет пользователь, и в этом случае это может быть уникальный идентификатор пользователя в поставщике идентификационной информации;
- `aud` – аудитория токена с указанием доменов, для которых этот токен был создан. Когда API проверяет токен, API должен подтвердить, что утверждение `aud` JWT-токена содержит домен API;

- scope – области действия, предоставленные в токене. Области действия определяют, на что согласился пользователь/приложение (и что ему разрешено делать). На примере раздела 25.1, dogphotos.com, возможно, запросил области photos.read и photos.edit, но если пользователь согласился только с областью photos.read, области photos.edit не будет в JWT, который он получит для использования с API фотографий Facebook. Сам API должен интерпретировать, что означает каждая область видимости для бизнес-логики запроса;
- exp – время истечения срока действия токена, после которого он больше не действителен, выражается как количество секунд, прошедших с полуночи 1 января 1970 года (известное как временная метка Unix).

Важно понимать, что JWT-токены не шифруются. Это означает, что по умолчанию любой может прочитать их содержимое. Еще один стандарт, JSON Web Encryption (JWE), можно использовать для упаковки JWT-токена в зашифрованный конверт, который невозможно прочитать, если у вас нет ключа. Многие поставщики идентификационной информации включают поддержку использования JWE со вложенными JWT, а в ASP.NET Core существует поддержка обоих стандартов «из коробки», так что это стоит учитывать.

Токен на предъявителя, токен доступа, ссылочный токен... О, Боже!

Концепция токена на предъявителя, описанная в этом разделе, представляет собой общую идею, которую можно использовать несколькими способами и для разных целей. Вы уже читали о токенах доступа и токенах идентификации, используемых в OpenID Connect. Это оба токена на предъявителя; их разные имена описывают назначение токена.

В следующем списке описаны некоторые типы токенов, о которых вы можете прочитать или с которыми вы можете столкнуться:

- токен доступа – токены доступа используются для авторизации доступа к ресурсу. Это токены, которые обычно упоминаются, когда речь идет об аутентификации предъявителя. Они бывают двух видов:
 - **автономный.** Это наиболее распространенные токены, чаще всего в формате JWT. Они содержат метаданные, утверждения и подпись. Сила автономных токенов – они содержат все данные и могут быть проверены в автономном режиме – также является их слабостью, поскольку их нельзя отозвать. Из-за этого они обычно имеют ограниченный срок действия. Они также могут стать большими, если содержат много утверждений, что увеличивает размер запросов;
 - **ссылочный токен.** Они не содержат никаких данных и обычно представляют собой случайную строку. Когда защищенный API получает ссылочный токен, он должен обменяться ссылочным токеном с поставщиком идентификационной информации для получения утверждений (например, JWT). Такой подход обе-

спечивает большую конфиденциальность, поскольку утверждения никогда не раскрываются клиенту, а токен можно отозвать у поставщика идентификационной информации. Однако для этого требуется дополнительный HTTP-запрос каждый раз, когда API получает запрос. Это делает ссылочные токены хорошим вариантом для сред с высоким уровнем безопасности, где влияние на производительность менее критично;

- **токен идентификатора.** Этот токен используется в OpenID Connect (<http://mng.bz/a1M7>) для описания события аутентификации. Он может содержать дополнительные утверждения об аутентифицированном пользователе, но это не обязательно; если утверждения не указаны в токене идентификатора, их можно получить из конечной точки UserInfo поставщика идентификационной информации. Токен идентификатора всегда является JWT, но никогда не следует отправлять его другим API; это не токен доступа. Токен идентификатора также можно использовать для выхода пользователя из системы в поставщике идентификационной информации;
- **refresh-токен.** По соображениям безопасности токены доступа обычно имеют относительно короткий срок действия, иногда всего 5 минут. По истечении этого времени токен доступа становится недействительным, и вам необходимо получить новый. Заставлять пользователей входить в систему своего поставщика идентификационной информации каждые 5 минут – явно плохой опыт, поэтому в рамках процесса OAuth или OpenID Connect вы также можете запросить refresh-токен.

По истечении срока действия токена доступа вы можете отправить токен обновления поставщику идентификационной информации, и он вернет новый токен доступа без необходимости повторного входа пользователя в систему. Возможность получения действительных токенов доступа означает, что очень важно защищать токены обновления; если злоумышленник получит токен обновления, он фактически сможет выдать себя за пользователя.

В большей части вашей работы по созданию API и взаимодействию с ними вы, скорее всего, будете использовать автономные токены доступа JWT. Это то, что я в первую очередь имею в виду в этой главе, когда упоминаю токены на предъявителя или аутентификацию на предъявителя.

Теперь вы знаете, что такое токен, а также как его выпускают поставщики идентификационной информации с помощью протоколов OpenID Connect и OAuth 2.0. Прежде чем перейти к написанию кода в разделе 25.3, мы посмотрим, как выглядит типичный процесс аутентификации для приложения API ASP.NET Core, в котором используются JWT-токены на предъявителя.

Аутентификация с использованием токенов на предъявителя идентична аутентификации с использованием файлов cookie для традиционного приложения, которое уже прошло аутентификацию, как показано на рис. 25.1. Запрос к API содержит токен на предъявителя в заголовке. Любое промежуточное ПО, идущее перед компонентом аутентификации, видит запрос как не прошедший аутентификацию,

точно так же, как и в случае с аутентификацией с использованием файлов cookie, как показано на рис. 25.8.

В AuthenticationMiddleware дела обстоят немного иначе. Вместо десериализации файла cookie, содержащего ClaimsPrincipal, компонент декодирует JWT-токен в заголовке Authorization. Он проверяет сигнатуру, используя ключи подписи из поставщика идентификационной информации, и проверяет, что аудитория имеет ожидаемое значение и срок действия токена не истек.

Если токен действителен, то компонент аутентификации создает объект ClaimsPrincipal, представляющий аутентифицированный запрос, и за-дает его для HttpContext.User. Все промежуточное ПО, идущее после компонента аутентификации, увидит запрос как аутентифицированный.

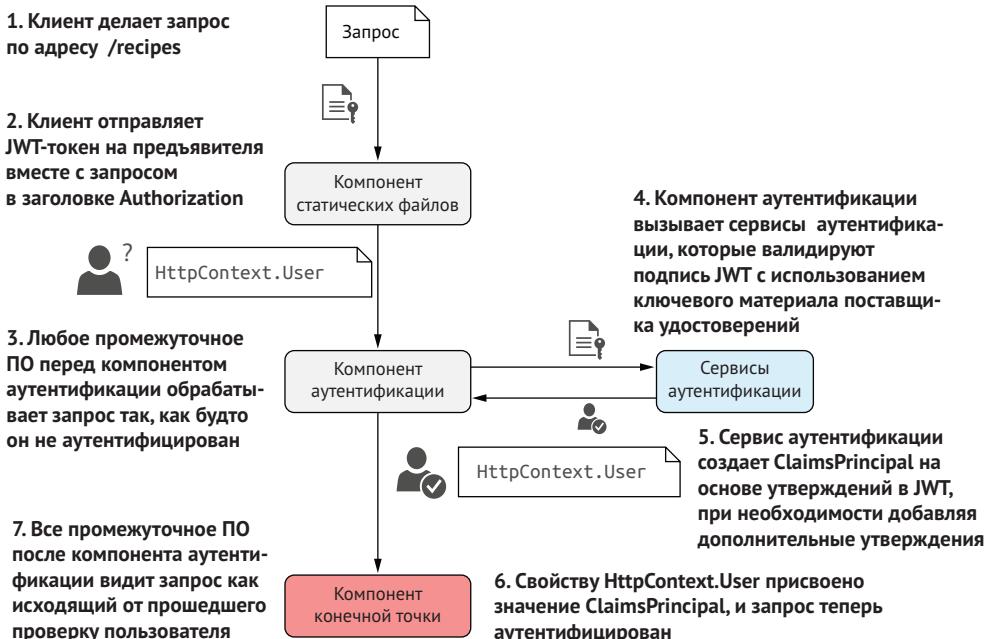


Рис. 25.8 Если запрос API содержит токен на предъявителя, токен проверяется и десериализуется компонентом аутентификации. Компонент создает объект ClaimsPrincipal из токена, при необходимости преобразуя его с помощью дополнительных утверждений, и задает свойство HttpContext.User. Последующее промежуточное ПО рассматривает запрос как аутентифицированный.

СОВЕТ Если утверждения в токене не соответствуют ожидаемым значениям ключей, можно использовать *преобразование утверждений*, чтобы сопоставить утверждения заново. Это также относится и к аутентификации с файлами cookie, но особенно часто это происходит, когда вы получаете токены от сторонних поставщиков идентификационной информации и не контролируете имена утверждений. Вы также можете использовать этот подход для добавления дополнительных утверждений для пользователя, кото-

рых не было в исходном токене. Дополнительную информацию о преобразовании утверждений см. на странице <http://mng.bz/gBJV>.

В этой главе было много теории, поэтому вам будет приятно узнать, что пришло время взглянуть на код!

25.3 Добавление аутентификации на основе JWT-токенов на предъявителя в минимальные API

В этом разделе вы узнаете, как добавить аутентификацию с использованием JWT-токенов на предъявителя в приложение ASP.NET Core. Я использую приложение с рецептами, работу над которым мы начали в главе 12, но если вы создаете приложение API с использованием контроллеров веб-API, то процесс идентичен.

Платформа .NET 7 значительно снизила количество шагов, необходимых для начала работы с аутентификацией на основе JWT-токенов, добавив некоторые соглашения, которые мы вскоре обсудим. Чтобы добавить JWT-токен к существующему приложению API, сначала установите NuGet-пакет Microsoft.AspNetCore.Authentication.JwtBearer с использованием интерфейса командной строки .NET:

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

или добавив <PackageReference> непосредственно в проект:

```
<PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer"
Version="7.0.0" />
```

Затем добавьте необходимые сервисы для настройки JWT-аутентификации для вашего приложения, как показано в листинге 25.1. Как вы, возможно, помните, компоненты аутентификации и авторизации автоматически добавляются в конвейер промежуточного ПО с помощью WebApplication, но если вы хотите контролировать расположение промежуточного ПО, то можете переопределить местоположение, как я делаю это здесь.

Листинг 25.1. Добавление аутентификации на основе JWT-токенов на предъявителя в минимальный API

```
Добавляет основные
сервисы автори-
зации
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthentication() ←
    .AddJwtBearer(); ←
builder.Services.AddAuthorization();
builder.Services.AddScoped<RecipeService>();

Добавляет основные
службы аутентификации
Добавляет и настраивает
аутентификацию на основе
JWT-тока на предъявителя
WebApplication app = builder.Build();
```

```

app.UseAuthentication();
app.UseAuthorization();           ← Добавляет компонент
                                 ← аутентификации
app.MapGet("/recipe", async (RecipeService service) =>
{
    return await service.GetRecipes();           ← Добавляет политику авторизации
}).RequireAuthorization();           ← в конечную точку минимального API

app.Run();

```

Помимо настройки аутентификации, в листинге 25.1 добавляется политика авторизации к одной конечной точке минимального API, показанной в приложении. Функция `RequireAuthorization()` добавляет к конечной точке простую политику авторизации «Аутентифицирован». Это абсолютно то же самое, что добавить атрибут `[Authorize]` к контроллерам MVC или веб-API. Любые запросы к этой конечной точке должны быть аутентифицированы; в противном случае компонент авторизации отклонит запрос с ответом 401 Unauthorized, как показано на рис. 25.9.

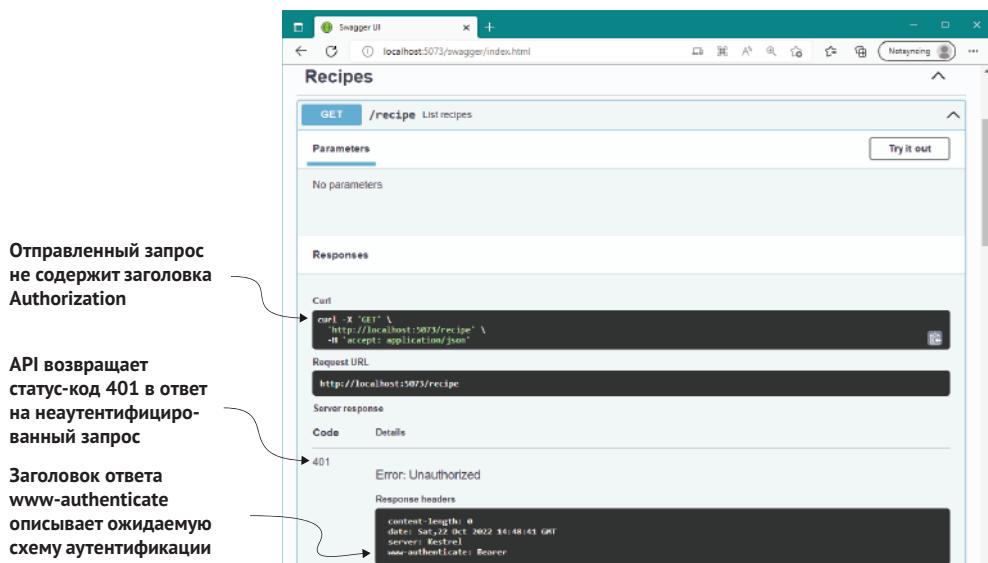


Рис. 25.9 Если вы отправите запрос к API, защищенному аутентификацией на основе JWT-токана на предъявителя, и не включите токен, то получите ответ 401 Unauthorized.

Схемы аутентификации: выбор между файлами cookie и токенами на предъявителя

При чтении об аутентификации на основе токенов на предъявителя у вас может возникнуть один вопрос: как компонент аутентификации определяет, искать файл cookie или заголовок. Ответ – схемы аутентификации.

Схема аутентификации в ASP.NET Core имеет идентификатор и связанный с ним обработчик аутентификации, который управляет способом

аутентификации пользователя, а также тем, как следует обрабатывать ошибки аутентификации и авторизации.

Например, в главе 23 схема аутентификации с файлами cookie неявно использовалась ASP.NET Core Identity. Обработчик аутентификации в этом случае аутентифицирует пользователей путем поиска файлов cookie и перенаправляет пользователей на страницу входа или страницу «Отказано в доступе» в случае сбоя аутентификации или авторизации.

В листинге 25.1 мы зарегистрировали схему аутентификации с JWT-токеном на предъявителя. Обработчик аутентификации считывает токены из заголовка `Authorization` и возвращает ответы 401 и 403 в случае сбоя аутентификации или авторизации.

Когда вы регистрируете только одну схему аутентификации, например как в листинге 25.1, ASP.NET Core автоматически устанавливает ее как схему по умолчанию, но можно зарегистрировать несколько схем. Это особенно распространено, если вы используете, например, OpenID Connect с традиционным веб-приложением. В этих случаях вы можете выбрать, какая схема будет использоваться для событий или сбоев аутентификации и как должны взаимодействовать схемы.

Использование нескольких схем аутентификации может запутать, поэтому важно внимательно следовать документации при настройке аутентификации для своего приложения. Подробнее о схемах аутентификации можно прочитать на странице <http://mng.bz/5w1a>. Если вам нужна только одна схема, проблем возникнуть не должно, в противном случае не говорите, что вас не предупреждали!

Здорово! Ответ с кодом состояния 401 на рис. 25.9 подтверждает, что приложение правильно ведет себя при обработке неаутентифицированных запросов. Очевидно, что следующий шаг – отправить запрос в API, который включает допустимый JWT-токен на предъявителя. К сожалению, здесь обычно начинаются трудности. Как создать допустимый токен? К счастью, в .NET 7 в состав интерфейса командной строки .NET входит инструмент, упрощающий создание тестовых токенов.

25.4 Использование инструмента `user-jwts` для локального тестирования JWT-токенов

В предыдущем разделе вы добавили в приложение аутентификацию на основе JWT-токенов и защитили свой API с помощью базовой политики авторизации. Проблема состоит в том, что вы не сможете протестировать API, если не умеете генерировать JWT-токены. В промышленном окружении у вас, скорее всего, будет поставщик идентификационной информации, например Auth0, Azure Active Directory или IdentityServer, который будет генерировать для вас токены с помощью OpenID Connect. Но это может усложнить локальное тестирование. В этом разделе вы узнаете, как использовать интерфейс командной строки .NET для создания JWT-токенов для локального тестирования.

В .NET 7 в комплект интерфейса командной строки .NET входит инструмент `user-jwts`, который можно использовать для создания токенов. Он действует как мини-поставщик идентификационной информации. Это означает, что вы можете создавать токены с любыми утверждениями, которые могут вам понадобиться, а ваш API сможет проверять их, используя материал ключа подписи, созданный этим инструментом.

СОВЕТ Инструмент `user-jwts` встроен в комплект разработки программного обеспечения (SDK), поэтому дополнительного не нужно ничего устанавливать. Необходимо активировать `User Secrets` для своего проекта, но `user-jwts` сделает это за вас, если вы еще этого не сделали. `User-jwts` использует секреты пользователя для хранения материала ключа подписи, применяемого для создания JWT-токенов, которые приложение использует для проверки сигнатур.

Посмотрим, как создать JWT-токен с помощью инструмента `user-jwts` и использовать его для отправки запроса в наше приложение.

25.4.1 Создание JWT-токенов с помощью инструмента `user-jwts`

Чтобы создать JWT-токен, который вы можете использовать в запросах к вашему API, выполните следующую команду с помощью инструмента `user-jwts` из папки проекта:

```
dotnet user-jwts create
```

Эта команда выполняет несколько действий:

- активирует секреты пользователя в проекте, если они еще не настроены, как если бы вы вручную выполнили команду `dotnet user-secrets init`;
- добавляет материал ключа подписи в секреты пользователя, который можно просмотреть, выполнив команду `dotnet user-secrets list`, как описано в главе 10, которая выводит конфигурацию материала ключа, как в этом примере:

```
Authentication:Schemes:Bearer:SigningKeys:0:Value =
rIhUzb3DIbtbUwiIxkgoKffDkLpY+gIJ0B4eaQzcq8=
Authentication:Schemes:Bearer:SigningKeys:0:Length = 32
Authentication:Schemes:Bearer:SigningKeys:0:Issuer = dotnet-user-jwts
Authentication:Schemes:Bearer:SigningKeys:0:Id = c99a872d
```

- настраивает сервисы аутентификации на основе JWT-токенов для поддержки токенов, созданных инструментом `user-jwts`, путем добавления конфигурации в файл `appsettings.Development.json`:

```
{
  "Authentication": {
    "Schemes": {
      "Bearer": {
        "ValidAudiences": [
          "http://localhost:5073",
          "https://localhost:7112"
        ],
      }
    }
  }
}
```

```
        "ValidIssuer": "dotnet-user-jwts"  
    }  
}  
}  
}
```

Инструмент user-jwts автоматически настраивает допустимые аудитории на основе профилей в файле launchSettings.json. Все URL-адреса приложений, перечисленные в файле launchSettings.json, указаны как допустимые аудитории, поэтому не имеет значения, какой профиль вы используете для запуска своего приложения; сгенерированный токен должен быть действительным. Служба аутентификации JWT-токенов на предъявителя автоматически считывает эту конфигурацию и настраивает себя для поддержки JWT-токенов user-jwts;

- *создает JWT*. По умолчанию токен создается с параметрами `sub` и `unique_claim`, заданными для имени пользователя вашей операционной системы, с утверждениями `aud` для каждого URL-адреса приложения в файле `launchSettings.json` и эмитентом `dotnet-user-jwts`. Вы заметите, что они соответствуют значениям, добавленным в файл конфигурации вашего API.

После вызова команды `dotnet user-jwts create` JWT-токен выводится на консоль вместе с используемым дополнительным именем и идентификатором токена. Для краткости я сократил обозначения в этой главе:

New JWT saved with ID 'f2080e51'.
Name: andrewlock

Token: evJhbGciOiJlUzI1NiIsInR5cCI6IkpxVCJ9.evJ1bmIxdlWfbmFtZSI6ImEuZHJl...

СОВЕТ Вы можете визуализировать, что именно содержится в токене, скопировав и вставив его на сайт <https://jwt.io>, как показано на рис. 25.7.

Теперь, когда у вас есть токен, пришло время его протестировать. Чтобы использовать токен, вам необходимо добавить заголовок авторизации для запросов в следующем формате (где `<token>` – полный токен, напечатанный утилитой `user-jwts`):

Authorization: Bearer <token>

Если какая-либо часть этого заголовка неверна – если вы допустили ошибку в написании слов Authorization, Beagel, не включили пробел между Beagel и вашим токеном или неправильно указали токен, – вы получите ответ 401 Unauthorized.

COBET Если вы получаете ответ 401 Unauthorized даже после добавления заголовка Authorization к вашим запросам, дважды проверьте правильность написания и убедитесь, что токен добавлен правильно.

но с префиксом "Bearer ". Опечатки имеют свойство закрадываться сюда! Вы также можете повысить уровень журналирования в своем API, чтобы понять, почему происходят сбои, как вы узнаете в главе 26.

Добавив токен, вы можете вызвать свой API, который теперь должен вернуть успешный ответ, как показано на рис. 25.10.

Запрос отправляется с заголовком Authorization.

Заголовок имеет формат Bearer <токен>

Запрос аутентифицирован правильно, поэтому API возвращает ответ 200 OK

Postman interface details:

- Request URL: `http://localhost:5073/recipe`
- Method: GET
- Headers tab selected
- Authorization header key: Authorization, value: Bearer eyJhbGciOiJIUzI1NiJ9.R0cC16kpXVCJ9.eyJhb...
- Body tab selected
- Response status: Status: 200 OK, Time: 157 ms, Size: 225 B

Рис. 25.10 Отправка запроса с токеном на предъявителя для авторизации с помощью Postman. Заголовок Authorization должен иметь формат Bearer <token>. Вы также можете настроить это на вкладке «Авторизация» в Postman.

Токена по умолчанию, созданного JWT, достаточно для аутентификации с помощью вашего API, но в зависимости от ваших требований вы можете настроить JWT для добавления или изменения утверждений. В следующем разделе вы узнаете, как это сделать.

25.4.2 Настройка JWT-токенов

По умолчанию инструмент user-jwts создает простой JWT-токен, который вы можете использовать для вызова приложения. Если вам нужны дополнительные настройки, то можете передать дополнительные параметры команде `dotnet user-jwts create` для управления генерируемым JWT. Некоторые из наиболее полезных вариантов:

- `--name` задает утверждения `sub` и `unique_name` для JWT вместо использования пользователя ОС в качестве имени;
- `--claim <key>=<value>` добавляет утверждение с именем `<key>` со значением `<value>` в JWT. Используйте эту опцию несколько раз, чтобы добавить утверждения;
- `--scope <value>` добавляет в JWT утверждение области с именем `<value>`. Используйте этот вариант несколько раз, чтобы добавить области.

Это не единственные варианты; вы можете контролировать практически все, что касается сгенерированного токена. Выполните команду `dotnet user-jwts create --help`, чтобы просмотреть все доступные параметры. Одним из вариантов, который может быть полезен в не-

которых автоматических сценариях или тестах, является параметр `--output`. Он контролирует, как JWT выводится на консоль после создания. Значение по умолчанию, `default`, печатает сводку JWT и самого токена, как вы видели ранее:

```
New JWT saved with ID 'f2080e51'.
```

```
Name: andrewlock
```

```
Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJ1bmJxdWVfbmFtZSI6ImFuZHJl...
```

Это удобно, если вы создаете токены для конкретного случая в командной строке, но альтернативные параметры вывода могут быть более полезны для сценариев. Например, при выполнении команды

```
dotnet user-jwts create --output token
```

выводится только токен

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJ1bmJxdWVfbmFtZSI6ImFuZHJl...
```

что гораздо удобнее, например, если вы пытаетесь проанализировать вывод в скрипте. В качестве альтернативы вы можете передать `--output json`, который вместо этого выведет подробную информацию о JWT, как в этом примере:

```
{
  "Id": "8bf9b2fd",
  "Scheme": "Bearer",
  "Name": "andrewlock",
  "Audience": " https://localhost:7236, http://localhost:5229",
  "NotBefore": "2022-10-22T17:50:26+00:00",
  "Expires": "2023-01-22T17:50:26+00:00",
  "Issued": "2022-10-22T17:50:26+00:00",
  "Token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJ1bmJxdWVfbmFtZSI6Im...",
  "Scopes": [],
  "Roles": [],
  "CustomClaims": {}
}
```

Обратите внимание, что это не полезная нагрузка токена; это детали конфигурации, используемые для создания JWT. Сам токен отображается в поле «Токен». Опять же, это может быть полезно, если вы генерируете JWT с помощью сценария и вам необходимо проанализировать выходные данные.

25.4.3 Управление локальными JWT

Когда вы создаете JWT-токены, инструмент `user-jwts` автоматически сохраняет конфигурацию JWT (JSON, показанный в разделе 25.4.2) на ваш жесткий диск. Он хранится рядом с файлом `secrets.json`, содержащим секреты пользователя, в месте, которое зависит от вашей операционной системы и `<UserSecretsId>` в файле проекта:

- `Windows` – `%APPDATA%\Microsoft\UserSecrets\<UserSecretsId>\user-jwts.json`;

- *Linux и macOS* – `~/.microsoft/usersecrets/<UserSecretsId>/user-jwts.json`.

Что касается пользовательских секретов, JWT-токены, созданные утилитой `user-jwts`, не зашифрованы, но они находятся за пределами каталога вашего проекта, поэтому являются лучшим подходом к локальному управлению секретами. Сгенерированные JWT-токены следует использовать только для локального тестирования; вам нужно использовать настоящего поставщика идентификационной информации для рабочих систем, чтобы безопасно создавать JWT-токены для вошедшего в систему пользователя.

Именно по этой причине инструмент `user-jwts` обновляет только файл `appsettings.Development.json` с необходимой конфигурацией, а не `appsettings.json`; это предотвращает случайное применение `user-jwts` в промышленном окружении. Вместо этого следует добавить данные поставщика идентификационной информации промышленного окружения в файл `appsettings.json`.

Помимо редактирования файла `user-jwts.json` вручную, вы можете использовать инструмент `user-jwts` для управления JWT-токенами, хранящимися локально. Помимо использования команды `create`, вы можете вызвать команду `dotnet userjwts <command>` из папки проекта, где `<command>` – это один из следующих параметров:

- `list` – список всех токенов, хранящихся в файле `user-jwts.json` для проекта;
- `clear` – удаляет все токены, созданные для проекта;
- `remove` – удаляет токен проекта, используя идентификатор токена, отображаемый командой `list`;
- `print` – выводит сведения о JWT-токене, используя идентификатор токена, в виде пар «ключ–значение»;
- `key` – может использоваться для просмотра или сброса материала ключа подписи токенов, хранящихся в диспетчере секретов пользователей. Обратите внимание, что сброс материала ключа делает все предыдущие JWT-токены, созданные инструментом, недействительными.

Инструмент `user-jwts` удобен для локального создания JWT-токенов, но вы должны не забыть добавить его в свой локальный инструмент тестирования для всех запросов. Если вы используете Postman для тестирования, вам необходимо добавить JWT-токен к вашему запросу, как показано на рис. 25.10. Однако если вы используете Swagger UI, как я описал в главе 11, все не так просто. В следующем разделе вы узнаете, как описать требования к авторизации в документе OpenAPI.

25.5 Описание требований к аутентификации для OpenAPI

В главе 11 вы узнали, как добавить в приложение ASP.NET Core документ OpenAPI, описывающий ваш API. Это используется для предо-

ставления информации таким инструментам, как автоматические генераторы клиентов, а также для пользовательского интерфейса Swagger. В данном разделе вы узнаете, как добавить требования аутентификации в документ OpenAPI, чтобы вы могли протестировать свой API с помощью Swagger UI с токенами, сгенерированными инструментом user-jwts. Одна из немного раздражающих вещей, связанных с добавлением аутентификации и авторизации в ваши API, заключается в том, что это усложняет тестирование. Вы не можете просто отправить веб-запрос из браузера; вы должны использовать такой инструмент, как Postman, к которому вы можете добавлять заголовки. Даже для любителей командной строки команды Curl могут стать громоздкими, если вам понадобится добавлять заголовки авторизации. Срок действия токенов истекает, и их, как правило, труднее генерировать. Список можно продолжить! Я видел, как эти трудности заставляли людей отключать требования аутентификации для локального тестирования или пытаться добавлять их только на поздних стадиях жизненного цикла продукта. Я настоятельно советую вам этого не делать! Попытка добавить настоящую аутентификацию на поздних стадиях проекта может вызвать головную боль и ошибки, которые вы легко могли бы обнаружить, если бы не пытались обойти сложность безопасности.

СОВЕТ Добавляйте реальную аутентификацию и авторизацию в свои API, как только вы разберетесь с требованиями, так как, скорее всего, вы обнаружите дополнительные ошибки, связанные с безопасностью.

Инструмент user-jwts может существенно помочь в решении этих проблем, поскольку вы можете легко генерировать токены в нужном вам формате, при необходимости с длительным сроком действия (поэтому вам не нужно постоянно их продлевать), без необходимости напрямую обращаться к поставщику удостоверений. Тем не менее вам нужен способ добавлять эти токены в любой инструмент, который вы используете для тестирования, например в Swagger UI. Swagger UI основан на OpenAPI спецификации вашего API, поэтому лучший (и самый простой) способ добавить поддержку аутентификации в Swagger UI – обновить требования безопасности вашего приложения в документе OpenAPI. Это состоит из двух шагов:

- определите схему безопасности, которую использует ваш API, например OAuth 2.0, OpenID Connect или простую аутентификацию на предъявителя;
- укажите, какие конечные точки вашего API используют схему безопасности.

В следующем листинге показано, как настроить документ OpenAPI с помощью Swashbuckle для API, использующего аутентификацию с JWT-токеном на предъявителя. Значения, определенные в OpenApiSecurityScheme, соответствуют настройкам по умолчанию, произведенным инструментом user-jwts при использовании метода

AddJwtBearer(). Метод AddSecurityDefinition() определяет схему безопасности для вашего API, а AddSecurityRequirement() заявляет, что весь API защищен с использованием схемы безопасности.

Листинг 25.2. Добавление аутентификации на основе токена на предъявителя в документ OpenAPI с помощью Swashbuckle

```
WebApplicationBuilder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(x =>
{
    x.SwaggerDoc("v1", new OpenApiInfo {
        Title = "Recipe App", Version = "v1" });

    var security = new OpenApiSecurityScheme
    {
        Name = HeaderNames.Authorization,
        Type = SecuritySchemeType.ApiKey,
        In = ParameterLocation.Header,
        Description = "JWT Authorization header",
        Reference = new OpenApiReference
        {
            Id = JwtBearerDefaults.AuthenticationScheme,
            Type = ReferenceType.SecurityScheme
        }
    };
    x.AddSecurityDefinition(security.Reference.Id, security);
    x.AddSecurityRequirement(new OpenApiSecurityRequirement
    {{security, Array.Empty<string>()}});
});

var app = builder.Build();

app.UseSwagger();
app.UseSwaggerUI();

app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();

app.MapGet("/", () => "Hello world!").RequireAuthorization();
app.Run();
```

Когда вы запустите приложение после добавления определения в документ OpenAPI, вы должны увидеть кнопку «Авторизовать» в правом верхнем Swagger UI, как показано на рис. 25.11. При выборе этой кнопки открывается диалоговое окно с описанием вашей схемы аутентификации, включая текстовое поле для ввода токена. В этом поле необходимо ввести Bearer <token> с пробелом между ними. Нажмите «Автори-

зовать», чтобы сохранить значение, а затем «Закрыть». Теперь, когда вы отправляете запрос к API, Swagger UI присоединяет токен к заголовку авторизации, и запрос выполняется успешно. Если вы специально используете OpenID Connect или OAuth 2.0 для защиты своих API, то можете настроить их в документе OpenApiSecurityScheme вместо использования аутентификации с токеном на предъявителя. В этом случае выбор «Авторизовать» в Swagger UI перенаправит вас к поставщику идентификационной информации для входа в систему и получения токена без необходимости что-либо копировать и вставлять.

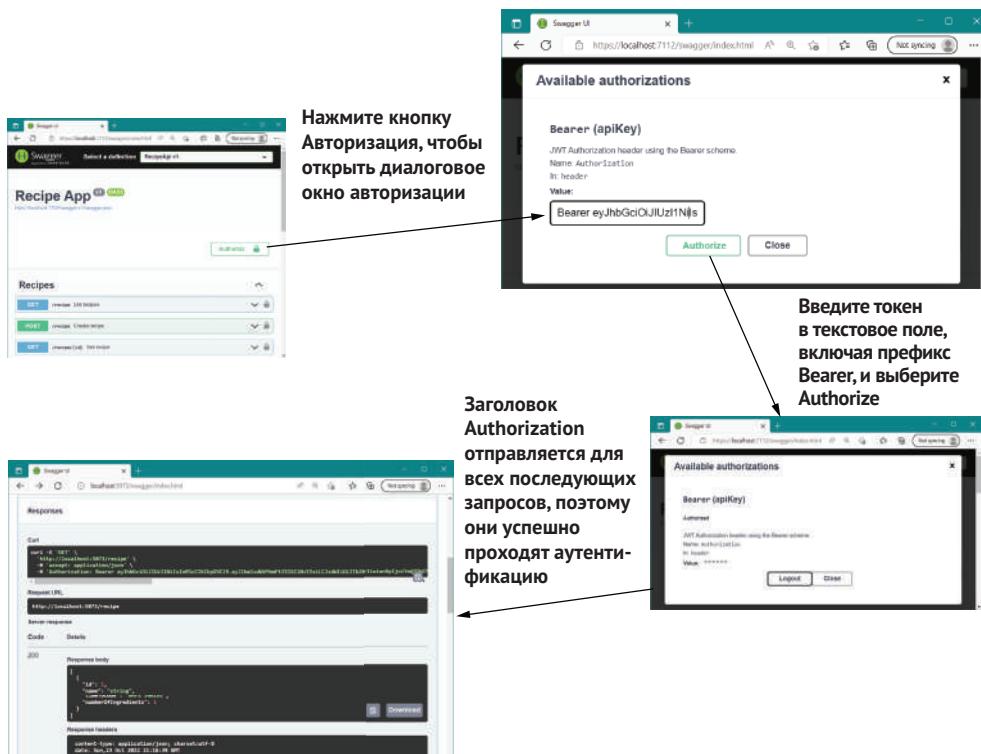


Рис. 25.11 Добавление заголовка авторизации с помощью Swagger UI. При добавлении токена обязательно введите Bearer <token>, включая префикс Bearer. Затем Swagger UI будет прикреплять токен ко всем последующим запросам, поэтому вы будете авторизованы для вызова API

Это чрезвычайно полезно, если вы используете поставщика идентификационной информации локально или применяете Swagger UI в рабочем окружении. Пример в листинге 25.2 показывает конфигурацию, когда весь ваш API защищен требованием авторизации. По моему опыту, это наиболее распространенная ситуация, но вы можете захотеть предоставить определенные конечные точки анонимным пользователям без каких-либо требований авторизации. В этом случае вы можете настроить Swashbuckle на условное применение требования только к тем конечным точкам, к которым предъявляется требование.

СОВЕТ См. документацию по Swashbuckle, чтобы узнать, как настроить эту и многие другие функции, связанные с генерацией документов OpenAPI: <http://mng.bz/6D1A>. Swashbuckle обладает широкими возможностями расширения, но, как всегда, стоит подумать, нужен ли компромисс, который вы вносите для достижения идеального документирования вашего API. Для общедоступных документов OpenAPI это вполне может быть так, но для локального тестирования или внутренних API аргументы могут быть более трудными.

В этой главе мы подробно рассмотрели использование токенов на предъявителя для аутентификации и исследовали параллели с аутентификацией с использованием файлов cookie для традиционных приложений. В последнем разделе данной главы мы рассмотрим авторизацию и способы применения различных политик авторизации к конечным точкам минимальных API.

25.6 Применение политик авторизации к конечным точкам минимальных API

До сих пор в этой главе мы фокусировались на аутентификации: процессе проверки личности инициатора запроса. Для API это обычно требует декодирования и проверки JWT-токена на предъявителя в промежуточном ПО аутентификации и установки `ClaimsPrincipal` для запроса, как вы видели в разделе 25.2. В этом разделе мы рассмотрим следующий этап защиты ваших API, авторизацию и способы применения различных требований авторизации к конечным точкам минимальных API. Хорошая новость состоит в том, что авторизация для минимальных API по сути идентична процессу авторизации, о котором вы узнали в главе 24 для Razor Pages и контроллеров MVC. Та же концепция политик авторизации, требований, обработчиков и авторизации на основе утверждений применяется одинаковым образом и использует одни и те же сервисы. На рис. 25.12 показано, как это выглядит при запросе к конечной точке минимального API, защищенной аутентификацией на основе токена на предъявителя, что очень похоже на эквивалент Razor Pages на рис. 24.2.

Вы уже видели, что можно применить общее требование авторизации, вызвав метод `RequireAuthorization()` конечной точки или группы маршрутов. Это напрямую эквивалентно добавлению атрибута `[Authorize]` к Razor Page или действию контроллера MVC. Фактически вы можете использовать тот же атрибут `[Authorize]` в конечной точке, если хотите, поэтому следующие два определения конечной точки эквивалентны:

```
app.MapGet("/", () => "Hello world!").RequireAuthorization();
app.MapGet("/", [Authorize] () => "Hello world!");
```



Рис. 25.12 Авторизация запроса к конечной точке минимального API. Компонент маршрутизации выбирает конечную точку, защищенную требованием авторизации. Промежуточное ПО аутентификации декодирует и проверяет токен на предъявителя, создавая ClaimsPrincipal, который компонент авторизации использует вместе с метаданными конечной точки, чтобы определить, авторизован ли запрос.

Если вы хотите запросить определенную политику (например, политику "CanCreate"), то можете передать имя политики методу `RequireAuthorization()` так же, как и для атрибута `[Authorize]`:

```
app.MapGet("/", () => "Hello world!").RequireAuthorization("CanCreate");
app.MapGet("/", [Authorize("CanCreate")] () => "Hello world!");
```

Аналогичным образом вы можете исключить конечные точки из требований аутентификации, используя функцию `AllowAnonymous()` или атрибут `[AllowAnonymous]`:

```
app.MapGet("/", () => "Hello world!").AllowAnonymous();
app.MapGet("/", [AllowAnonymous] () => "Hello world!");
```

Это хорошее начало, но, как вы видели в главе 24, вам часто приходится выполнять авторизацию на основе ресурсов. Например, в контексте API рецептов пользователям должно быть разрешено редактировать или удалять только те рецепты, которые они создали; они не могут редактировать чужой рецепт. Это означает, что вам необходимо знать подробности о ресурсе (рецепте) прежде определения того, авторизован ли запрос.

Авторизация на основе ресурсов, по существу, такая же как для минимальных конечных точек API, так и для Razor Pages или контроллеров MVC. Вам необходимо выполнить несколько шагов, большинство из которых мы рассмотрели в главе 24:

- 1 Создайте `AuthorizationHandler<TRequirement, TResource>` и зарегистрируйте его в DI-контейнере, как показано в главе 24.
- 2 Вставьте `IAuthService` в обработчик конечной точки.
- 3 Вызовите `IAuthService.AuthorizeAsync(user, resource, policy)`, передав `ClaimsPrincipal` для запроса, ресурс для авторизации доступа и политику, которую нужно применить.

Первый шаг идентичен процессу, показанному в главе 24, поэтому вы можете повторно использовать те же обработчики авторизации независимо от того, используете ли вы Razor Pages, минимальные API или и то, и другое! Вы можете получить доступ к `IAuthService` из конечной точки минимального API, используя стандартное внедрение зависимостей, о котором вы узнали в главах 8 и 9.

В листинге 25.3 показан пример конечной точки минимального API, которая использует авторизацию на основе ресурсов для защиты действия «удалить» для рецепта. `IAuthService` и свойство `HttpContext.User` внедряются в метод обработчика вместе с `RecipeService`. Затем конечная точка получает рецепт и вызывает `AuthorizeAsync()`, чтобы определить, продолжать удаление или вернуть ответ 403 Forbidden.

Листинг 25.3 Использование авторизации ресурсов для защиты конечной точки минимального API

```
app.MapDelete("recipe/{id}", async (
    int id, RecipeService service,
    IAuthorizationService authService,
    ClaimsPrincipal user) =>
{
    var recipe = await service.GetRecipe(id);
    var result = await authService.AuthorizeAsync(
        user, recipe, "CanManageRecipe");

    if (!result.Succeeded)
    {
        return Results.Forbid();
    }

    await service.DeleteRecipe(id);
    return Results.NoContent();
});
```

Как это часто бывает, когда вы начинаете добавлять функциональность, логика, лежащая в основе конечной точки, становится немножко запутанной по мере ее роста. Есть несколько возможных подходов, которые вы можете использовать сейчас:

- *ничего не делать.* Логика не так уж и запутана, и это только одна конечная точка. Поначалу это может быть хорошим подходом, но может стать проблематичным, если логика дублируется на нескольких конечных точках;

- *извлечь авторизацию в фильтр.* Как вы видели в главах 5 и 7, фильтры конечных точек могут быть полезны для выявления общих сквозных проблем, таких как проверка и авторизация. Вы можете обнаружить, что фильтры конечных точек помогают уменьшить дублирование в ваших обработчиках конечных точек, хотя это часто происходит за счет дополнительной сложности самого фильтра, а также уровня косвенности в ваших обработчиках. Вы можете увидеть этот подход в исходном коде, сопровождающем эту главу;
- *перенести обязанности по авторизации в предметную область.* Вместо выполнения авторизации на основе ресурсов в обработчиках конечных точек вы можете запустить проверки внутри предметной области, в данном случае в `RecipeService`. Это имеет преимущества, поскольку часто сокращает дублирование, упрощает конечные точки и гарантирует, что проверки авторизации всегда применяются независимо от того, как вы вызываете методы предметной области.

Недостатком этого подхода является тот факт, что он может привести к тому, что ваша модель предметной области или приложения будет напрямую зависеть от конструкций, специфичных для ASP.NET Core, таких как `IAuthorizationService`. Эту проблему можно обойти, создав обертку вокруг `IAuthorizationService`, но это также может добавить некоторую сложность. Даже если вы воспользуетесь этим подходом, вы обычно захотите применить политику авторизации декларативно для ваших конечных точек, а также для обеспечения того, чтобы конечная точка выполнялась только для пользователей, которые могут быть авторизованы.

Не существует единственного лучшего ответа на вопрос, какой подход выбрать; он будет варьироваться в зависимости от того, что лучше всего подходит для вашего приложения. Аутентификация и авторизация неизбежно являются сложными вопросами, поэтому важно учитывать их заранее и разрабатывать приложение с учетом безопасности.

Политики авторизации на основе области

В разделе 25.2 я описал роль областей в процессе аутентификации. Когда вы получаете токен на предъявителя от поставщика идентификационной информации (используете ли вы OpenID Connect или OAuth 2.0), вы определяете области, которые хотите получить. Затем пользователь может разрешить или запретить некоторые или все запрошенные области. Кроме того, поставщик может разрешить определенным клиентским приложениям доступ только к определенным областям. Итоговый токен доступа, который вы получаете от поставщика удостоверений и отправляете в API, может иметь некоторые запрошенные области или не иметь их вообще.

Сам API должен решить, что означает каждая область и как ее следует использовать для обеспечения соблюдения политик авторизации. Области не имеют собственной функциональности, как и утверждения, но вы можете создавать функциональность поверх них. Например, вы можете создать политику авторизации, требующую, чтобы токен имел область `"recipe.edit"`:

```
builder.Services.AddAuthorizationBuilder()
    .AddPolicy("RecipeEditScope", policy =>
        policy.RequireClaim("scope", "recipe.edit"));
```

Затем эту политику можно будет применить к любым конечным точкам, которые редактируют рецепт.

Другой распространенный шаблон – требование определенной области для авторизации выполнения любых запросов к данному приложению ASP.NET Core, например области "рецепт". Этот подход часто может заменить проверку аудитории при авторизации токена на предъявителя и может быть более гибким, поскольку вашему поставщику идентификационной информации не требуется знать домен, на котором будет размещено ваше приложение API.

В качестве альтернативы вы можете использовать области для разделения ваших API на группы, к которым могут получить доступ только определенные типы клиентов. Например, у вас может быть один набор API, доступ к которому могут получить только внутренние клиенты типа «машина–машина», другой набор, доступ к которому могут получить только пользователи с правами администратора, и еще один набор, доступ к которому могут получить только пользователи, не являющиеся администраторами.

У Даэнде есть множество практических примеров подходов к авторизации и аутентификации с использованием OpenID Connect: <http://mng.bz/o1Jp>. Примеры предназначены для пользователей IdentityServer, но демонстрируют множество передовых методов и шаблонов, которые можно использовать и с сервисами поставщиков идентификационной информации.

На этом мы подошли к концу главы, посвященной аутентификации и авторизации. Однако мы еще не закончили с безопасностью; в главе 27 мы рассмотрим потенциальные угрозы безопасности и способы их нейтрализации. Но сначала в главе 26 вы узнаете об абстракциях журналирования в ASP.NET Core и о том, как их можно использовать, чтобы следить за тем, чем именно занимается ваше приложение.

Резюме

- В больших системах с несколькими приложениями или API вы можете использовать поставщика идентификационной информации для централизации аутентификации и управления пользователями. Это часто снижает ответственность приложений за аутентификацию, уменьшая дублирование и упрощая добавление новых функций управления пользователями;
- вам следует серьезно рассмотреть возможность использования службы стороннего поставщика идентификационной информации вместо создания собственной. Управление пользователями редко является основой вашего бизнеса, и, делегируя ответственность третьей стороне, вы можете доверить защиту своих наиболее уязвимых активов экспертам;

- если вам необходимо создать собственного поставщика, вы можете использовать библиотеку IdentityServer или Open Iddict. Эти библиотеки реализуют протокол OpenID Connect, добавляя генерацию токенов к стандартному приложению ASP.NET Core. Вы должны самостоятельно создать компоненты управления пользователями и пользовательского интерфейса;
- OAuth 2.0 – это протокол авторизации, который позволяет пользователю делегировать авторизацию доступа к ресурсу другому приложению. Этот стандарт позволяет приложениям взаимодействовать без ущерба для безопасности;
- OAuth 2.0 имеет несколько типов разрешений, представляющих общие потоки авторизации. Поток кода авторизации с PKCE является наиболее распространенным типом интерактивного предоставления, когда пользователь инициирует взаимодействие. Для рабочих процессов только для компьютера, таких как API, вызывающий другой API, вы можете использовать тип представления учетных данных клиента;
- OpenID Connect построен на базе OAuth 2.0. Он добавляет соглашения, возможность обнаружения и аутентификацию в OAuth 2.0, упрощая взаимодействие со сторонними поставщиками и получение идентификационной информации о пользователе;
- JWT – это наиболее распространенный формат токена на предъявителя. Он состоит из заголовка, полезных данных и подписи и имеет кодировку Base64. При получении JWT вы всегда должны проверять подпись, чтобы убедиться, что она не была подделана;
- JWT-токены не зашифрованы, поэтому по умолчанию их может прочитать любой. JWE – это стандарт, который оборачивает JWT и шифрует его, защищая содержимое. Многие поставщики идентификационной информации поддерживают создание JWE, а ASP.NET Core поддерживает автоматическое декодирование JWE;
- проверка подлинности токена на предъявителя в ASP.NET Core аналогична проверке подлинности файлов cookie в традиционных веб-приложениях. Промежуточное ПО аутентификации десериализует токен и проверяет его. Если токен действителен, промежуточное ПО создает `ClaimsPrincipal` и устанавливает `HttpContext.User`;
- настройте аутентификацию JWT-тока на предъявителя, добавив пакет NuGet `Microsoft.AspNetCore.Authentication.JwtBearer` и вызвав `AddAuthentication().AddJwtBearer()`, чтобы добавить необходимые сервисы в ваше приложение;
- чтобы создать JWT для локального тестирования, выполните команду `dotnet user-jwt create`. Это настроит ваш API для поддержки токенов, созданных этим инструментом, и выведет на терминал токен, который вы можете использовать для локального тестирования вашего API. Добавьте токен в запросы в заголовке `Authorization`, используя формат "`Bearer <token>`";

- передайте дополнительные параметры команде `dotnet user-jwts create` для настройки сгенерированного JWT-токена. Добавьте дополнительные утверждения в сгенерированный токен, используя параметр `--claim`, измените имя вложенного утверждения, используя `--name`, или добавьте утверждения `scope` в JWT, используя `--scope`;
- чтобы включить авторизацию в Swagger UI, следует добавить схему безопасности в ваш документ OpenAPI. Создайте объект `OpenApiSecurityScheme` и зарегистрируйте его с документом OpenAPI, вызвав `AddSecurityDefinition()`. Примените его ко всем API в вашем приложении, вызвав `AddSecurityRequirement()` и передав объект схемы;
- чтобы добавить авторизацию к конечным точкам минимальных API, вызовите `RequireAuthorization()` или добавьте атрибут `[Authorize]` в обработчик конечной точки. При этом необязательно требуется имя применяемой политики авторизации, точно так же, как если бы вы применяли политики к Razor Pages и контроллерам MVC. Вы можете вызвать `RequireAuthorization()` в группах маршрутов, чтобы применить авторизацию к нескольким конечным точкам API одновременно;
- переопределите требование авторизации конечной точки, вызвав `AllowAnonymous()` или добавив атрибут `[AllowAnonymous]` в обработчик конечной точки. Это устраняет любые требования аутентификации с конечной точки, поэтому пользователи могут вызывать конечную точку без токена на предъявителя в запросе.

26

Мониторинг и устранение ошибок с помощью журналирования

В этой главе:

- структура сообщения журнала;
- запись сообщений в несколько источников;
- контроль детализации сообщений для различных окружений с помощью фильтрации;
- использование структурного журналирования для обеспечения возможности гибкого поиска.

Журналирование – одна из вещей, которые кажутся ненужными до тех пор, пока вы не начнете отчаянно нуждаться в ней! Нет ничего более неприятного, чем столкнуться с проблемой, которую можно воспроизвести только в промышленном окружении, а затем обнаружить, что у вас нет журналов, которые могли бы помочь вам отладить ее. Журналирование – это процесс записи событий или действий в приложении, который

часто включает в себя ведение записи в консоль, файл, журнал событий Windows или другие системы. В сообщение журнала можно записывать что угодно, хотя обычно существуют два разных типа сообщений:

- *информационные сообщения* – произошло стандартное событие: пользователь выполнил вход, продукт был помещен в корзину или в приложении для ведения блога было создано новое сообщение;
- *предупреждения и ошибки* – произошла ошибка или непредвиденное событие: у пользователя отрицательная сумма в корзине покупок либо произошло исключение.

Раньше существовала распространенная проблема с журналированием в крупных приложениях, которая заключалась в том, что каждая библиотека и фреймворк генерировали журналы в несколько разном формате, если вообще это делали. Когда в вашем приложении происходила ошибка и вы пытались ее диагностировать, из-за такого несоответствия было трудно связать все факты в единое целое, чтобы получить полную картину и понять проблему.

К счастью, в ASP.NET Core есть новый обобщенный интерфейс журналирования, который вы можете подключить. Он используется во всем коде ASP.NET Core, а также сторонними библиотеками, и вы можете с легкостью применять его для создания журналов в своем коде. С помощью фреймворка журналирования ASP.NET Core вы можете контролировать избыточность сообщений журналов, поступающих из всех частей кода, включая фреймворк и библиотеки, и писать сообщение журнала в любое место назначения, которое подключается к фреймворку.

В этой главе я подробно расскажу о фреймворке журналирования ASP.NET Core и объясню, как использовать его для записи событий и диагностики ошибок в собственных приложениях. В разделе 26.1 я опишу архитектуру фреймворка журналирования. Вы узнаете, как внедрение зависимостей позволяет библиотекам и приложениям создавать сообщения журнала, а также записывать эти сообщения в несколько мест назначения.

В разделе 26.2 вы узнаете, как писать собственные сообщения журнала в приложениях с помощью интерфейса `ILogger`. Мы разберем анатомию типичной записи журнала и рассмотрим ее свойства, такие как уровень сообщения журнала, категория и сообщение.

Запись сообщений в журналы полезна только в том случае, если вы умеете их читать, поэтому в разделе 26.3 вы узнаете, как добавить поставщиков журналирования в свое приложение. Поставщики журналирования контролируют, куда ваше приложение пишет сообщения. Это может быть консоль, файл или даже внешний сервис. Я покажу вам, как добавить поставщика журналирования, который пишет журналы в файл, и как сконфигурировать популярного стороннего поставщика журналирования `Serilog` в своем приложении.

Журналирование – важная часть любого приложения, но определение его *объема* может быть непростым вопросом. С одной стороны, вам нужно предоставить достаточно информации, чтобы иметь возможность диагностировать любые проблемы. С другой стороны, вы не хотите за-

полнять свои журналы данными, которые затрудняют поиск важной информации, когда вам она нужна. Хуже того. То, что достаточно для разработки, может оказаться чересчур для промышленного окружения.

В разделе 26.4 я объясню, как фильтровать сообщения журнала из различных разделов приложения, например библиотек инфраструктуры ASP.NET Core, чтобы поставщики журналирования писали только важные сообщения. Это позволяет поддерживать баланс между общирным журналированием в окружении разработки и записывать только важные сообщения в промышленном окружении.

В последнем разделе данной главы я коснусь преимуществ *структурного журналирования*, подхода, который можно использовать с некоторыми поставщиками для фреймворка журналирования ASP.NET Core.

Структурное журналирование включает в себя прикрепление данных к сообщениям журнала в виде пар «ключ–значение», чтобы упростить поиск по журналам и выполнение запросов к ним. Вы можете прикрепить, например, уникальный идентификатор клиента к каждому сообщению журнала, созданному вашим приложением. Найти все сообщения журнала, связанные с пользователем, намного проще, используя данный подход, по сравнению с записью идентификатора клиента непоследовательным образом как часть сообщения журнала.

Мы начнем эту главу с того, что разберемся, что включает в себя журналирование и почему в будущем вы скажете себе спасибо за его эффективное использование в своем приложении. Затем мы рассмотрим части фреймворка журналирования ASP.NET Core, которые вы будете использовать непосредственно в своих приложениях, и как они сочетаются друг с другом.

26.1 Эффективное использование журналирования в промышленном приложении

Представьте, что вы только что развернули новое приложение в промышленном окружении, когда клиент звонит и говорит, что получает сообщение об ошибке при работе в вашем приложении. Как бы вы определили, что вызвало проблему? Вы могли бы спросить клиента, какие шаги он предпринимает, и потенциально попробовать воспроизвести ошибку самостоятельно, но если это не сработает, то вам придется рыться в коде, пытаясь выявить ошибки. Это все, что остается сделать.

Журналирование может предоставить дополнительный контекст, необходимый для быстрой диагностики проблемы. Возможно, наиболее важные события в журнале фиксируют подробную информацию о самой ошибке, но события, которые привели к ошибке, могут быть столь же полезны при диагностике ее причины.

Есть много причин, чтобы добавить журналирование в приложение, но, как правило, они относятся к одной из трех категорий:

- журналирование с целью аудита или аналитики, чтобы отслеживать, когда произошли события;
- журналирование ошибок;

- журналирование событий, не связанных с ошибками, чтобы обеспечить запись цепочки событий к моменту, когда ошибка действительно происходит.

Первая из этих причин проста. Вам может потребоваться вести учет, например каждый раз, когда пользователь выполняет вход, или вы можете отслеживать, сколько раз вызывается конкретный метод API. Журналирование – простой способ записать поведение своего приложения, записывая сообщение в журнал каждый раз, когда происходит интересное событие.

Я считаю, что вторая причина является наиболее распространенной.

Когда приложение работает отлично, журналы часто остаются совершенно нетронутыми. А вот когда возникает проблема и звонит клиент, они становятся просто незаменимыми. Хороший набор журналов поможет понять условия, которые вызвали ошибку, включая контекст самой ошибки, а также контекст в предыдущих запросах.

СОВЕТ Даже при обширном журналировании вы можете не осознавать, что у вас в приложении есть проблема, если вы не просматриваете журналы регулярно. Для любого среднего и крупного приложения это становится непрактичным, поэтому такие службы мониторинга, как Raygun (<https://raygun.io>) или Sentry (<https://sentry.io>), могут стать бесценными инструментами для быстрого уведомления о проблемах.

Если это смахивает на то, что вам предстоит серьезно поработать, значит, вам повезло. ASP.NET Core выполнит за вас массу работы по журналированию навигационных цепочек, чтобы вы могли сосредоточиться на создании качественных сообщений журнала, имеющих наибольшую ценность при диагностике проблем.

26.1.1 Выявление проблем с помощью специальных сообщений журнала

ASP.NET Core использует журналирование во всех своих библиотеках. В зависимости от того, как вы настраиваете приложение, у вас будет доступ к деталям каждого запроса и запроса EF Core, даже без добавления дополнительных сообщений журнала в собственный код. На рис. 26.1 видны сообщения журнала, созданные при просмотре одного рецепта в приложении рецептов. Это дает вам много полезной информации. Вы можете увидеть, какой URL-адрес был запрошен, страницу Razor Page и обработчик страниц, которые были вызваны, команду базы данных EF Core, вызванный результат действия и ответ. Эта информация может оказаться бесценной, когда вы пытаетесь изолировать проблему, будь то ошибка в рабочем приложении или какая-то функция в окружении разработки, если вы работаете локально. Такое журналирование может быть полезно, но сообщения журнала, которые вы создаете сами, могут иметь еще большую ценность. Например, вы можете определить причину ошибки из сообщений жур-

нала на рис. 26.1 – мы пытаемся просмотреть рецепт с неизвестным RecipeId 5, но это далеко не очевидно. Если явно добавить сообщение журнала в свое приложение, когда это происходит, как показано на рис. 26.2, то проблема становится более очевидной.

```

[...]
INFO: Microsoft.AspNetCore.Hosting.Diagnostics[1]
Request starting HTTP/1.1 GET http://localhost:5105/Recipes/View/5 - -
INFO: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
Executing endpoint '/Recipes/View'
INFO: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[103]
Route matched with {page = "Recipes/View", area = ""}. Executing page /Recipes/View
INFO: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[105]
Executing handler method RecipesApplication.Pages.Recipes.ViewModel.OnGetAsync - ModelState is Valid
INFO: Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (1ms) [Parameters=@__id_0=? {DbType = Int32}], CommandType='Text', CommandTimeout='30'
SELECT "t"."RecipeId", "t"."Name", "t"."Method", "t"."LastModified", "i"."Name", "i"."Quantity", "i"."Unit", "i"."IngredientId"
FROM (
    SELECT "r"."RecipeId", "r"."Name", "r"."Method", "r"."LastModified"
    FROM "Recipes" AS "r"
    WHERE "r"."RecipeId" = @_id_0 AND NOT ("r"."IsDeleted")
    LIMIT 2
) AS "t"
LEFT JOIN "Ingredient" AS "i" ON "t"."RecipeId" = "i"."RecipeId"
ORDER BY "t"."RecipeId"
INFO: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[108]
Executed handler method OnGetAsync, returned result Microsoft.AspNetCore.Mvc.NotFoundResult.
INFO: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
Executing StatusCodeResult, setting HTTP status code 404
INFO: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[104]
Executed page /Recipes/View in 38.519ms
INFO: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
Executed endpoint '/Recipes/View'

```

Выполняется один запрос к URL /Recipes/View/5. Внутренние библиотеки ASP.NET Core генерируют журналы для конечной точки Razor Pages

Выполняется обработчик OnGetAsync на странице Recipes/View.cshtml

EF Core журналирует SQL-запросы, отправляемые в базу данных

Тип ActionResult и код HTTP-ответа журналируются

Рис. 26.1 Библиотеки ASP.NET Core используют журналирование повсюду. Один запрос генерирует несколько сообщений журнала, описывающих прохождение запроса через ваше приложение

Это специальное сообщение журнала легко выделяется и четко указывает на проблему (рецепта с запрошенным идентификатором не существует) и параметры или переменные, которые привели к проблеме (значение идентификатора 5). Добавление подобных сообщений журнала в приложения облегчит диагностику проблем, отслеживание важных событий и в целом даст знать, что делает приложение. Надеюсь, теперь у вас есть мотивация добавить журналирование в свои приложения, поэтому мы рассмотрим подробности того, что сюда входит. В разделе 26.1.2 вы увидите, как создать сообщение журнала и определить, куда пишутся эти сообщения. Мы подробно рассмотрим два этих аспекта в разделах 26.2 и 26.3; однако сначала выясним, где они сочетаются с точки зрения фреймворка журналирования ASP.NET Core в целом.

```

[...]
INFO: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[103]
Route matched with {page = "Recipes/View", area = ""}. Executing page /Recipes/View
INFO: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[105]
Executing handler method RecipesApplication.Pages.Recipes.ViewModel.OnGetAsync - ModelState is Valid
INFO: Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (1ms) [Parameters=@__id_0=? {DbType = Int32}], CommandType='Text', CommandTimeout='30'
SELECT "t"."RecipeId", "t"."Name", "t"."Method", "t"."LastModified"
FROM "Recipes" AS "t"
WHERE "t"."RecipeId" = @_id_0 AND NOT ("t"."IsDeleted")
LIMIT 2
) AS "t"
LEFT JOIN "Ingredient" AS "i" ON "t"."RecipeId" = "i"."RecipeId"
ORDER BY "t"."RecipeId"
INFO: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[108]
Executed handler method OnGetAsync, returned result Microsoft.AspNetCore.Mvc.NotFoundResult.
INFO: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
Executing StatusCodeResult, setting HTTP status code 404
INFO: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[104]
Executed page /Recipes/View in 958.5054ms

```

Вы можете создавать свои собственные журналы из классов вашего приложения, например этот журнал из PageModel страницы View.cshtml. Они часто более полезны для диагностики проблем и отслеживания поведения вашего приложения

Рис. 26.2 Вы можете писать собственные сообщения. Часто они более полезны для выявления проблем и интересных событий в ваших приложениях

26.1.2 Абстракции журналирования ASP.NET Core

Фреймворк журналирования ASP.NET Core состоит из ряда абстракций (интерфейсы, реализации и вспомогательные классы), наиболее важные из которых показаны на рис. 26.3:

- `ILogger` – это интерфейс, с которым вы будете взаимодействовать в своем коде. У него есть метод `Log()`, который используется для записи сообщения журнала;
- `ILoggerProvider` – используется для создания специального экземпляра `ILogger`, в зависимости от поставщика. «Консольный» `ILoggerProvider` создает `ILogger`, который пишет в консоль, тогда как «файловый» `ILoggerProvider` создает `ILogger`, который пишет в файл;
- `ILoggerFactory` – это связующее звено между экземплярами `ILoggerProvider` и `ILogger`, которые вы используете в своем коде. Вы регистрируете экземпляры `ILoggerProvider` с помощью `ILoggerFactory` и вызываете метод `CreateLogger()` для `ILoggerFactory`, когда вам нужен `ILogger`. Фабрика создает `ILogger`, который обертывает каждого из поставщиков, а поэтому, когда вы вызываете метод `Log()`, журнал записывается в каждого поставщика.

Схема на рис. 26.3 позволяет легко добавлять или изменять место, куда ваше приложение пишет сообщения журнала, не изменения при этом код приложения. В следующем листинге показан весь код, необходимый для добавления `ILoggerProvider`, который осуществляет журналирование в консоль.

Листинг 26.1. Добавление поставщика журналов консоли в файле Program.cs

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Logging.AddConsole()           ←
WebApplication app = builder.Build();
                                         | Добавляет нового поставщика,
                                         | используя свойство Logging
                                         | в WebApplicationBuilder
app.MapGet("/", () => "Hello World!");
app.Run();
```

ПРИМЕЧАНИЕ Средство записи журнала в консоли по умолчанию добавляется `WebApplicationBuilder`, как будет показано в разделе 26.3.

Помимо этой конфигурации для `WebApplicationBuilder`, вы не взаимодействуете с экземплярами `ILoggerProvider` напрямую. Вместо этого вы пишете сообщения в журналы, используя экземпляр `ILogger`. Об этом – в следующем разделе.

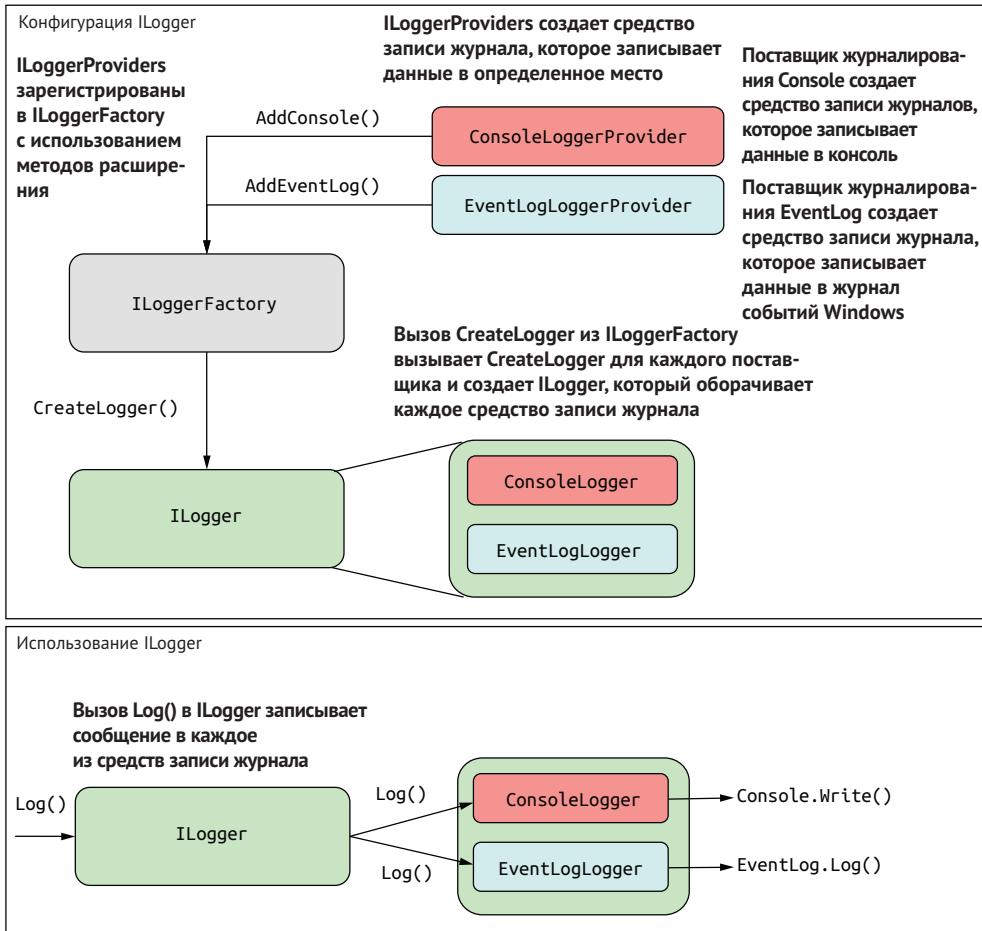


Рис. 26.3 Компоненты фреймворка журналирования ASP.NET Core. Вы регистрируете поставщиков журналирования с помощью **ILoggerFactory**, который используется для создания реализаций **ILogger**, и пишете сообщения в журналы в **ILogger**, который использует реализации **ILogger** для вывода журналов в консоль или файл. Такая схема позволяет отправлять журналы в несколько местоположений без необходимости настраивать их при создании сообщения журнала

26.2 Добавление сообщений журнала в ваше приложение

В этом разделе мы подробно рассмотрим, как создавать сообщения журнала в вашем приложении. Вы узнаете, как создать экземпляр **ILogger** и использовать его для добавления журналов в существующее приложение. Наконец, мы рассмотрим свойства, из которых состоит запись журналирования, что они означают и для чего их можно использовать.

Журналирование, как и почти все в ASP.NET Core, доступно через внедрение зависимостей. Чтобы добавить его в свои сервисы, нужно только внедрить экземпляр **ILogger<T>**, где **T** – тип вашего сервиса.

ПРИМЕЧАНИЕ При внедрении `ILogger<T>` контейнер внедрения зависимостей косвенно вызывает метод `ILoggerFactory.CreateLogger<T>()` для создания обернутого `ILogger`, показанного на рис. 26.3. В разделе 26.2.2 вы увидите, как работать напрямую с `ILoggerFactory`, если предпочитаете. Интерфейс `ILogger<T>` также реализует необобщенный интерфейс `ILogger`, но добавляет дополнительные удобные методы.

Можно использовать внедренный экземпляр `ILogger` для создания сообщений журнала, которые он записывает в каждый сконфигурированный `ILoggerProvider`. В следующем листинге показано, как внедрить экземпляр `ILogger<>` в модель страницы `Index.cshtml` для приложения с рецептами из предыдущих глав и записать сообщение журнала с указанием количества найденных рецептов.

Листинг 26.2. Внедрение `ILogger` в класс и запись сообщения журнала

```
public class IndexModel : PageModel
{
    private readonly RecipeService _service;
    private readonly ILogger<IndexModel> _log; ←

    public ICollection<RecipeSummaryViewModel> Recipes { get; set; }

    public IndexModel(
        RecipeService service, ←
        ILogger<IndexModel> log) ←
    {
        _service = service; ←
        _log = log; ←
    }

    public void OnGet()
    {
        Recipes = _service.GetRecipes(); ←
        _log.LogInformation( ←
            "Loaded {RecipeCount} recipes", Recipes.Count); ←
    }
}
```

Добавляет запись в журнал
уровня `Information`. В сообщении подставляется переменная `RecipeCount`

В этом примере используется один из множества методов расширения `ILogger` для создания сообщения журнала, `.LogInformation()`. В `ILogger` есть много методов расширения, позволяющих с легкостью указать `LogLevel` для сообщения.

ОПРЕДЕЛЕНИЕ Уровень сообщения журнала – то, насколько важно сообщение. Он определяется перечислением `LogLevel`. У каждого сообщения есть уровень.

Также видно, что у сообщения, которое вы передаете методу `.LogInformation`, есть заполнитель (placeholder), обозначенный фигурными

скобками, {RecipeCount}, и вы передаете дополнительный параметр, Recipes.Count, средству ведения журнала. Тот заменяет заполнитель параметром во время выполнения. Заполнители совпадают с параметрами по расположению, поэтому если, например, включить сюда два заполнителя, то второй заполнитель будет сопоставлен со вторым параметром.

СОВЕТ Для создания сообщения журнала можно было бы использовать обычную интерполяцию строк, например \$"Loaded {Recipes.Count} recipes". Но я рекомендую всегда использовать заполнители, поскольку они предоставляют дополнительную информацию для средства ведения журнала, которую можно использовать для структурного журналирования, как будет показано в разделе 26.5.

ILogger пишет сообщение во все настроенные поставщики журналирования при выполнении обработчика страницы OnGet в IndexModel.

Точный формат сообщения журнала будет отличаться в зависимости от поставщика, но на рис. 26.4 показано, как поставщик консоли будет отображать сообщение журнала из листинга 26.2.

Уровень журналирования

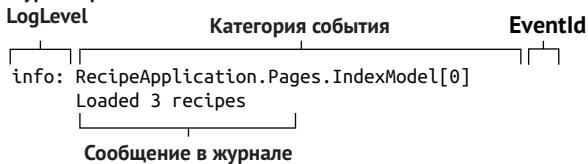


Рис. 26.4 Пример сообщения журнала, записанного в поставщик консоли по умолчанию. Уровень сообщения журнала предоставляет информацию о том, насколько важно сообщение и где оно было сгенерировано. EventId позволяет идентифицировать похожие сообщения журнала

Точный вид сообщения будет зависеть от того, куда оно пишется, но каждая запись включает в себя до шести общих элементов:

- *уровень сообщения журнала* – уровень сообщения журнала зависит от того, насколько он важен, и определяется перечислением LogLevel;
- *категория события* – категория может быть любым строковым значением, но обычно задается как полное имя класса, создающего сообщение. Для ILogger<T> полное имя типа T – это категория;
- *сообщение* – это содержание сообщения журнала. Это может быть статическая строка, или может содержать заполнители для переменных, как показано в листинге 17.2. Заполнители обозначаются фигурными скобками, {}, и заменяются предоставленными значениями параметров;
- *параметры* – если сообщение содержит заполнители, они ассоциируются с указанными параметрами. Для примера из листинга 17.2 значение Recipes.Count назначается заполнителю RecipesCount. Некоторые средства ведения журналов могут из-

влекать эти значения и предоставлять их в журналах, как будет показано в разделе 26.5;

- *исключение* — если возникает исключение, вы можете передать объект исключения функции журналирования наряду с сообщением и другими параметрами. Регистратор зарегистрирует исключение в дополнение к самому сообщению;
- *EventId* — это необязательный целочисленный идентификатор ошибки, который можно использовать, чтобы быстро найти все похожие записи в серии сообщений. Можно использовать EventId, равный 1000, когда пользователь пытается загрузить несуществующий рецепт, и EventId, равный 1001, когда пользователь пытается получить доступ к рецепту, когда у него нет соответствующих полномочий на это. Если вы не укажете EventId, будет использоваться значение 0.

Высокопроизводительное ведение журнала с помощью генераторов исходного кода

Генераторы исходного кода — это функция компилятора, представленная в C# 9. Используя эту функцию, вы можете автоматически генерировать шаблонный код при компиляции проекта. .NET 7 включает в себя несколько встроенных генераторов исходного кода, таких как генератор Regex, который я описал в главе 14. Существует также генератор исходного кода, работающий с ILogger, который может помочь избежать ошибок, таких как случайное использование интерполированных строк, и упростить использование более сложных и производительных шаблонов журналирования.

Чтобы использовать генератор исходного кода журнала в обработчике OnGet из листинга 26.2, определите частичный метод в классе IndexModel, декорируйте его атрибутом [LoggerMessage] и вызовите метод внутри метода обработчика OnGet:

```
[LoggerMessage(10, LogLevel.Information, "Loaded {RecipeCount} recipes")]
partial void LogLoadedRecipes(int recipeCount);
```

```
public void OnGet()
{
    Recipes = _service.GetRecipes();
    LogLoadedRecipes(Recipes.Count);
}
```

Атрибут [LoggerMessage] определяет идентификатор события, уровень журнала и сообщение, которое использует сообщение журнала, а параметры частичного метода, который он декорирует, подставляются в сообщение во время выполнения. Этот шаблон также поставляется с несколькими анализаторами, которые позволяют убедиться, что вы правильно используете его в своем коде, одновременно оптимизируя сгенерированный код за кулисами, чтобы предотвратить выделение памяти, где это возможно.

Генератор исходного кода журналирования не является обязательным, поэтому вам решать, использовать ли его. Вы можете прочитать больше о генераторе исходного кода, дополнительных параметрах конфигурации и о том, как он работает, в моем блоге на странице <http://mng.bz/vn14> и в документации: <http://mng.bz/4D1j>.

Не каждое сообщение журнала будет содержать все эти элементы. Например, у вас не всегда будет исключение или параметры. Существуют различные перегруженные варианты методов журналирования, которые принимают эти элементы как дополнительные параметры метода. Помимо этих необязательных элементов, у каждой записи будет как минимум уровень, категория и сообщение. Это ключевые характеристики сообщения журнала, поэтому мы рассмотрим каждую из них по очереди.

26.2.1 Уровень сообщения журнала: насколько важно сообщение журнала?

Всякий раз, когда вы создаете журнал с помощью ILogger, вы должны указать *уровень сообщения журнала*. Он указывает, насколько серьезным или важным является сообщение журнала, а это важный фактор, когда дело доходит до фильтрации сообщений журналов, которые пишет поставщик, а также для поиска важных сообщений постфактум.

Вы можете создать сообщение журнала уровня Information, когда пользователь начинает редактировать рецепт. Это полезно для отслеживания потока и поведения приложения, но это не является чем-то важным, потому что все идет нормально. Однако при возникновении исключения, когда пользователь пытается сохранить рецепт, вы можете создать сообщение журнала уровня Warning или Error.

Уровень сообщения журнала обычно задается с помощью одного из нескольких методов расширения интерфейса ILogger, как показано в листинге 26.3. В этом примере создается сообщение журнала уровня Information, когда выполняется метод View, и ошибка уровня Warning, если запрашиваемый рецепт не найден.

Листинг 26.3 Определение уровня сообщения журнала с использованием методов расширения ILogger

```
private readonly ILogger _log;
public async IActionResult OnGet(int id)
{
    _log.LogInformation(
        "Loading recipe with id {RecipeId}", id); // Записывает в журнал сообщение уровня Information

    Recipe = _service.GetRecipeDetail(id);
    if (Recipe is null)
    {
        _log.LogWarning(
            "Could not find recipe with id {RecipeId}", id); // Записывает в журнал сообщение уровня Warning
        return NotFound();
    }
    return Page();
}
```

Записывает в журнал сообщение уровня Warning

Экземпляр ILogger внедряется в контроллер с помощью внедрения в конструктор

Методы расширения `LogInformation` и `LogWarning` создают сообщения журнала с уровнем `Information` и `Warning` соответственно. Существует шесть уровней. Здесь они отсортированы в порядке убывания, от наиболее серьезного до наименее серьезного:

- `Critical` – для катастрофических сбоев, которые могут сделать невозможной правильную работу приложения, например исключения из-за нехватки памяти или если на жестком диске нет места либо горит сервер;
- `Error` – для ошибок и исключений, которые вы не можете обработать корректно. Например, исключения, возникающие при сохранении отредактированной сущности в EF Core. Операция не удалась, но приложение может продолжить работу для других запросов и пользователей;
- `Warning` – если возникает непредвиденная ситуация или ошибка, которую вы можете обработать. Вы можете использовать этот уровень для обработанных исключений или если сущность не найдена, как в листинге 26.3;
- `Information` – для отслеживания обычного потока приложений, например когда пользователь выполняет вход или просматривает определенную страницу в вашем приложении. Обычно эти сообщения журнала предоставляют контекст, когда вам нужно понять шаги, ведущие к появлению сообщения об ошибке;
- `Debug` – для отслеживания подробной информации, которая особенно полезна во время разработки. Обычно это приносит лишь краткосрочную пользу;
- `Trace` – для отслеживания очень подробной информации, которая может содержать конфиденциальные данные, такие как пароли или ключи. Редко используется и вообще не используется библиотеками фреймворков.

Можно представить эти уровни в виде пирамиды, как показано на рис. 26.5. По мере продвижения вниз по уровням важность сообщений снижается, а частота повышается. Как правило, вы будете встречать в своем приложении много сообщений журнала уровня `Debug`, но (надеюсь) всего лишь несколько сообщений уровня `Critical` или `Error`.

Важность этой пирамиды становится очевидной, если посмотреть на фильтрацию в разделе 26.4. Когда приложение находится в промышленном окружении, обычно не нужно записывать все сообщения уровня `Debug`, генерируемые вашим приложением. Было бы утомительно разбираться в таком огромном объеме сообщений, и в конечном итоге ваш диск заполнился бы сообщениями, гласящими: «Все в порядке!» Кроме того, сообщения уровня `Trace` не должны быть активированы в промышленном окружении, поскольку это может привести к утечке конфиденциальных данных. Отфильтровывая нижние уровни журнала, вы гарантируете, что создаете нормальный объем журналов в промышленном окружении, но имеете доступ ко всем уровням журнала в окружении разработки.



Рис. 26.5 Пирамида уровней сообщения журнала. Сообщения с уровнем около основания пирамиды используются чаще, но менее важны. Сообщения с уровнем ближе к верхнему должны быть редкими, но важными

В целом сообщения журнала более высокого уровня более важны, чем сообщения более низкого уровня, поэтому сообщения уровня `Warning` важнее сообщений уровня `Information`, но нужно учитывать еще один аспект. Откуда пришло сообщение или кто его создал – это ключевая информация, которая записывается с каждым сообщением журнала и называется *категорией*.

26.2.2 Категория журнала: какой компонент создал журнал?

Помимо уровня сообщения журнала, у каждого сообщения также есть категория. Вы задаете уровень самостоятельно для каждого сообщения журнала, но категория задается при создании экземпляра `ILogger`. Как и уровни, категория особенно полезна для фильтрации, как будет показано в разделе 26.4. Она записывается в каждое сообщение журнала, как показано на рис. 26.6.

Каждое сообщение журнала имеет соответствующую категорию

Обычно в качестве категории задается имя класса, который создал сообщение журнала

```
Command Prompt (light) - dpr x + v
Info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[103]
      Route matched with {page = "/Recipes/View", area = ""}. Executing page /Recipes/View
Info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[105]
      Executing handler method RecipeApplication.Pages.Recipes.ViewModel.OnGetAsync - ModelState is Valid
Info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (14ms) [Parameters=@_id_0=? (DbType = Int32)], CommandType='Text', CommandTime=30ms
      SELECT "t"."RecipeId", "t"."Name", "t"."Method", "t"."LastModified", "i"."Name", "i"."Quantity", "i"."Unit", "i"."IngredientId"
      FROM (
          SELECT "r"."RecipeId", "r"."Name", "r"."Method", "r"."LastModified"
          FROM "Recipes" AS "r"
          WHERE "r"."RecipeId" = @_id_0 AND NOT ("r"."IsDeleted")
          LIMIT 2
      ) AS "t"
      LEFT JOIN "Ingredient" AS "i" ON "t"."RecipeId" = "i"."RecipeId"
      ORDER BY "t"."RecipeId"
warn: RecipeApplication.Pages.Recipes.ViewModel[112]
      Could not find recipe with id 5
Info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[108]
```

Рис. 26.6 У каждого сообщения журнала есть ассоциированная категория. Обычно это имя класса компонента, создающего журнал. Поставщик журналирования консоли по умолчанию выводит категорию сообщения для каждого сообщения журнала

Категория представляет собой объект типа `string`, поэтому вы можете задать для нее любое значение, но по соглашению это должно быть полное имя типа, использующего `ILogger`. В разделе 26.2 я сделал это путем внедрения `ILogger<T>` в `RecipeController`; чтобы задать категорию `ILogger`, используется обобщенный параметр `T`.

В качестве альтернативы можно внедрить в свои методы `ILoggerFactory` и передать явную категорию при создании экземпляра `ILogger`. Это позволяет изменить категорию на произвольную строку.

Листинг 26.4 Внедрение `ILoggerFactory` для использования специальной категории

```
public class RecipeService
{
    private readonly ILogger _log;
    public RecipeService(ILoggerFactory factory) ←
    {
        _log = factory.CreateLogger("RecipeApp.RecipeService"); ←
    }
}
```

Внедряет экземпляр `ILoggerFactory` вместо экземпляра `ILogger`

Передает категорию в качестве объекта типа `string` при вызове метода `CreateLogger`

Также существует перегруженный вариант метода `CreateLogger()` с обобщенным параметром, который использует предоставленный класс, чтобы задать категорию. Если `RecipeService` в листинге 26.4 находился бы в пространстве имен `RecipeApp`, вызов `CreateLogger` можно было бы записать как

```
_log = factory.CreateLogger<RecipeService>();
```

Итоговый экземпляр `ILogger`, созданный этим вызовом, будет таким же, как если бы вы напрямую внедрили `ILogger<RecipeService>` вместо `ILoggerFactory`.

СОВЕТ Если вы по какой-то причине не используете свои категории в большом количестве, отдавайте предпочтение внедрению `ILogger<T>` в свои методы вместо `ILoggerFactory`.

Последняя обязательная часть каждой записи журнала довольно очевидна: *сообщение журнала*. На самом простом уровне это может быть любая строка, но стоит хорошенько подумать, какую информацию было бы полезно записать. Это может быть любая информация, которая поможет вам позже диагностировать проблемы.

26.2.3 Форматирование сообщений и сбор значений параметров

Каждый раз, когда вы создаете запись в журнале, то должны предоставить *сообщение*. Это может быть любая строка, которая вам нравится, но, как вы видели в листинге 26.2, вы также можете включить сюда заполнители, обозначаемые фигурными скобками, `{}`:

```
_log.LogInformation("Loaded {RecipeCount} recipes", Recipes.Count);
```

Включение заполнителя и значения параметра в сообщение журнала по сути создает пару «ключ–значение», которую некоторые поставщики журналирования могут хранить в качестве дополнительной информации, ассоциированной с журналом. Предыдущее сообщение журнала присваивает значение `Recipes.Count` ключу, `RecipeCount`, а само сообщение журнала создается путем замены заполнителя значением параметра, чтобы получить следующее (где `Recipes.Count=3`):

```
"Loaded 3 recipes"
```

Вы можете включить несколько заполнителей в сообщение журнала, и они будут ассоциированы с дополнительными параметрами, переданными методу журнала. Порядок заполнителей в строке форматирования должен соответствовать порядку предоставленных вами параметров.

ВНИМАНИЕ Вы должны передать в метод журнала как минимум столько же параметров, сколько есть заполнителей в сообщении. Если вы не передали достаточно параметров, то получите исключение во время выполнения.

Например, сообщение журнала

```
_log.LogInformation("User {UserId} loaded recipe {RecipeId}", 123, 456)
```

создаст параметры `UserId=123` и `RecipeId=456`. Поставщики *структурного журналирования* могут хранить эти значения в дополнение к отформатированному сообщению журнала "User 123 loaded recipe 456". Это упрощает поиск в журнальных записях определенного `UserId` или `RecipeId`.

ОПРЕДЕЛЕНИЕ *Структурное, или семантическое, журналирование* добавляет к сообщениям журнала дополнительную структуру, чтобы сделать их более доступными для поиска и фильтрации.

Вместо того чтобы хранить только текст, сохраняется дополнительная контекстная информация, обычно в виде пар «ключ–значение». JSON – распространенный формат, используемый для сообщений структурного журнала, поскольку обладает всеми этими свойствами.

Не все поставщики используют семантическое журналирование. Провайдер консоли по умолчанию, например, нет – сообщение форматируется так, чтобы заменить заполнители, но поиск в консоли по парам «ключ–значение» невозможен.

СОВЕТ Вы можете включить вывод JSON для поставщика консоли, вызвав `WebApplicationBuilder.Logging.AddJsonConsole()`. Вы можете дополнительно настроить формат провайдера, как описано в документации: <http://mng.bz/QP8v>.

Но даже если изначально вы не используете структурное журналирование, я рекомендую писать сообщения журнала, как если бы вы его использовали, с явными заполнителями и параметрами. Таким обра-

зом, если вы решите добавить поставщика структурного журналирования позже, то сразу увидите преимущества. Кроме того, я считаю, что, размышляя о параметрах, которые вы можете записывать таким образом, вы придетете к выводу записать больше значений параметров, а не только сообщение журнала.

Нет ничего более неприятного, чем увидеть сообщение типа "Cannot insert record due to duplicate key" (Не удается вставить запись из-за дублирования ключа), но значение ключа не указано!

СОВЕТ В целом я поклонник интерполированных строк C#, но не используйте их для сообщений журнала, если заполнитель и параметр тоже имеют смысл. Использование заполнителей вместо интерполированных строк даст вам то же самое выходное сообщение, а также создаст пары «ключ–значение», по которым можно вести поиск позже.

Мы уже достаточно изучили, как создавать сообщения журнала в приложении, но не рассматривали, куда пишутся эти сообщения. В следующем разделе мы рассмотрим встроенные поставщики журналирования ASP.NET Core, как они настраиваются и как заменить их на стороннего поставщика.

26.3 Контроль места записи журналов с помощью поставщиков журналирования

В этом разделе вы узнаете, как контролировать, куда записываются сообщения журнала, добавляя дополнительных поставщиков (`ILoggerProvider`) в свое приложение. В качестве примера вы увидите, как добавить простого поставщика файлового средства ведения журнала, который пишет ваши сообщения журнала в файл, в дополнение к существующему поставщику средства ведения журнала в консоли.

До этого момента мы записывали все сообщения журнала в консоль. Если вы запускали все примеры приложений ASP.NET Core локально, то, вероятно, уже видели сообщения журнала, записанные в окно консоли.

ПРИМЕЧАНИЕ Если вы используете Visual Studio и выполняете отладку с помощью IIS Express (по умолчанию), то не увидите окно консоли (хотя сообщения журнала записываются в окно Debug Output).

Запись сообщений журнала в консоль – это замечательно, когда вы выполняете отладку, но это не так уж и полезно для промышленного окружения. Никто не будет контролировать окно консоли на сервере, и журналы не будут нигде сохраняться, и в них ничего нельзя будет найти. Ясно, что вам нужно будет писать сообщения для промышленного окружения в другое место.

Как вы видели в разделе 26.1, поставщики журналирования контролируют место назначения ваших сообщений в ASP.NET Core. Они прини-

мают сообщения, которые вы создаете с помощью интерфейса `ILogger`, и записывают их в место вывода, которое зависит от поставщика.

Это название всегда меня удивляет – *поставщик журналирования*, по сути, *потребляет* создаваемые вами сообщения и выводит их в место назначения. Название более-менее отражает суть изображенного на рис. 26.3, но я все же считаю его несколько нелогичным.

Microsoft написала несколько собственных поставщиков журналирования для ASP.NET Core, доступных прямо из коробки. Сюда входят:

- *поставщик консоли* – записывает сообщения в консоль, как вы уже видели;
- *поставщик отладки* – записывает сообщения в окно отладки, например при отладке приложения в Visual Studio или Visual Studio Code;
- *поставщик журнала событий* – записывает сообщения в журнал событий Windows. Сообщения журнала записываются только при работе в Windows, поскольку для этого требуются специфичные для Windows API;
- *поставщик EventSource* – пишет сообщения с помощью Event Tracing в Windows (ETW) или LTTrng в Linux.

Существует также множество сторонних реализаций поставщиков журналирования, таких как Azure App Service, elmah.io и Elasticsearch. Помимо этого, существуют интеграции с другими уже существующими фреймворками для журналирования, такими как NLog и Serilog. Всегда стоит проверить, есть ли у вашей любимой библиотеки или службы журналирования .NET поставщик для ASP.NET Core.

СОВЕТ Serilog (<https://serilog.net>) – моя любимая платформа ведения журналов. Это зрелый фреймворк с огромным количеством поддерживаемых мест для записи логов. Подробную информацию об использовании Serilog с приложениями ASP.NET Core см. в репозитории интеграции ASP.NET Core Serilog: <https://github.com/serilog/serilog-aspnetcore>.

Вы настраиваете поставщиков журналов для своего приложения в `Program.cs`. `WebApplicationBuilder` автоматически настраивает поставщиков консоли и отладки для приложения, но вполне вероятно, что вы захотите их изменить или что-то добавить.

В этом разделе я покажу, как добавить поставщика журналирования, который ведет запись в изменяемый (rolling) файл. Таким образом, наше приложение будет ежедневно записывать сообщения журнала в новый файл. Мы продолжим вести журналы, используя поставщиков консоли и отладки, потому что они более полезны, чем поставщик файлов при локальной разработке.

Чтобы добавить стороннего поставщика журналирования в ASP.NET Core, выполните следующие действия.

- 1 Добавьте в решение пакет провайдера журналирования NuGet. Я буду использовать поставщика `NetEscapades.Extensions.Logging.Rolling-`

File, доступного на NuGet и GitHub. Вы можете добавить его в свое решение с помощью диспетчера пакетов NuGet в Visual Studio или с помощью интерфейса командной строки .NET, выполнив команду

```
dotnet add package NetEscapades.Extensions.Logging.RollingFile
```

из папки проекта своего приложения.

ПРИМЕЧАНИЕ Этот пакет представляет собой простого поставщика журналирования в файлы и доступен по адресу <https://github.com/andrewlock/NetEscapades.Extensions.Logging>. Он основан на поставщике журналирования Azure App Service. Если вам нужно больше контроля над журналами, например вы хотите указать формат файла, рассмотрите возможность использования Serilog, как описано в разделе 26.3.2.

- 2 Добавьте поставщика с помощью расширения метода `IHostBuilder.ConfigureLogging()`. Вы можете добавить поставщика файлов, вызвав метод `AddFile()`, как показано в следующем листинге. `AddFile()` – это метод расширения, предоставляемый пакетом поставщика журналирования, чтобы упростить добавление поставщика в приложение.

Листинг 26.5. Добавление стороннего поставщика журналов в `WebApplicationBuilder`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Logging.AddFile(); ←
WebApplication app = builder.Build();
app.MapGet("/", () => "Hello world!");
app.Run();
```

WebApplicationBuilder настраивает поставщиков консоли и отладки, как обычно

Добавляет нового поставщика журналирования в файлы в фабрику

ПРИМЕЧАНИЕ Добавление нового поставщика не заменяет существующих поставщиков. `WebApplicationBuilder` автоматически добавляет поставщиков консоли и ведения журналов отладки в листинге 26.5. Чтобы удалить их, вызовите `builder.Logging.ClearProviders()` перед добавлением поставщика файлов.

После настройки поставщика файлов вы можете запустить приложение и сгенерировать журналы. Каждый раз, когда ваше приложение записывает сообщение в журнал с использованием экземпляра `ILogger`, `ILogger` пишет сообщение всем настроенным поставщикам, как показано на рис. 26.7. Сообщения в консоли остались доступны, но у вас также есть постоянная запись журналов, хранящаяся в файле.

СОВЕТ По умолчанию поставщик файлов пишет сообщения журнала в подкаталог вашего приложения. Вы можете указать дополнительные параметры, такие как имена файлов и ограничения на размер файла с использованием перегруженных вариантов мето-

да `AddFile()`. Для промышленного окружения я рекомендую использовать более авторитетного поставщика, такого как `Serilog`.

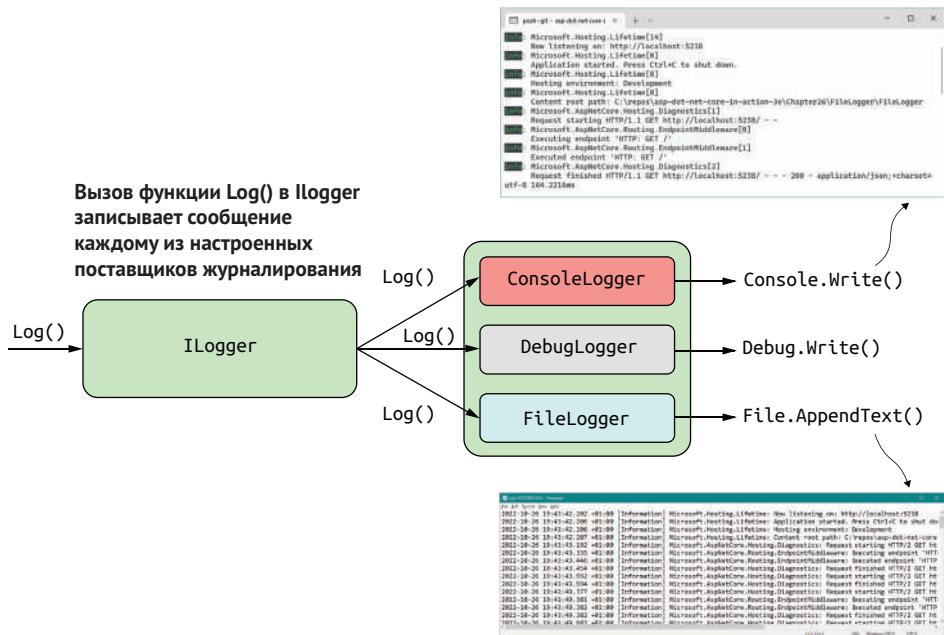


Рис. 26.7 Запись сообщения в журнал с помощью `ILogger` использует всех настроенных поставщиков. Это позволяет, например, записывать удобное сообщение в консоль, сохраняя при этом журналы в файл

Ключевой вывод, который можно сделать из листинга 26.5, заключается в том, что система поставщиков упрощает интеграцию существующих фреймворков и поставщиков журналирования с абстракциями журналирования ASP.NET Core. Какого бы поставщика вы ни выбрали для использования в своем приложении, принципы остаются прежними: вызовите метод `ConfigureLogging` для `IHostBuilder` и добавьте нового поставщика журналирования, используя в этом случае методы расширения, такие как `AddConsole()` или `AddFile()`.

Запись сообщений вашего приложения в файл может быть полезна в некоторых сценариях, и, конечно, это лучше, чем использовать для данной цели несуществующее окно консоли в промышленном окружении, но это по-прежнему, возможно, не лучший вариант.

Если вы, например, обнаружили ошибку в промышленном окружении и вам нужно быстро просмотреть журналы, чтобы выяснить, что произошло, нужно войти на удаленный сервер, найти файлы журналов на диске и пролистать их, чтобы найти проблему. Если у вас несколько веб-серверов, вам предстоит гигантская работа по получению всех журналов, прежде чем вы сможете приступить к устранению ошибки. Звучит невесело. Добавьте к этому возможность проблем с правами доступа к файлам или дискового простран-

ства, и журналирование в файлы будет казаться не таким уж и привлекательным.

Вместо этого часто лучше отправлять журналы в централизованное место, отдельно от приложения. Точное расположение этого места зависит от вас; суть в том, что каждый экземпляр вашего приложения отправляет свои журналы в одно и то же место, отдельно от самого приложения.

Если вы запускаете приложение в Azure, то получаете централизованное журналирование бесплатно, потому что можете собирать журналы с помощью поставщика Azure App Service. В качестве альтернативы можно отправить свои журналы стороннему агрегатору журналов, например Loggr (<http://loggr.net>), elmah.io (<https://elmah.io/>) или Seq (<https://getseq.net/>). Вы можете найти поставщиков журналирования для каждого из этих сервисов в NuGet, поэтому их добавление ничем не отличается от добавления поставщика файлов, которое вы уже видели.

Какого бы поставщика вы ни добавили, как только вы начнете запускать приложения в рабочем окружении, вы быстро обнаружите новую проблему: огромное количество сообщений журнала, генерируемых приложением! В следующем разделе вы узнаете, как держать это под контролем, не влияя на локальную разработку.

26.4 Изменение избыточности сообщений журналов с помощью фильтрации

В этом разделе вы увидите, как уменьшить количество сообщений журнала, записываемых в поставщики журналирования. Вы узнаете, как применить фильтр базового уровня, отфильтровывать сообщения из определенных пространств имён и использовать фильтры для конкретного поставщика журналирования.

Если вы уже экспериментировали с журналированием, то, вероятно, заметили, что получаете большое количество сообщений, даже для одного запроса, такого как на рис. 26.2: это сообщения от сервера Kestrel и от EF Core, не говоря уже о собственных. Когда вы выполняете отладку локально, доступ ко всей этой информации чрезвычайно полезен, но в промышленном окружении вы будете так поглощены шумом, что вам будет непросто выбрать важные сообщения.

ASP.NET Core включает возможность фильтрации сообщений журнала *до* того, как они будут записаны, основываясь на сочетании трех вещей:

- уровень сообщения журнала;
- категория средства ведения журнала (создавшего сообщение журнала);
- поставщик журналирования.

Вы можете создать несколько правил, используя эти свойства, и для каждого созданного журнала будет применяться наиболее конкретное правило, чтобы определить, следует ли записывать сообщение журнала в вывод. Можно создать три следующих правила:

- *минимальный уровень сообщения журнала по умолчанию – Information:* если другие правила не применяются, только сообщения с уровнем Information или выше будут записываться в поставщики;
- *для категорий, начинающихся со слова Microsoft, минимальный уровень – это Warning:* любое средство ведения журнала, создаваемое в пространстве имен, которое начинается со слова Microsoft, будет записывать только те сообщения, которые имеют уровень Warning или выше. Так вы отфильтруете шумные сообщения фреймворка, которые видели на рис. 26.6;
- *для поставщика консоли минимальный уровень – это Error:* сообщения журнала, записываемые в консоль поставщика, должны иметь минимальный уровень Error. Сообщения с более низким уровнем не будут писаться в консоль, хотя они могут быть записаны с использованием других поставщиков.

Обычно цель фильтрации сообщений журналов – уменьшить количество сообщений, записываемых в определенные поставщики или из определенных пространств имен (в зависимости от категории журнала).

На рис. 26.8 показан возможный набор правил фильтрации, которые применяются к поставщикам консоли и файлового журналирования.

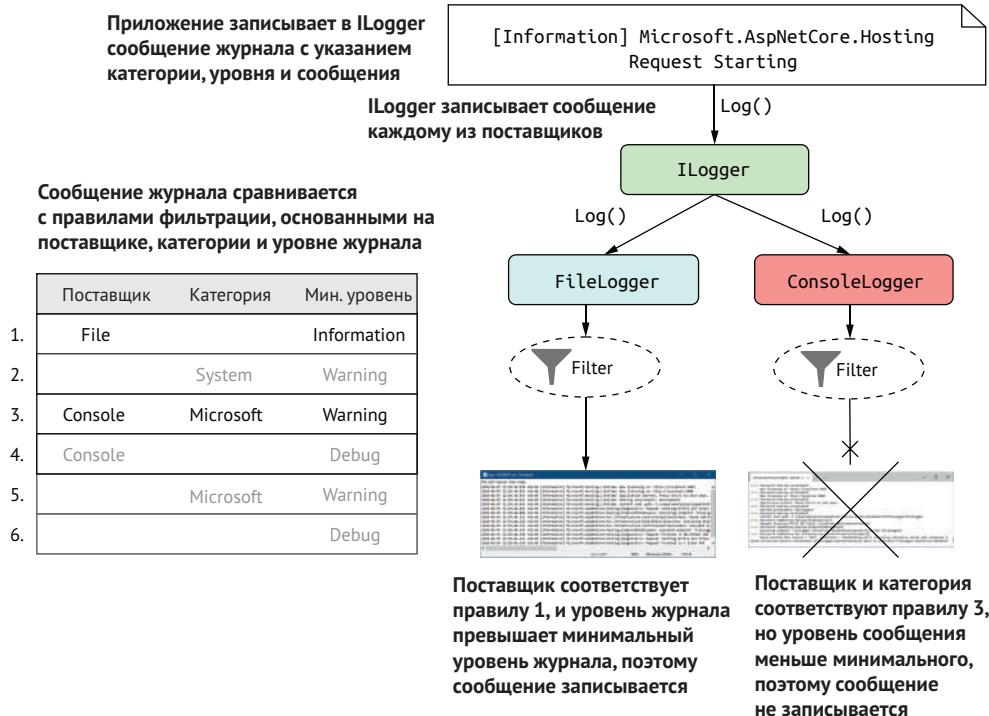


Рис. 26.8 Применение правил фильтрации к сообщению журнала для определения необходимости записи его в журнал. Для каждого поставщика выбирается самое конкретное правило. Если журнал превышает требуемый минимальный уровень правил, поставщик ведет запись в журнал; в противном случае он игнорирует сообщение

В этом примере средство ведения журнала явно ограничивает сообщения, пишущиеся в пространстве имен Microsoft, уровнем `Warning` или выше, поэтому журнал игнорирует показанное сообщение.

И наоборот, у средства ведения журнала для файлов нет правила, которое явно ограничивает пространство имен Microsoft, поэтому он использует настроенный минимальный уровень `Information` и записывает сообщение журнала.

СОВЕТ При решении, следует ли записывать сообщение в журнал, выбирается только одно правило; они не сочетаются. На рис. 26.8 правило 1 считается более строгим, чем правило 5, поэтому сообщение записывается в поставщика файлов, хотя оба правила технически могут быть применимы.

Обычно набор правил журналирования для приложения определяется с использованием многоуровневой конфигурации, описанной в главе 10, потому что он позволяет легко использовать разные правила при работе в окружении разработки или промышленном окружении.

СОВЕТ Как вы видели в главе 11, вы можете загружать параметры конфигурации из нескольких источников, таких как файлы JSON и переменные окружения, и загружать их условно на основе `IHostingEnvironment`. Обычной практикой является включение настроек ведения журнала для промышленного окружения в файл `appsettings.json` и переопределений для вашей локальной среды разработки в `appsettings.Development.json`.

`WebApplicationBuilder` автоматически загружает правила конфигурации из раздела `Logging` объекта `IConfiguration`. Это происходит автоматически, и вам редко придется прибегать к настройке, но в листинге 26.6 показано, что вы также можете добавить правила конфигурации из раздела `«LoggingRules»` с помощью `AddConfiguration()`.

ПРИМЕЧАНИЕ `WebApplicationBuilder` всегда добавляет конфигурацию для загрузки из раздела `Logging`; вы не можете это удалить. По этой причине редко стоит добавлять конфигурацию самостоятельно; вместо этого, где это возможно, используйте раздел конфигурации по умолчанию.

Листинг 26.6. Загрузка конфигурации журналирования с помощью метода `AddConfiguration()`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Logging.AddConfiguration(
    builder.Configuration.GetSection("LoggingRules")); <-- Загружает конфигурацию фильтрации журналов из раздела LoggingRules

var app = builder.Build();

app.MapGet("/", () => "Hello world!");
app.Run();
```

Если вы не переопределите раздел конфигурации, ваш файл appsettings.json обычно будет содержать раздел Logging, который определяет правила конфигурации для вашего приложения. В листинге 26.7 показано, как это может выглядеть при определении всех правил, показанных на рис. 26.8.

Листинг 26.8 Раздел конфигурации фильтрации журналов файла appsettings.json

```
{  
    "Logging": {  
        "LogLevel": {  
            "Default": "Debug",  
            "System": "Warning",  
            "Microsoft": "Warning"  
        },  
        "File": {  
            "LogLevel": {  
                "Default": "Information"  
            }  
        },  
        "Console": {  
            "LogLevel": {  
                "Default": "Debug",  
                "Microsoft": "Warning"  
            }  
        }  
    }  
}
```

Правила, которые будут применяться, если для поставщика нет применимых правил

Правила, которые будут применяться к поставщику файлов

Правила, которые будут применяться к поставщику консоли

При создании правил журналирования важно помнить, что если у вас есть *какие-либо* правила для конкретного поставщика, они будут иметь приоритет над правилами на основе категорий, определенными в секции "LogLevel". Следовательно, для конфигурации, определенной в листинге 26.7, если ваше приложение использует только поставщиков файлов или консоли, правила в секции "LogLevel" никогда не применяются.

Если вас это сбивает с толку, не волнуйтесь – меня тоже. Каждый раз, когда я настраиваю журналирование, я обращаюсь к алгоритму, используемому для того, чтобы определить, какое правило будет применяться для определенного поставщика и категории. Вот он:

- 1 Выбрать все правила для данного поставщика. Если правила не применяются, выбрать все правила, которые не определяют поставщика (верхняя секция "LogLevel" из листинга 26.7).
- 2 Из выбранных правил выбрать правила с самым длинным совпадающим префиксом категории. Если ни одно из выбранных правил не соответствует префиксу категории, выбрать "Default", если он есть.
- 3 Если выбрано несколько правил, использовать последнее.
- 4 Если правила не выбраны, использовать глобальный минимальный уровень "LogLevel:Default" (это Debug в листинге 26.7).

Каждый из этих шагов (кроме последнего) сужает применимые правила для сообщения журнала, пока не останется одно. Вы видели, как это действует для категории "Microsoft" на рис. 26.8. На рис. 26.9 этот процесс показан более подробно.

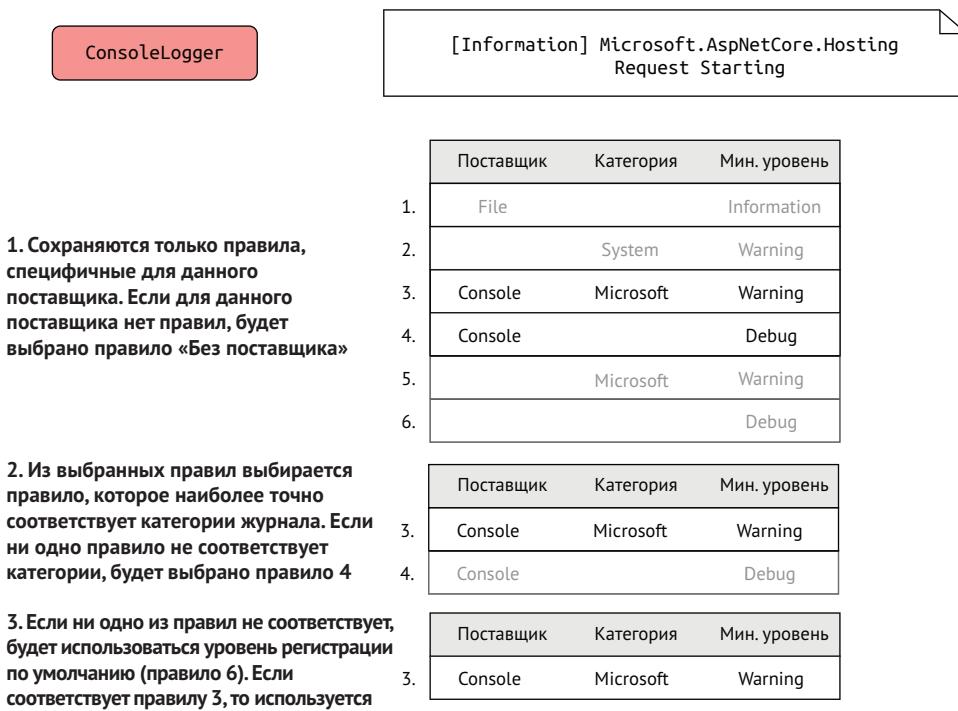


Рис. 26.9 Выбор правила для применения из доступного набора для поставщика консоли и уровня Information. Каждый шаг уменьшает количество применяемых правил, пока не останется только одно

ВНИМАНИЕ! Правила фильтрации сообщений журналов не объединяются; выбирается одно правило. Включение правил для конкретного поставщика переопределит глобальные правила для конкретных категорий, поэтому я стараюсь придерживаться правил для конкретных категорий в своих приложениях, чтобы общий набор правил было легче понять.

После эффективной фильтрации ваши сообщения журналов для промышленного окружения должны быть намного более управляемыми, как показано на рис. 26.10. Обычно я считаю, что лучше ограничить сообщения журналов из инфраструктуры ASP.NET Core и библиотеки, на которые она ссылается, уровнем Warning или выше, сохраняя журналы, которые мое приложение записывает в Debug в окружении разработки и Information в промышленном окружении.

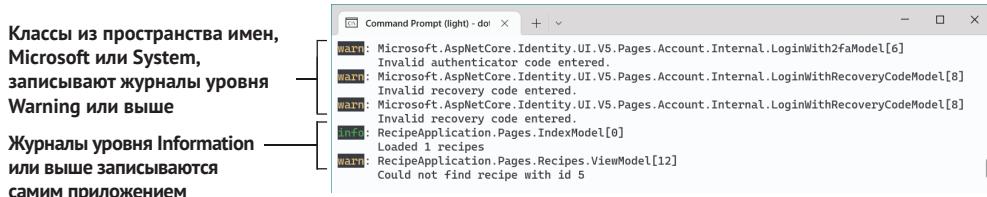


Рис. 26.10 Использование фильтрации для уменьшения количества записываемых сообщений журналов. В этом примере фильтры категорий были добавлены в пространства имен Microsoft и System, поэтому записываются только сообщения уровня Warning и выше. Это увеличивает долю журналов, имеющих прямое отношение к вашему приложению

Это близко к конфигурации по умолчанию, используемой в шаблонах ASP.NET Core. Может оказаться, что вам нужно добавить дополнительные фильтры для конкретных категорий, в зависимости от используемых библиотек NuGet и категорий, в которые они пишут. Обычно лучший способ выяснить – это запустить приложение и посмотреть, не завалило ли вас бесполезными сообщениями журнала.

СОВЕТ Большинство поставщиков журналов отслеживают изменения конфигурации и динамически обновляют свои фильтры. Это означает, что вы должны иметь возможность изменить свой appsettings.json или appsettings.Development.json и проверить влияние на сообщения журнала без перезапуска приложения.

Даже когда вы контролируете количество журналов, если вы будете придерживаться поставщиков по умолчанию, таких как файловые журналы или консоли, то, вероятно, пожалеете об этом в долгосрочной перспективе. Эти поставщики отлично работают, но когда дело дойдет до поиска конкретных сообщений об ошибках или анализа журналов, у вас появится много работы. В следующем разделе вы увидите, как структурное журналирование может помочь решить эту проблему.

26.5 Структурное журналирование: создание полезных сообщений журналов с возможностью поиска

В этом разделе вы узнаете, как структурное журналирование помогает работать с сообщениями журнала. Вы научитесь прикреплять пары «ключ–значение» к сообщениям журнала, а также научитесь хранить и защищать значения ключей с помощью поставщика структурного журналирования Seq. Наконец, вы узнаете, как использовать области для прикрепления пар «ключ–значение» ко всем сообщениям журнала в блоке.

Представим, что вы развернули приложение с рецептами, над которым мы работаем, в промышленном окружении. Вы добавили в приложение журналирование, чтобы иметь возможность отслеживать любые ошибки в своем приложении, и сохраняете журналы в файле.

Однажды клиент звонит вам и говорит, что не может увидеть свой рецепт. Конечно, когда вы просматриваете сообщения журнала, то видите предупреждение:

```
warn: RecipeApplication.Pages.Recipes.ViewModel [12]
      Could not find recipe with id 3245
```

Это вызывает у вас интерес: почему это произошло? С этим *клиентом* такое случалось раньше? Случалось ли такое раньше с этим *рецептом*? Было ли такое с *другими* рецептами? Случается ли такое регулярно?

Как бы вы ответили на эти вопросы? Учитывая, что журналы хранятся в текстовом файле, можно было бы приступить к обычному поиску в выбранном вами редакторе, разыскивая фразу "Could not find recipe with id" (Не удалось найти рецепт с идентификатором). В зависимости от ваших навыков работы с блокнотом вы, вероятно, могли бы получить ответы на свои вопросы, но, скорее всего, это будет трудоемкий, подверженный ошибкам и болезненный процесс.

Ограничивающим фактором является то, что журналы хранятся в виде *неструктурированного* текста, поэтому обработка текста – единственный доступный для вас вариант. Лучше хранить сообщения журналов в структурированном формате, чтобы их можно было легко запрашивать, фильтровать и создавать аналитику. Сообщения журналов, содержащих структурированные данные, можно хранить в любом формате, но в наши дни для них обычно используется формат JSON. Например, структурированная версия того же сообщения журнала с предупреждением могла бы выглядеть примерно так:

```
{
  "eventLevel": "Warning",
  "category": "RecipeApplication.Pages.Recipes.ViewModel",
  "eventId": "12",
  "messageTemplate": "Could not find recipe with {recipeId}",
  "message": "Could not find recipe with id 3245",
  "recipeId": "3245"
}
```

Это сообщение содержит все те же сведения, что и неструктурированная версия, но в формате, который позволит вам легко искать определенные записи журнала. Это упрощает фильтрацию журналов по EventLevel, или показ только тех журналов, которые относятся к идентификатору определенного рецепта.

ПРИМЕЧАНИЕ Это лишь пример того, как может выглядеть сообщение журнала, содержащее структурированные данные. Формат, используемый для журналов, будет варьироваться в зависимости от используемого поставщика, и это может быть что угодно.

Ключевым моментом является то, что свойства журнала доступны в виде пары «ключ–значение». Для добавления структурного журналирования в приложение требуется поставщик, который может создавать и хранить сообщения журналов, содержащих структурированные

данные. Elasticsearch – популярная поисковая и аналитическая система, которую можно использовать для хранения и запроса журналов. В Serilog есть получатель данных для записи журналов в Elasticsearch, который вы можете добавить в свое приложение так же, как вы добавляли получатель данных консоли в разделе 26.3.

ПРИМЕЧАНИЕ Elasticsearch – это поисковая система на основе REST, которая часто используется для агрегирования журналов. Подробности см. на странице <https://www.elastic.co/elasticsearch>.

Elasticsearch – это мощное средство промышленного масштаба для хранения журналов, но его настройка – занятие не для слабонервных. Даже после того, как вы его запустите, вы столкнетесь с крутой криевой обучения, связанной с синтаксисом запроса. Если вас интересует что-то более удобное, то Seq (<https://getseq.net>) – отличный вариант. В следующем разделе я покажу вам, как добавить Seq в качестве поставщика структурного журналирования, что значительно упрощает анализ журналов.

26.5.1 Добавление поставщика структурного журналирования в приложение

Чтобы продемонстрировать преимущества структурного журналирования, в этом разделе вы настроите приложение для записи сообщений журналов в Seq. Вы увидите, что конфигурация практически идентична поставщикам неструктурного журналирования, но возможности, предоставляемые структурным журналированием, облегчают задачу.

Seq устанавливается на сервере или на локальном компьютере и собирает сообщения журналов, содержащих структурированные данные по протоколу HTTP, предоставляя веб-интерфейс для просмотра и анализа журналов. В настоящее время он доступен в виде приложения для Windows или контейнера Docker для Linux. Вы можете установить бесплатную версию для разработки, которая позволит вам поэкспериментировать со структурным журналированием в целом.

СОВЕТ Вы можете скачать установщик Windows для Seq на странице <https://getseq.net/Download>.

С точки зрения приложения, процесс добавления поставщика Seq должен быть вам знаком.

- 1 Установите поставщика Seq с помощью Visual Studio или интерфейса командной строки .NET:

```
dotnet add package Seq.Extensions.Logging
```

- 2 Добавьте поставщика ведения журнала Seq в Program.cs, вызвав AddSeq():

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Logging.AddSeq();
```

Это все, что нужно, чтобы добавить Seq в приложение. Он будет отправлять журналы на локальный URL-адрес по умолчанию, когда Seq будет установлен в вашем локальном окружении. Метод расширения `AddSeq()` включает дополнительные перегруженные варианты для настройки Seq при переходе в промышленное окружение, но это все, что вам нужно, чтобы начать проводить эксперименты локально.

Если вы еще этого не сделали, установите Seq на своей машине, используемой для разработки, и перейдите к приложению Seq по адресу `http://localhost: 5341`. На другой вкладке откройте свое приложение и запустите просмотр и создание сообщений журналов. Вернувшись в Seq, если вы обновите страницу, увидите список сообщений журналов, примерно такой, что показан на рис. 26.11. Щелкнув мышью по сообщению, вы раскроете его и увидите структурированные данные.

Журналы отображаются в обратном хронологическом порядке

Обновите список журналов или включите автоматическое обновление

При нажатии на журнал отображаются структурированные данные

Журналы уровня Warning выделены желтым цветом

Рис. 26.11 Пользовательский интерфейс Seq. Сообщения журнала представлены в виде списка. Вы можете посмотреть детали структурного журналирования отдельных сообщений, аналитику сообщений и делать поиск по свойствам сообщений

ASP.NET Core поддерживает структурное журналирование, обрабатывая каждый перехваченный параметр из строки форматирования сообщения в виде пары «ключ–значение». Если вы создаете сообщение журнала, используя следующую строку форматирования:

```
_log.LogInformation("Loaded {RecipeCount} recipes", Recipes.Count);
```

поставщик создаст параметр `RecipeCount` со значением `Recipes.Count`. Эти параметры добавляются как свойства в каждое сообщение журнала, содержащее структурированные данные, как видно на рис. 26.11.

Такие сообщения, как правило, легче читать, чем стандартный вывод консоли, но их истинная мощь проявляется, когда нужно ответить на конкретный вопрос. Рассмотрим предыдущую проблему, где вы видите эту ошибку:

```
Could not find recipe with id 3245
```

Вы хотите понять, насколько широко распространена данная проблема. В качестве первого шага можно было бы определить, сколько раз возникала эта ошибка, и узнать, возникала ли она с другими рецептами. Seq позволяет фильтровать журналы, а также создавать SQL-запросы для анализа данных, поэтому поиск ответа на вопрос занимает считанные секунды, как показано на рис. 26.12.

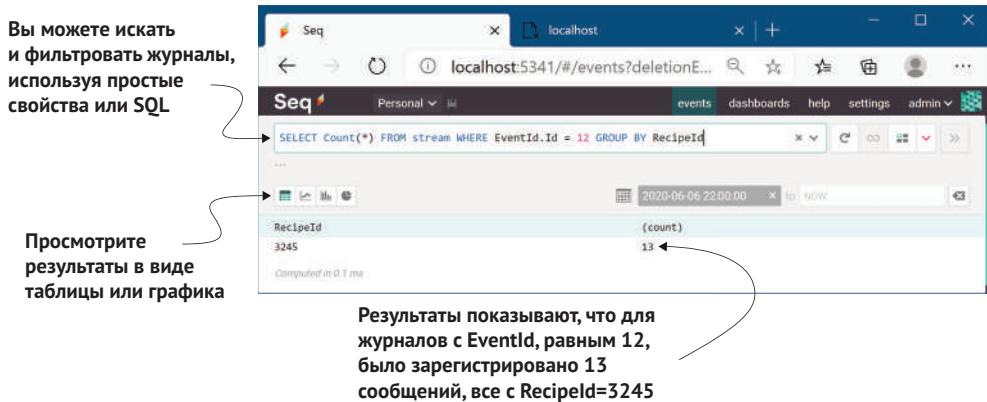


Рис. 26.12 Запрос журналов в Seq. Структурное журналирование упрощает анализ журнала, как в этом примере

ПРИМЕЧАНИЕ Вам не нужны такие языки запросов, как SQL, для простых запросов, но так проще разбираться в данных. Другие поставщики могут предоставлять языки запросов, отличные от SQL, но принцип такой же, как и в этом примере с Seq.

Быстрый поиск показывает, что вы записали сообщение журнала с `EventId.Id=12` (EventId интересующего нас предупреждения) 13 раз, и каждый раз идентификатор – нарушитель рецепта был равен 3245. Это говорит о том, что с этим рецептом что-то не так, а это указывает вам правильное направление для поиска проблемы.

Чаще всего для выявления ошибок в промышленном окружении требуется подобная детективная работа, чтобы определить, где возникла проблема. Структурное журналирование значительно упрощает этот процесс, поэтому стоит подумать, что выбрать: Seq, Elasticsearch или другого поставщика.

Я уже писал, как добавлять структурированные свойства в сообщения журнала, используя переменные и параметры из сообщения, но, как видно на рис. 26.11, видимых свойств гораздо больше, чем существует только в сообщении.

Области журналирования позволяют добавлять произвольные данные в сообщения журнала. Они доступны в некоторых поставщиках неструктурного журналирования, но предстают во всей своей красе при использовании с поставщиками структурного журналирования. В последнем разделе этой главы я продемонстрирую, как использовать их, чтобы добавить дополнительные данные в сообщения журнала.

26.5.2 Использование областей журналирования для добавления дополнительных свойств в сообщения журнала

В своих приложениях вы часто будете сталкиваться с тем, что у вас есть группа операций, которые используют одни и те же данные, которые было бы полезно прикрепить к сообщениям журнала. Например, у вас может быть серия операций с базой данных, которые используют один и тот же идентификатор транзакции, или вы можете выполнять несколько операций с одним и тем же идентификатором пользователя либо идентификатором рецепта. *Области журналирования* позволяют связывать одни и те же данные с каждым сообщением журнала в такой группе.

ОПРЕДЕЛЕНИЕ *Области журналирования* используются для группировки нескольких операций путем добавления одних и тех же данных в каждое сообщение журнала.

Области журналирования в ASP.NET Core создаются путем вызова `ILogger.BeginScope<T>(T state)` и предоставления данных состояния (`state`) для средства ведения журнала. Эти области создаются внутри блока `using`; любые сообщения журнала, записанные внутри блока, будут иметь связанные данные, а те, что находятся за его пределами, – нет.

Листинг 26.8 Добавление свойств области журналирования в сообщения журнала с помощью BeginScope

```
_logger.LogInformation("No, I don't have scope");
using(_logger.BeginScope("Scope value"))
{
    using(_logger.BeginScope(new Dictionary<string, object>
        {{ "CustomValue1", 12345 } }))
    {
        _logger.LogInformation("Yes, I have the scope!");
    }
    _logger.LogInformation("No, I lost it again");
```

Сообщения журнала, записанные вне блока области, не включают состояние области

Вызов `BeginScope` начинает блок области с состоянием «Scope value»

Вы можете передать что угодно в качестве состояния для области

Сообщения журнала, записанные внутри блока области, включают состояние области

Состояние этой области может быть любым объектом: например, целым числом, строкой или словарем. Реализация каждого поставщика должна решать, как обрабатывать состояние, которое вы предоставляете в вызове `BeginScope`, но обычно оно будет сериализовано с помощью метода `ToString()`.

СОВЕТ Наиболее частый вариант использования областей журналирования, с которым сталкивался я, – это добавление дополнительных пар «ключ–значение» в сообщения журнала. Чтобы добиться такого поведения в Seq и Serilog, необходимо передать `Dictionary<string, object>` в качестве объекта состояния.

Когда записываются сообщения журнала внутри блока, состояние области фиксируется и записывается как часть журнала, как показано на рис. 26.13. `Dictionary<>` добавляется непосредственно в сообщение журнала (`CustomValue1`), а оставшиеся значения состояния добавляются в свойство `Scope`. Вы, вероятно, найдете подход с использованием словаря более полезным, поскольку добавленные свойства гораздо легче фильтруются, как видно на рис. 26.12.

На этом мы подошли к концу главы, посвященной журналированию. Используете ли вы встроенных поставщиков журналирования или выберете стороннего поставщика, такого как Serilog или NLog, ASP.NET Core позволяет легко получать подробные сообщения журналов не только для кода своего приложения, но и для библиотек, составляющих инфраструктуру приложения, например Kestrel и EF Core. Какой бы вариант вы не выбрали, я призываю вас добавлять больше сообщений журналов, чем, как вы *думаете*, вам нужно, – в будущем вы скажете мне спасибо, когда дело дойдет до отслеживания проблем.

Состояние словаря добавляется в журнал в виде пар ключ–значение. Таким образом добавляется свойство `CustomValue1`

Другие значения состояния добавляются в свойство `Scope` в виде массива значений

Журналы, записанные вне блока scope, не имеют дополнительного состояния

Event	Level (Information)	Type	Raw JSON
07 Jun 2020 22:18:58.282	No, I lost it again	Type (0x9E792E6)	3c760ff-bef4-4f44-9aac-8f5202c21fe9 SeqLogger.Controllers.ScopesController.Get (SeqLogger) 12345
07 Jun 2020 22:18:58.282	Yes, I have the scope!		8898006a-8881-f280-b63f-84710c79670b /scopes [{"Scope_value"}] SeqLogger.Controllers.ScopesController [e1794b7b-4a0a8941e0175e60. e1794b7b-4a0a8943e0175e60]
07 Jun 2020 22:18:58.282	No, I don't have scope	Type (0xA080618D)	3c760ff-bef4-4f44-9aac-8f5202c21fe9 SeqLogger.Controllers.ScopesController.Get (SeqLogger) 0000000e-0001-f280-b63f-84710c79670b /scopes SeqLogger.Controllers.ScopesController [e1794b7b-4a0a8941e0175e60. e1794b7b-4a0a8943e0175e60]

Рис. 26.13 Добавление свойств в журналы с использованием областей журналирования. Состояние области, добавленное с использованием словаря, добавляется в виде свойств структурного журналирования, а другое состояние добавляется к свойству `Scope`. Добавление свойств упрощает связывание журналов друг с другом

В следующей главе мы рассмотрим наше приложение ASP.NET Core с другой точки зрения. Вместо того чтобы сосредотачиваться на коде и логике приложения, мы рассмотрим, как подготовить его к развертыванию в промышленном окружении. Вы узнаете, как указать URL-адреса, которые использует приложение, и как опубликовать его, чтобы его можно было разместить в IIS.

Резюме

- Журналирование имеет решающее значение для быстрой диагностики ошибок в рабочих приложениях. Следует всегда настраивать журналирование для своего приложения, чтобы журналы писались в надежное место.
- Вы можете добавить журналирование в свои сервисы, внедрив `ILogger<T>`, где `T` – это имя сервиса. Или можно внедрить `ILoggerFactory` и вызвать метод `CreateLogger()`.
- Уровень сообщения журнала указывает, насколько оно важно, и варьируется от `Trace` до `Critical`. Обычно вы будете создавать много неважных сообщений журнала и только несколько сообщений, представляющих ценность.
- Уровень сообщения указывается с помощью соответствующего метода расширения `ILogger` для создания журнала. Чтобы написать сообщение уровня `Information`, используйте `ILogger.LogInformation(message)`.
- Категория сообщения журнала указывает, какой компонент создал сообщение. Обычно задается как полное имя класса, создавшего сообщение, но вы можете использовать любое строковое значение, если хотите. `ILogger<T>` будет иметь категорию сообщения журнала `T`.
- Можно форматировать сообщения с помощью значений-заполнителей, аналогично методу `string.Format`, но с понятными именами параметров. Вызывая `logger.LogInformation("Loading Recipe with id {RecipeId}", 1234)`, вы создаете запись в журнале "Loading Recipe with id 1234", а также захватываете значение `RecipeId=1234`. Такое *структурное журналирование* значительно упрощает анализ сообщений журнала;
- ASP.NET Core включает множество готовых поставщиков журналирования. Это и поставщики консоли, отладки, `EventLog` и `Event-Source`. В качестве альтернативы можно добавить сторонних поставщиков.
- Вы можете настроить несколько экземпляров `ILoggerProvider` в ASP.
- NET Core, которые определяют, где выводятся журналы. Метод `CreateDefaultBuilder` добавляет поставщиков консоли и отладки, а вы можете добавить дополнительных поставщиков, вызвав метод `ConfigureLogging()`.

- Serilog – уже давно существующий фреймворк для журналирования с поддержкой большого количества мест назначения. Вы можете добавить его в свое приложение с помощью пакета Serilog. AspNetCore. Он заменяет ILoggerFactory по умолчанию на версию конкретно для Serilog.
- Вы можете контролировать избыточное число сообщений журналов с помощью конфигурации. Вспомогательный метод CreateDefaultBuilder использует секцию конфигурации "Logging", чтобы контролировать избыточность вывода. Обычно в промышленном окружении фильтруется больше сообщений журналов по сравнению с окружением разработки.
- Только одно правило фильтрации выбирается для каждого поставщика при определении, выводить ли сообщение журнала. Выбирается наиболее конкретное правило на основе поставщика и категории сообщения журнала.
- Структурное журналирование включает в себя запись сообщений журналов, чтобы их можно было легко запрашивать и фильтровать, вместо неструктурированного формата по умолчанию, который выводится в консоль. Благодаря этому проще анализировать журналы, искать проблемы и выявлять закономерности.
- Вы можете добавить дополнительные свойства в сообщение журнала, содержащее структурированные данные, используя блоки области журналирования. Они создаются путем вызова ILogger.BeginScope<T>(state) в блоке using. Состояние может быть любым объектом и добавляется ко всем сообщениям журнала внутри блока.

27

Публикация и развертывание приложения

В этой главе:

- публикация приложения ASP.NET Core;
- размещение приложения ASP.NET Core в IIS;
- настройка URL-адресов для приложения ASP.NET Core.

В этой книге мы рассмотрели очень много вопросов. Мы рассмотрели базовые механизмы создания приложения ASP.NET Core, среди которых – настройка внедрения зависимостей, загрузка настроек приложения и создание конвейера промежуточного ПО. Мы рассмотрели создание приложений с использованием минимальных API и веб-контроллеров, изучили пользовательский интерфейс с отрисовкой на стороне сервера, используя шаблоны и макеты Razor для формирования HTML-ответа, а также абстракции более высокого уровня, такие как EF Core и ASP.NET Core Identity, позволяющие взаимодействовать с базой данных и добавлять пользователей в приложение. В этой главе мы пойдем немного другим путем. Вместо того чтобы искать способы создания крупных и лучших приложений, мы сосредоточимся на том,

что значит развертывание приложения, чтобы пользователи могли получить к нему доступ.

Мы начнем с рассмотрения модели хостинга ASP.NET Core в разделе 27.1 и выясним, почему лучше размещать свое приложение за обратным прокси-сервером, вместо того чтобы предоставлять к нему доступ напрямую из интернета. Я покажу вам разницу между запуском приложения ASP.NET Core в окружении разработки с использованием команды `dotnet run` и публикацией приложения для использования на удаленном сервере. Наконец, я опишу некоторые варианты развертывания, которые можно использовать, при принятии решения о том, как и где развернуть приложение.

В разделе 27.2 я покажу вам, как развернуть приложение с использованием одного из вариантов: сервера Windows под управлением IIS (Internet Information Services). Это типичный сценарий развертывания для многих разработчиков, уже знакомых с ASP.NET, поэтому он послужит полезным примером, но, конечно же, это не единственная возможность.

Я не буду вдаваться во все технические подробности конфигурирования системы IIS, но предоставлю вам минимум информации, необходимой для запуска. Если вы специализируетесь на кросс-платформенной разработке, не волнуйтесь, я не буду слишком углубляться в IIS.

В разделе 27.3 я расскажу о хостинге в Linux. Вы увидите, чем он отличается от хостинга приложений в Windows, узнаете, какие изменения необходимо внести в свои приложения, и познакомитесь с подводными камнями, на которые следует обратить внимание. Я опишу, чем обратные прокси-серверы в Linux отличаются от IIS, и приведу ссылки на ресурсы, которые можно использовать для настройки своего окружения, вместо того чтобы давать исчерпывающие инструкции в этой книге.

Если вы *не* размещаете свое приложение с помощью IIS, вам, вероятно, потребуется задать URL-адрес, который использует ваше приложение ASP.NET Core при развертывании. В разделе 27.4 я покажу два способа сделать это: с помощью специальной переменной окружения `ASPNETCORE_URLS` и используя аргументы командной строки. Хотя, как правило, это не является проблемой во время разработки, установка правильных URL-адресов для приложения имеет решающее значение при развертывании.

В этой главе рассматривается относительно широкий круг тем, связанных с развертыванием приложения. Но, прежде чем перейти к деталям, я рассмотрю модель хостинга для ASP.NET.Core, раз уж мы здесь.

Она существенно отличается от модели хостинга предыдущей версии ASP.NET, поэтому, если вы ориентируетесь на эти знания, лучше попытаться забыть то, что вы знаете!

27.1 Что такое модель хостинга ASP.NET Core

Если вернуться к первой главе, то вы, наверное, помните, что мы обсуждали модель хостинга ASP.NET Core. Приложения ASP.NET Core, по сути, консольные. В них есть метод `static void Main`, служащий точкой входа для приложения, как у стандартного консольного приложения .NET.

ПРИМЕЧАНИЕ Точка входа для программ, использующих операторы верхнего уровня, автоматически создается компилятором. Она не называется Main (обычно она имеет «недопустимое» имя, например <Main>\$), но в остальном она имеет ту же сигнатуру, что и классическая, – статический метод void Main, который вы могли бы написать вручную.

Что делает приложение приложением ASP.NET Core, так это то, что оно работает на веб-сервере, обычно Kestrel, внутри процесса консольного приложения. Kestrel предоставляет функциональность HTTP, чтобы получать запросы и возвращать ответы клиентам. Kestrel передает все полученные запросы в код вашего приложения, чтобы сгенерировать ответ, как показано на рис. 27.1.

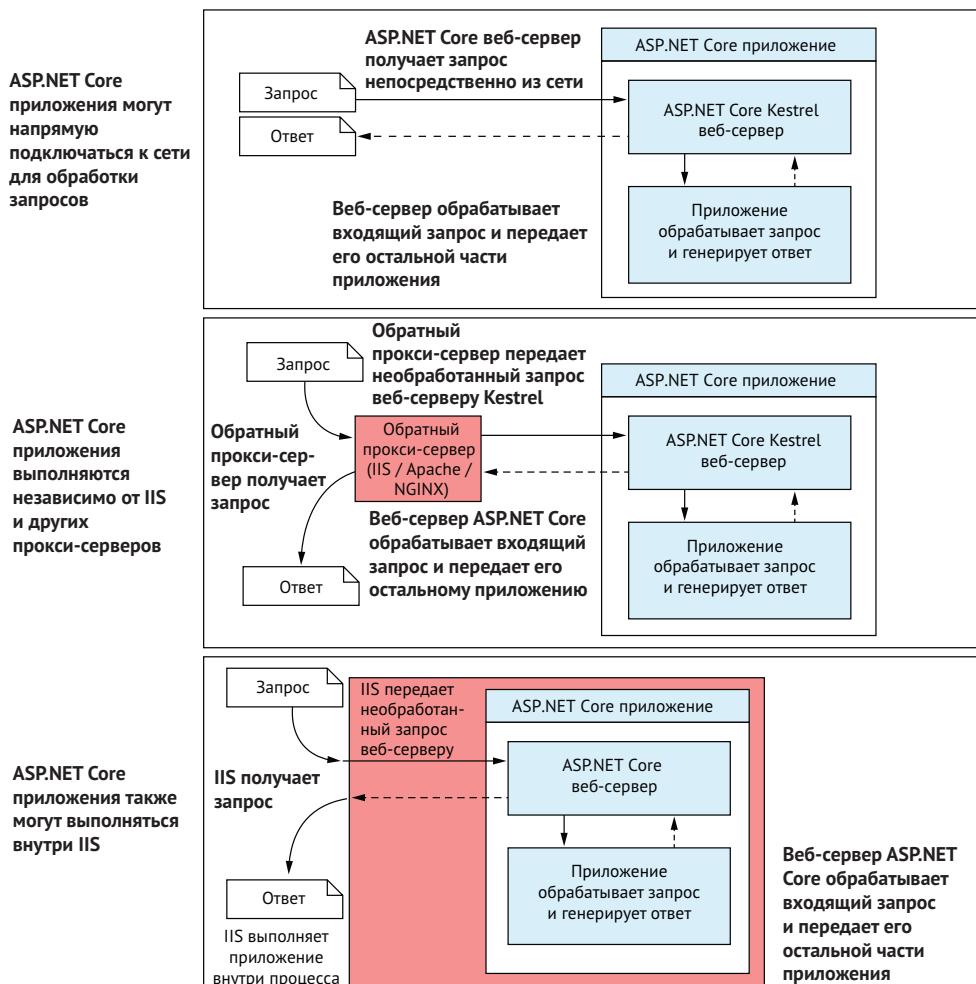


Рис. 27.1 Модель хостинга ASP.NET Core. Запросы принимаются обратным прокси-сервером и перенаправляются на веб-сервер Kestrel. Одно и то же приложение может работать за разными обратными прокси-серверами без изменений

Такая модель хостинга отделяет сервер и обратный прокси от самого приложения, чтобы одно и то же приложение могло без изменений в коде работать в нескольких окружениях.

В этой книге мы сосредоточились на нижней половине рис. 27.1 – самом приложении ASP.NET.Core, но на самом деле вам часто придется размещать свое приложение за обратным прокси-сервером, например IIS в Windows или NGINX или Apache в Linux. *Обратный прокси-сервер* – это программа, которая слушает HTTP-запросы из интернета, а затем отправляет запросы в ваше приложение, как если бы запрос шел напрямую из интернета.

ОПРЕДЕЛЕНИЕ *Обратный прокси-сервер* – это программное обеспечение, отвечающее за получение запросов и их пересылку на соответствующий веб-сервер с приложением. Обратный прокси-сервер доступен непосредственно из сети Интернет, тогда как основной веб-сервер доступен только для прокси.

Если вы запускаете свое приложение с помощью модели «Платформа как услуга» (PaaS), например Azure App Service, вы также используете обратный прокси-сервер, которым управляет Azure. Использование обратного прокси-сервера имеет много преимуществ:

- **безопасность** – обратные прокси-серверы специально разработаны для защиты от вредоносного интернет-трафика, поэтому они, как правило, хорошо протестированы;
- **производительность** – можно настроить обратные прокси-серверы для повышения производительности путем агрессивного кеширования ответов на запросы;
- **управление процессами** – к сожалению, приложения иногда дают сбой. Некоторые обратные прокси-серверы могут работать как мониторы или планировщики, чтобы гарантировать, что в случае сбоя приложения прокси-сервер сможет автоматически перезапустить его;
- **поддержка нескольких приложений** – обычно на одном сервере работают несколько приложений. Использование обратного прокси-сервера упрощает поддержку такого сценария, применяя имя хоста приложения, чтобы решить, какое приложение должно получить запрос.

Я не хочу, чтобы казалось, будто когда вы используете обратный прокси-сервер, то все солнечно и радужно. Здесь есть и свои недостатки:

- **сложность** – одна из самых больших жалоб заключается в том, насколько сложными могут быть обратные прокси-серверы. Если вы сами управляете прокси (вместо того чтобы полагаться на реализацию PaaS), то может возникнуть множество трудностей;
- **межпроцессное взаимодействие** – для большинства обратных прокси-серверов требуются два процесса: обратный прокси и ваше веб-приложение. Обмен данными между ними часто происходит медленнее, чем если бы вы напрямую предоставили доступ к своему веб-приложению для запросов из интернета;

- *ограниченные функции* – не все обратные прокси-серверы поддерживают те же функции, что и приложение ASP.NET Core. Например, Kestrel поддерживает протокол HTTP/2, но если ваш обратный прокси-сервер не поддерживает его, то преимущества вы не увидите.

Независимо от того, решите вы использовать обратный прокси-сервер или нет, когда приходит время разместить свое приложение, вы не можете копировать файлы кода прямо на сервер. Сначала нужно *опубликовать* приложение, чтобы оптимизировать его для промышленного окружения. В разделе 27.1.1 мы сравним создание приложения ASP.NET Core для машины разработчика с публикацией, необходимой для запуска на сервере.

27.1.1 Запуск и публикация приложения ASP.NET Core

Одно из ключевых изменений в ASP.NET Core по сравнению с предыдущими версиями ASP.NET состоит в том, что теперь вам проще создавать приложения с помощью своих любимых редакторов кода и интегрированной среды разработки. Ранее для разработки ASP.NET требовалась Visual Studio, но сейчас, используя интерфейс командной строки .NET и плагин OmniSharp, можно создавать приложения с помощью удобных инструментов на любой платформе.

В результате, независимо от того, используете вы Visual Studio или интерфейс командной строки .NET, под капотом используются одни и те же инструменты. Visual Studio предоставляет дополнительный графический интерфейс, функциональность и оболочки для создания вашего приложения, но за кулисами выполняет те же команды, что и интерфейс командной строки .NET.

Напомню, что пока вы использовали четыре основные команды .NET CLI для создания своих приложений:

- `dotnet new` – создает приложение ASP.NET Core из шаблона;
- `dotnet restore` – скачивает и устанавливает все указанные пакеты NuGet для вашего проекта;
- `dotnet build` – компилирует и собирает проект;
- `dotnet run` – запускает ваше приложение, чтобы вы могли отправлять ему запросы.

Если вы когда-либо создавали приложение .NET, будь то приложение ASP.NET или консольное приложение .NET Framework, то знаете, что результат процесса сборки записывается в папку `bin`. То же самое происходит и в случае с приложениями ASP.NET Core.

Если ваш проект успешно компилируется при вызове команды `dotnet build`, .NET CLI запишет его вывод в папку `bin` в каталоге вашего проекта. Внутри этой папки есть несколько файлов, необходимых для запуска вашего приложения, включая файл с расширением `.dll`, содержащий код приложения. На рис. 27.2 показана часть папки `bin` для приложения ASP.NET Core.

ПРИМЕЧАНИЕ В Windows у вас также будет исполняемый файл с расширением `.exe`, `ExampleApp.exe`. Это простой файл-оболочка,

который упрощает запуск приложения, содержащегося в файле ExampleApp.dll.

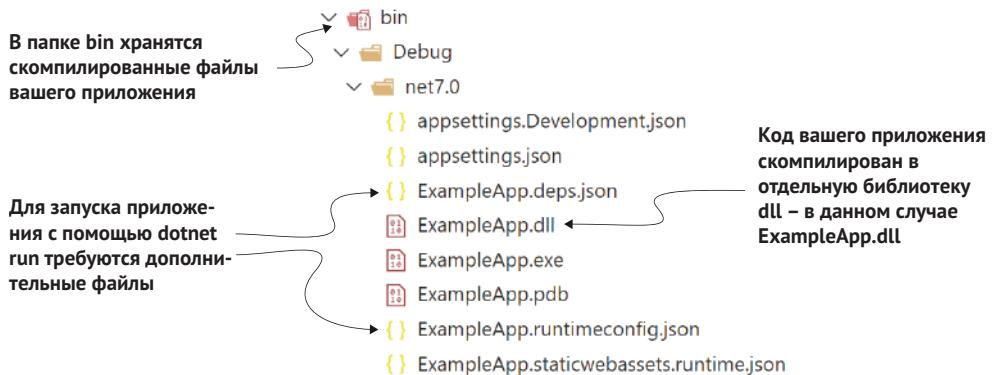


Рис. 27.2 Папка bin для приложения ASP.NET Core после запуска команды `dotnet build`. Приложение скомпилировано в один файл с расширением .dll, ExampleApp.dll

Когда вы вызываете команду `dotnet run` в папке проекта (или запускаете приложение с помощью Visual Studio), интерфейс командной строки .NET использует файл с расширением .dll для запуска приложения. Но он не содержит всей информации, необходимой для развертывания.

Чтобы разместить и развернуть приложение на сервере, сначала необходимо его опубликовать. Это можно сделать из командной строки с помощью команды `dotnet publish`. Она собирает и упаковывает все, что нужно вашему приложению для запуска. Команда упаковывает приложение из текущего каталога и собирает его во вложенную папку `publish`.

Я использовал конфигурацию `Release` вместо конфигурации `Debug` по умолчанию, поэтому вывод будет полностью оптимизирован для запуска в промышленном окружении:

```
dotnet publish --output publish --configuration Release
```

СОВЕТ Всегда используйте конфигурацию `release` при публикации приложения для развертывания. Это гарантирует, что компилятор генерирует оптимизированный код.

После завершения команды вы найдете опубликованное приложение в папке `publish`, как показано на рис. 27.3.

Как видите, файл `ExampleApp.dll` все еще здесь, наряду с некоторыми дополнительными файлами. В частности, процесс публикации скопировал папку статических файлов `wwwroot`. При локальном запуске вашего приложения с помощью команды `dotnet run` интерфейс командной строки .NET использует эти файлы напрямую из папки проекта вашего приложения. При запуске команды `dotnet publish` файлы копируются в выходной каталог, чтобы они были включены при развертывании приложения на сервере.

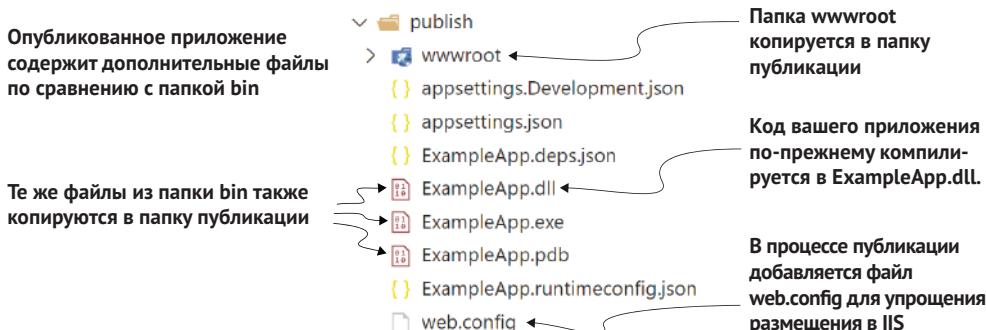


Рис. 27.3 Папка publish для приложения после запуска команды `dotnet publish`. Приложение по-прежнему компилируется в один файл с расширением .dll, но все дополнительные файлы, такие как wwwroot и appsettings.json, также копируются в целевую папку

Если первое, о чём вы инстинктивно подумали, – попробовать запустить приложение в папке publish, используя команду `dotnet run`, которую вы уже знаете и любите, то вас ждёт разочарование. Вместо запуска приложения вы увидите сбивающее с толку сообщение: **Couldn't find a project to run** (Не удалось найти проект для запуска).

Чтобы запустить опубликованное приложение, нужно использовать другую команду. Вместо вызова команды `dotnet run` вы должны вызвать команду `dotnet` с путем к DLL-файлу приложения. Если вы запускаете команду из папки publish, то для примера приложения на рис. 27.3 это будет выглядеть примерно так:

```
dotnet ExampleApp.dll
```

Это команда, которую ваш сервер будет выполнять при запуске приложения в промышленном окружении.

СОВЕТ Вы также можете использовать команду `dotnet exec` для достижения той же цели, например `dotnet exec exampleApp.dll`. Это делает доступными некоторые расширенные параметры времени выполнения, как описано в документации по адресу <http://mng.bz/x4d8>.

Во время разработки команда `dotnet run` выполняет всю работу, чтобы облегчить вам задачу: она проверяет, создано ли приложение, ищет файл проекта в текущей папке, выясняет, где будут находиться соответствующие файлы с расширением .dll (в папке bin), и, наконец, запускает приложение.

В промышленном окружении вся эта дополнительная работа вам не нужна. Ваше приложение уже собрано, нужно только запустить его. Команда `dotnet <dll>` делает только это, поэтому приложение запускается намного быстрее.

ПРИМЕЧАНИЕ Команда `dotnet`, используемая для запуска опубликованного приложения, является частью среды выполнения .NET,

тогда как команда `dotnet`, используемая для сборки и запуска вашего приложения во время разработки, является частью .NET SDK.

Развертывания, зависящие от платформы, и автономные развертывания

Приложения .NET Core можно развертывать двумя способами: это развертывания, зависящие от среды выполнения (RDD), и автономные развертывания (SCD).

В большинстве случаев вы будете использовать RDD. Оно зависит от среды выполнения .NET 7, устанавливаемой на целевой машине, на которой запущено ваше опубликованное приложение, но вы можете запускать его на любой платформе – Windows, Linux или macOS – без необходимости повторной компиляции.

Напротив, SCD содержит весь код, необходимый для запуска вашего приложения, поэтому на целевой машине не обязательно должна быть установлена .NET 7. Вместо этого при публикации вашего приложения среда выполнения .NET 7 будет упакована с кодом и библиотеками приложения.

У каждого подхода есть свои плюсы и минусы, но в большинстве случаев я отдаю предпочтение RDD. Окончательный размер RDD намного меньше, поскольку файлы в этом случае содержат только код приложения, а не всю платформу .NET 7, как в SCD. Кроме того, вы можете развернуть свои RDD-приложения на любой платформе, тогда как SCD-приложения должны быть скомпилированы специально для операционной системы целевой машины, например 64-разрядной Windows 10 или 64-разрядной Red Hat Enterprise Linux.

Тем не менее SCD отлично подходит для изоляции вашего приложения от зависимостей на машине, используемой для хостинга. SCD не зависит от версии .NET, установленной у провайдера хостинга, поэтому можно, например, использовать предварительные версии .NET в Azure App Service, даже если там эти версии не поддерживаются.

В этой книге я обсуждаю RDD только для простоты, но если вы хотите создать SCD, просто укажите идентификатор среды выполнения при публикации, в данном случае 64-разрядную версию Windows 10:

```
dotnet publish -c Release -r win10-x64 --self-contained -o publish_folder
```

Вывод будет содержать файл с расширением .exe, который является вашим приложением, и огромное количество файлов платформы .NET 7 с расширением .dll (около 65 МБ, если говорить о приложении, используемом в качестве примера). Вам нужно развернуть всю эту папку на целевой машине для запуска своего приложения. В .NET 7 можно уменьшить некоторые из этих сборок в процессе публикации, но в некоторых сценариях это сопряжено с рисками. Дополнительные сведения см. в документе Microsoft «Обзор публикации приложений .NET Core» по адресу <https://docs.microsoft.com/dotnet/core/deploying/>.

Мы пришли к тому, что публикация приложения важна для его подготовки к запуску в промышленном окружении, но как его развернуть? Как отправить файлы со своего компьютера на сервер, чтобы пользователи могли получить доступ к вашему приложению? Для этого есть очень много вариантов, поэтому в следующем разделе я приведу краткий список подходов, которые следует рассмотреть.

27.1.2 Выбор метода развертывания для приложения

Для развертывания любого приложения в промышленном окружении обычно есть два основных требования:

- сервер, на котором можно запускать приложение;
- средство загрузки приложения на сервер.

Поначалу развертывание приложения в промышленном окружении представляло собой трудоемкий и подверженный ошибкам процесс, и для многих такая ситуация по-прежнему актуальна. Если вы работаете в компании, которая не меняла практики в течение последних лет, то вам может потребоваться запросить сервер или виртуальную машину для своего приложения и предоставить его команде эксплуатации, которая подготовит его за вас. В этом случае у вас могут быть связаны руки и некоторые сценарии будут для вас недоступны.

Для тех, кто выбрал непрерывную интеграцию (CI) или непрерывную доставку/развертывание (CD), есть больше возможностей. CI/CD – это процесс обнаружения изменений в вашей системе управления версиями (например, Git, SVN, Mercurial, Team Foundation Version Control) и автоматическая сборка и потенциальное развертывание приложения на сервере практически без вмешательства человека.

ПРИМЕЧАНИЕ Между этими терминами есть тонкие, но важные отличия. У Atlassian есть хорошая сравнительная статья «Непрерывная интеграция, непрерывная доставка и непрерывное развертывание»: <http://mng.bz/vzp4>.

Можно найти много разных систем непрерывной интеграции или непрерывного развертывания: Azure DevOps, GitHub Actions, Jenkins, TeamCity, AppVeyor, Travis и Octopus Deploy, и это лишь малая часть. Каждая из них может управлять некоторыми или всеми процессами CI/CD и может быть интегрирована со многими другими системами.

Вместо того чтобы рекомендовать какую-либо конкретную систему, я предлагаю опробовать некоторые из доступных сервисов и посмотреть, какой из них лучше всего вам подходит. Некоторые лучше подходят для проектов с открытым исходным кодом, другие – для развертывания в облачных сервисах – все зависит от конкретной ситуации.

Если вы только начинаете работать с ASP.NET Core и не хотите проходить через процесс настройки непрерывной интеграции, у вас все еще остается много вариантов. Самый простой способ – использовать встроенные параметры развертывания Visual Studio. Они доступны в Visual Studio через пункт меню **Build > Publish AppName**, который представляет собой форму, показанную на рис. 27.4.

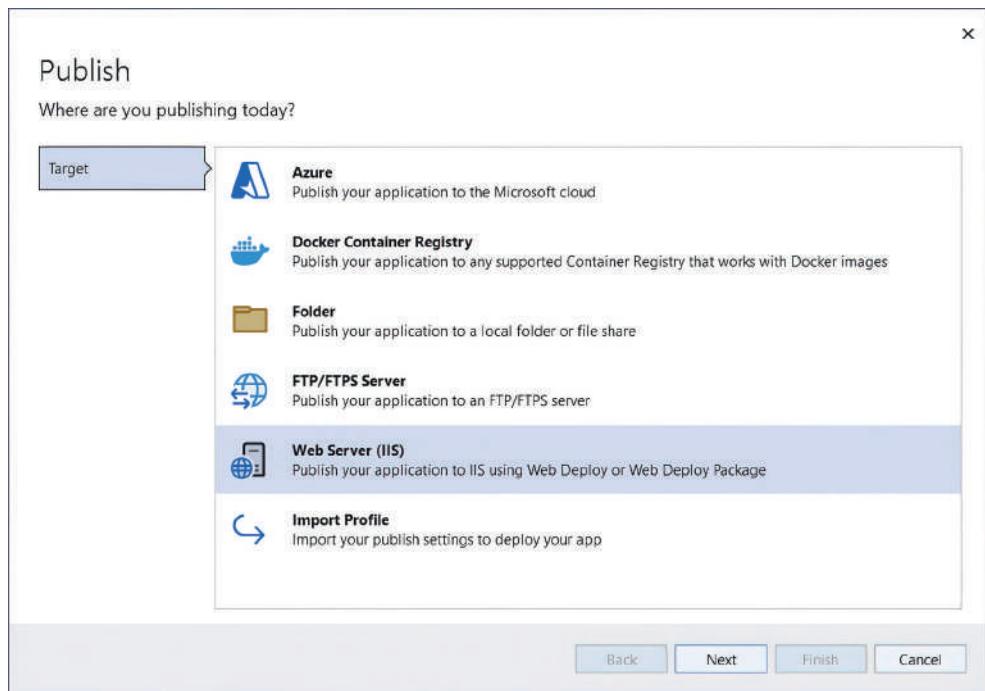


Рис. 27.4 Экран публикации приложения в Visual Studio 2022. Здесь представлены простые варианты публикации приложения напрямую в Azure App Service, IIS, на FTP-сайт или в папку на локальном компьютере

Здесь вы можете публиковать свое приложение прямо из Visual Studio в разные места. Все это замечательно, но я рекомендую присмотреться к более автоматизированному и контролируемому подходу для крупных приложений или когда над одним приложением работает целая команда.

Учитывая количество доступных возможностей и скорость, с которой эти варианты меняются, в этой главе я сосредоточусь на одном конкретном сценарии: вы создали приложение ASP.NET Core, и вам нужно его развернуть. У вас есть доступ к серверу Windows, который уже обслуживает устаревшую версию приложения ASP.NET (.NET Framework) с помощью IIS, и вы хотите запустить приложение ASP.NET Core наряду с ним.

В следующем разделе вы увидите обзор шагов, необходимых для запуска приложения ASP.NET Core в промышленном окружении, используя IIS в качестве обратного прокси-сервера. Это не мастер-класс по настройке IIS (в этом продукте, появившемся 20 лет назад, так много всего, что я бы просто не знал, с чего начать!), но я расскажу об основах, необходимых для того, чтобы ваше приложение обслуживало запросы.

СОВЕТ Рекомендации по выбору параметров публикации Visual Studio см. в документации Microsoft «Развертывание приложения в папке, IIS, Azure или другом месте назначения» по адресу <http://mng.bz/4Z8j>.

27.2 Публикация приложения в IIS

В этом разделе я вкратце покажу, как опубликовать свое первое приложение на IIS. Вы добавите пул приложений и веб-сайт в IIS и убедитесь, что ваше приложение имеет необходимую конфигурацию для работы с IIS в качестве обратного прокси-сервера. Само развертывание будет таким же простым, как и копирование опубликованного приложения в папку хостинга IIS.

В разделе 27.1 вы узнали о необходимости публикации приложения перед развертыванием и преимуществах использования обратного прокси-сервера при запуске приложения ASP.NET Core в промышленном окружении. Если вы разворачиваете свое приложение в Windows, IIS будет вашим обратным прокси-сервером и будет отвечать за управление приложением.

IIS – старая и сложная штука, и я не могу охватить в этой книге все, что связано с его настройкой. Да вы бы и не хотели, чтобы я это сделал, – это было бы очень скучно! Вместо этого в этом разделе я предложу обзор основных требований для запуска ASP.NET Core в IIS, наряду с изменениями, которые вам, возможно, потребуется внести в свое приложение для поддержки IIS.

Если вы работаете в Windows и хотите опробовать эти шаги локально, вам придется вручную включить IIS на машине, используемой для разработки. Если вы делали это в более старых версиях Windows, то ничего особо не изменилось. Пошаговое руководство по настройке IIS и советы по устранению неполадок можно найти в документации по ASP.NET Core на странице <http://mng.bz/6g2R>.

27.2.1 Конфигурирование IIS для ASP.NET Core

Первым шагом в подготовке IIS для размещения приложений ASP.NET Core является установка пакета ASP.NET Core Windows Hosting Bundle (<http://mng.bz/opED>). Он включает в себя несколько компонентов, необходимых для запуска приложений .NET:

- *среда выполнения .NET* – запускает приложение .NET 7;
- *среда выполнения ASP.NET Core* – требуется для запуска приложений .NET Core;
- *модуль IIS AspNetCore* – обеспечивает связь между IIS и вашим приложением, чтобы IIS мог работать как обратный прокси-сервер.

Если вы собираетесь запускать IIS на машине для разработки, убедитесь, что вы установили пакет, иначе будете получать странные ошибки от IIS.

СОВЕТ Пакет хостинга Windows предоставляет все необходимое для запуска ASP.NET Core в IIS в Windows. Если вы размещаете свое приложение в Linux или Mac или не используете IIS в Windows, вам необходимо установить только среду выполнения .NET и среду выполнения ASP.NET Core для запуска приложений ASP.NET Core, зависящих от среды выполнения. Обрати-

те внимание, что вам нужно установить модуль IIS AspNetCore, даже если вы используете SCD.

После установки пакета вам необходимо настроить *пул приложений* в IIS для приложений ASP.NET Core. Предыдущие версии ASP.NET работали в *управляемом* пуле приложений, который использовал .NET Framework, но для ASP.NET Core нужно создать *неуправляемый* (*No Managed Code*) пул. Модуль ASP.NET Core запускается внутри пула, который загружает саму среду выполнения .NET 7.

ОПРЕДЕЛЕНИЕ Пул приложений в IIS представляет собой процесс приложения. Вы можете запускать каждое приложение в IIS в отдельном пуле приложений, чтобы изолировать их друг от друга.

Чтобы создать неуправляемый пул приложений, щелкните правой кнопкой мыши **Application Pools** (Пулы приложений) в IIS и выберите **Add Application Pool** (Добавить пул приложений). В появившемся диалоговом окне укажите имя для пула приложений, например NetCore, и выберите в поле **.NET CLR version** значение **No Managed Code**, как показано на рис. 27.5.

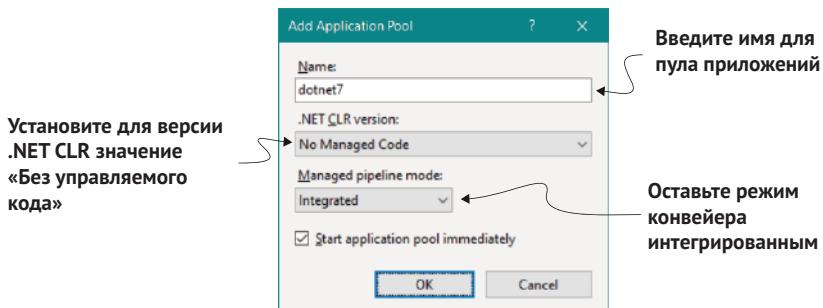
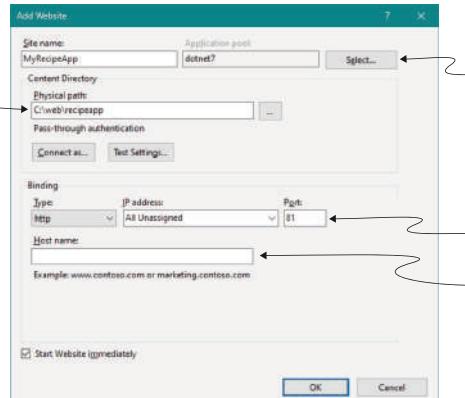


Рис. 27.5 Создание пула приложений в IIS для вашего приложения. В поле .NET CLR version должно быть выбрано значение No Managed Code

Теперь, когда у вас есть пул приложений, можно добавить новый веб-сайт в IIS. Щелкните правой кнопкой мыши по узлу *Сайты* и выберите **Add Website** (Добавить веб-сайт). В диалоговом окне «Добавить веб-сайт», показанном на рис. 27.6, вы указываете имя веб-сайта и путь к папке, в которой будете публиковать свой сайт. Я создал папку, которую буду использовать для развертывания приложения Recipie из предыдущих глав. Важно изменить пул приложений на новый созданный вами пул приложений NetCore. В промышленном окружении также нужно указать имя хоста для приложения, но я пока оставил его пустым и изменил порт на 81, поэтому приложение будет привязано к URL-адресу <http://localhost:81>.

ПРИМЕЧАНИЕ При развертывании приложения в промышленном окружении необходимо зарегистрировать имя хоста у регистратора домена, чтобы ваш сайт был доступен для пользователей в интернете. Используйте это имя хоста при настройке приложения в IIS, как показано на рис. 27.6.

Введите путь к папке, в которой вы будете публиковать свое приложение



Измените пул приложений на пул «Без управляемого кода»

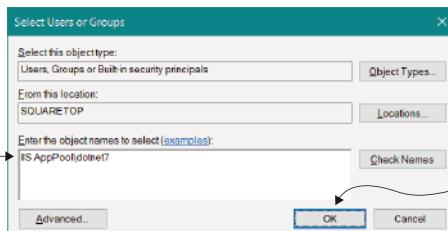
В промышленном окружении вы, скорее всего, оставите порт 80 и должны будете ввести имя хоста

Рис. 27.6 Добавление нового веб-сайта в IIS для своего приложения. Обязательно смените значение Application Pool на No Managed Code pool, созданный на предыдущем этапе. Также укажите имя, путь, по которому вы будете публиковать файлы приложения, и URL-адрес, который IIS будет использовать для вашего приложения

После того как вы нажмете **OK**, IIS создаст приложение и попытается запустить его. Но вы не опубликовали приложение в папке, поэтому пока не можете открыть его в браузере.

Прежде чем вы сможете опубликовать и запустить приложение, необходимо выполнить еще один важный шаг настройки: вы должны предоставить разрешения для пула приложений NetCore, чтобы получить доступ к папке опубликованного приложения. Для этого щелкните правой кнопкой мыши по папке, в которой будет размещено приложение в проводнике Windows, и выберите **Properties** (Свойства). В диалоговом окне **Properties** выберите **Security > Edit > Add** (Безопасность > Изменить > Добавить). Введите в текстовое поле IIS AppPool\NetCore, как показано на рис. 27.7, где NetCore – это имя вашего пула приложений, и нажмите **OK**. Закройте все диалоговые окна, нажав кнопку **OK**. Все готово.

Введите IIS AppPool, а затем имя пула приложений без управляемого кода, например IIS AppPool\dotnet7



Нажмите **OK**, чтобы добавить разрешение

Рис. 27.7 Добавление разрешений для пула приложений dotnet7 в папку публикации веб-сайта

27.2.2 Подготовка и публикация приложения в IIS

Как мы обсуждали в разделе 27.1, IIS действует как обратный прокси-сервер для вашего приложения ASP.NET Core. Это означает, что IIS должен иметь возможность напрямую обмениваться данными с вашим приложением для пересылки входящих запросов и исходящих ответов из вашего приложения.

IIS использует для этой цели модуль ASP.NET Core, но между IIS и вашим приложением требуется определенная степень согласования. Чтобы все работало правильно, необходимо настроить приложение для использования интеграции с IIS.

Интеграция с IIS добавляется по умолчанию при применении вспомогательного метода `IHostBuilder.ConfigureWebHostDefaults()`, используемого в шаблонах по умолчанию. Если вы настраиваете собственный `HostBuilder`, то необходимо убедиться, что вы добавили интеграцию с IIS с методами расширения `UseIIS()` или `UseIISIntegration()`.

- `UseIIS()` настраивает ваше приложение для поддержки IIS с внутрипроцессной моделью размещения.
- `UseIISIntegration()` настраивает ваше приложение для поддержки IIS с моделью внепроцессного размещения.

Эти методы автоматически вызываются `WebApplicationBuilder`, но если вы не используете свое приложение с IIS, методы `UseIIS()` и `UseIISIntegration()` не окажут никакого влияния на ваше приложение, поэтому их в любом случае безопасно включать.

Внутрипроцессный и внепроцессный хостинг в IIS

Описание обратного прокси-сервера, которое я дал в разделе 27.1, предполагает, что ваше приложение выполняется в отдельном процессе. Это касается ситуации, если вы работаете в Linux и вплоть до выхода ASP.NET Core версии 3.0 для IIS это был вариант по умолчанию.

В ASP.NET Core версии 3.0 ASP.NET Core перешел на использование модели внепроцессного хостинга по умолчанию для приложений, развернутых в IIS. В этой модели IIS размещает ваше приложение напрямую в процессе IIS, уменьшая межпроцессное взаимодействие и повышая производительность.

Вы можете переключиться на модель внепроцессного хостинга с IIS, если хотите, что иногда может быть полезно для устранения неполадок. У Рика Штраля есть отличный пост, посвященный различиям между моделями хостинга. В нем рассказывается о том, как переключаться между ними, и о преимуществах каждой из этих моделей: <http://mng.bz/QmEv>.

В целом не нужно беспокоиться о различиях между моделями хостинга, но о них нужно знать, если вы развертываете приложение в IIS. Если вы решите использовать внепроцессную модель хостинга, то следует использовать метод расширения `UseIISIntegration()`. Если вы используете внутрипроцессную модель, используйте метод `UseIIS()`. В качестве альтернативы можно перестраховаться и задействовать оба – правильный метод расширения будет активирован в зависимости от модели хостинга, используемой в промышленном окружении. Если вы не используете IIS, то эти методы ничего делать не будут.

При запуске приложения в IIS эти методы расширения настраивают ваше приложение для сопряжения с IIS, чтобы оно могло беспрепятственно принимать запросы. Помимо прочего, эти методы делают следующее:

- определяют URL-адрес, который IIS будет использовать для пересылки запросов в ваше приложение, и настраивают его на прослушивание на этом URL-адресе;
- настраивают приложение для интерпретации запросов, поступающих от IIS, как идущих от клиента, путем настройки пересылки заголовков;
- активируют аутентификацию Windows, если требуется.

Добавление этих методов расширения – единственное изменение, которое нужно внести в приложение, чтобы иметь возможность разместить его в IIS, но есть еще один аспект, о котором нужно знать, когда вы публикуете свое приложение.

Как и в предыдущих версиях ASP.NET, IIS использует файл web.config для настройки приложений, которые он запускает. Важно, чтобы приложение содержало этот файл, когда оно публикуется на IIS; в противном случае вы можете нарушить поведение или даже предоставить доступ к закрытым файлам.

COBET Дополнительные сведения об использовании файла web.config для настройки модуля Asp.Net Core см. в статье Microsoft «Модуль ASP.NET Core»: <http://mng.bz/Xdna>.

Если ваш проект ASP.NET Core уже включает в себя файл web.config, .NET CLI или Visual Studio скопируют его в каталог публикации при публикации приложения. В противном случае команда publish создаст его за вас. Если вам не нужно настраивать этот файл, лучше не включать его в свой проект и позволить интерфейсу командной строки создать его за вас.

После этих изменений вы, наконец, можете опубликовать свое приложение в IIS. Опубликуйте его в папку либо из Visual Studio, либо с помощью интерфейса командной строки .NET:

```
dotnet publish --output publish_folder --configuration Release
```

Так вы опубликуете свое приложение в папке publish_folder. Затем вы можете скопировать его в путь, указанный в IIS, как показано на рис. 27.6.

На данный момент, если все прошло гладко, вы можете перейти по URL-адресу, указанному для вашего приложения (в моем случае это <http://localhost:81>), и посмотреть, как оно работает, что показано на рис. 27.8.

И вот оно, ваше первое приложение, работающее за обратным прокси-сервером. Хотя ASP.NET Core использует другую модель хостинга по сравнению с предыдущими версиями ASP.NET, процесс настройки IIS аналогичен.

Как это часто бывает при развертывании, ваш успех во многом зависит от конкретного окружения и самого приложения. Если после этих шагов вы обнаружите, что не можете запустить приложение, я настоятельно рекомендую обратиться к документации на странице <https://docs.microsoft.com/aspnet/core/publishing/iis>. Там содержится множество шагов по устранению неполадок, которые помогут вам вернуться в нужное русло, если IIS решит взбрыкнуть.

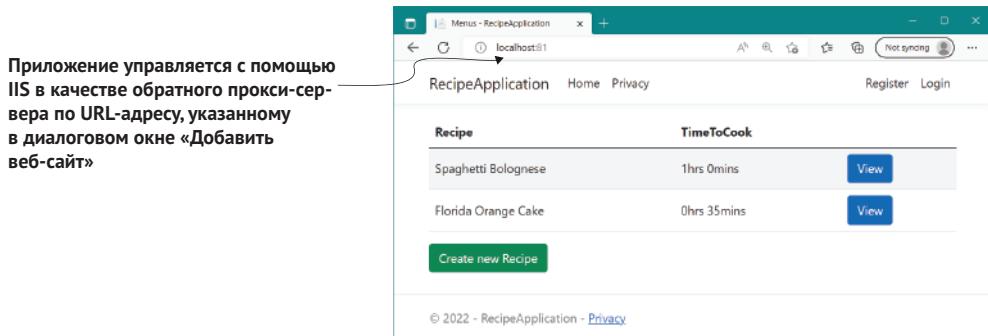


Рис. 27.8 Опубликованное приложение, использующее IIS в качестве обратного прокси-сервера, прослушивает запросы по адресу `http://localhost:81`

Этот раздел был специально создан для описания развертывания приложения в IIS, поскольку он обеспечивает отличный переход для разработчиков, которые уже привыкли к развертыванию приложений ASP.NET и хотят развернуть свое первое приложение ASP.NET Core. Но нельзя сказать, что IIS – единственное или лучшее место для хостинга.

В следующем разделе я кратко расскажу о размещении вашего приложения в Linux, за обратным прокси-сервером, таким как NGINX или Apache. Я не буду вдаваться в настройку самого сервера, но предо-ставлю обзор вещей, которые следует учитывать, и ресурсов, которые вы можете использовать для запуска своих приложений в Linux.

27.3 Размещение приложения в Linux

Одной из замечательных новых функций ASP.NET Core является возможность разработки и развертывания кросс-платформенных приложений, запускаемых на Windows, Linux или macOS. Способность работать в Linux, в частности, открывает возможность для менее затратных развертываний в облачном хостинге, развертываний на небольших устройствах вроде Raspberry Pi или в Docker-контейнерах.

Одной из отличительных черт Linux является то, что ее можно настраивать практически до бесконечности. Однако это может быть крайне сложно, особенно если вы пришли из мира инструментов под названием «мастер» (wizard) и графических интерфейсов Windows. В этом разделе содержится обзор того, что нужно для запуска приложения в Linux. Здесь описаны общие шаги, а не утомительные детали самой конфигурации. Я приведу список ресурсов, к которым вы можете обратиться при необходимости.

27.3.1 Запуск приложения ASP.NET Core за обратным прокси-сервером в Linux

Вы будете рады услышать, что запуск приложения в Linux во многом аналогичен запуску приложения в Windows с помощью IIS.

- 1 *Опубликуйте свое приложение с помощью команды dotnet publish.* Если вы создаете RDD, результат будет тот же, что и с IIS. В случае с SCD вы должны предоставить идентификатор целевой платформы, как описано в разделе 27.1.1.
- 2 *Установите необходимые компоненты на сервер.* Для развертывания RDD нужно установить среду выполнения .NET 7 и необходимые компоненты. Подробности см. в документации Microsoft: <https://docs.microsoft.com/en-gb/dotnet/core/install/linux>.
- 3 *Скопируйте свое приложение на сервер.* Вы можете использовать любой механизм, который вам нравится: FTP, USB-накопитель и все, что можно использовать для загрузки файлов на сервер!
- 4 *Настройте обратный прокси-сервер и укажите ему на свое приложение.* Как вы уже знаете, у вас может возникнуть желание запустить приложение за обратным прокси-сервером по причинам, описанным в разделе 27.1. В Windows вы бы использовали IIS, но в Linux у вас есть больше возможностей: NGINX, Apache или HAProxy.
- 5 *Настройте инструмент управления процессами для своего приложения.* В Windows IIS действует как обратный прокси-сервер и диспетчер процессов, перезапуская приложение в случае сбоя или если оно перестает отвечать. В Linux обычно требуется настроить отдельный диспетчер процессов для выполнения этих обязанностей; обратные прокси-серверы не сделают этого за вас.

Первые три пункта обычно одинаковы, независимо от того, работаете вы в Windows с IIS или Linux, но вот последние два пункта более интересны.

В отличие от монолитного IIS, Linux придерживается философии небольших приложений, у каждого из которых единственная ответственность.

IIS работает на том же сервере, что и ваше приложение, и выполняет несколько задач – проксирует трафик из интернета в ваше приложение, а также отслеживает работу самого приложения. Если ваше приложение дает сбой или перестает отвечать, IIS перезапустит процесс, чтобы продолжить обрабатывать запросы.

В Linux обратный прокси-сервер может работать на том же сервере, что и ваше приложение, но также часто бывает, что он работает полностью на другом сервере, как показано на рис. 27.9. То же самое относится и к ситуации, если вы решите развернуть свое приложение в Docker; обычно приложение развертывается в контейнере без обратного прокси-сервера, а обратный прокси на сервере будет указывать на ваш контейнер Docker.

Поскольку обратные прокси-серверы не обязательно находятся на том же сервере, что и ваше приложение, их нельзя использовать для перезапуска приложения в случае сбоя. Вместо этого для мониторинга приложений нужно использовать диспетчер процессов, например systemd. Если вы работаете с Docker, то обычно используете программное обеспечение для оркестрации контейнеров, такое как Kubernetes (<https://kubernetes.io>), для мониторинга работоспособности контейнеров.

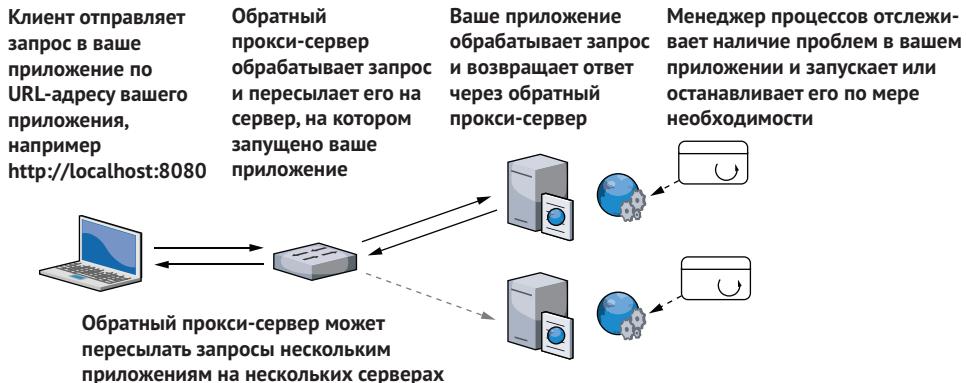


Рис. 27.9 В Linux обратный прокси-сервер обычно находится на сервере, отличном от вашего приложения. Обратный прокси-сервер перенаправляет входящие запросы в ваше приложение, в то время как диспетчер процессов, например systemd, отслеживает работу приложения на предмет сбоев и при необходимости перезапускает процесс

Запуск приложений ASP.NET Core в Docker

Docker – это наиболее часто используемый инструмент для контейнеризации приложений. Контейнер – это своего рода небольшая легковесная виртуальная машина для вашего приложения. Он содержит операционную систему, приложение и все его зависимости. Этот контейнер может быть запущен на любом компьютере с Docker, и ваше приложение будет работать точно так же, независимо от операционной системы хоста и того, что на ней установлено. Это обеспечивает высокую повторяемость развертываний: вы можете быть уверены, что если контейнер будет работать на вашей машине, он также будет работать и на сервере. ASP.NET Core хорошо подходит для развертывания контейнеров, но переход к Docker требует большого сдвига в методологии развертывания и может подойти или не подойти для вас и ваших приложений. Если вас интересуют возможности, предоставляемые Docker, и вы хотите узнать больше, предлагаю ознакомиться со следующими ресурсами:

- «*Docker in Practice*», 2-е изд., Иэна Миелла и Эйдана Хобсона Сэйерса (Manning, 2019) предлагает широкий спектр практических приемов, которые помогут вам получить максимальную отдачу от Docker (<http://mng.bz/nM8d>);
- даже если вы не выполняете развертывание в Linux, то можете использовать Docker для Windows. Ознакомьтесь с бесплатной электронной книгой «Введение в контейнеры Windows» Джона Маккеяба и Майкла Фрииса (Microsoft Press, 2017): <https://aka.ms/containersebook>;
- вы можете найти большое количество подробностей о создании и запуске приложений ASP.NET Core на Docker в документации .NET на странице <http://mng.bz/vz5a>;
- у Стива Гордона есть отличная серия постов в блоге о Docker для разработчиков ASP.NET Core: <https://www.stevejgordon.co.uk/docker-dotnetdevelopers>.

Конфигурирование этих систем – трудоемкая задача, требующая чтения скучных текстов, поэтому я не буду подробно описывать это здесь. Вместо этого я рекомендую ознакомиться с документацией по ASP.NET Core. Там есть руководство по NGINX и systemd (<http://mng.bz/yYGd>) и руководство по настройке Apache с помощью systemd (<http://mng.bz/MXVB>).

Оба руководства охватывают базовую конфигурацию соответствующих обратных прокси-серверов и супервизоров systemd, но, что более важно, они также показывают, как *безопасно* настроить их. Обратный прокси-сервер находится между вашим приложением и интернетом, поэтому важно все сделать правильно!

Настройка обратного прокси-сервера и диспетчера процессов – обычно самая сложная часть развертывания в Linux и не относится к разработке на платформе .NET: то же самое касается развертывания веб-приложения Node.js. Но нужно учесть несколько моментов *внутри* своего приложения, если вы собираетесь развертывать его в Linux. Об этом – в следующем разделе.

27.3.2 Подготовка приложения к развертыванию в Linux

В целом приложению все равно, за каким обратным прокси-сервером оно находится, будь то NGINX, Apache или IIS, – оно получает запросы и отвечает на них без какого-либо влияния на эти действия со стороны данного сервера. Когда вы используете IIS, нужно добавить метод `UseIISIntegration()`; аналогично, когда вы используете Linux, нужно добавить метод расширения в настройки своего приложения.

Когда запрос поступает на обратный прокси-сервер, он содержит некоторую информацию, которая теряется после того, как запрос перенаправляется в ваше приложение. Например, исходный запрос идет с IP-адреса клиента/браузера, подключающегося к вашему приложению: после пересылки запроса от обратного прокси-сервера IP-адрес – это уже адрес *обратного прокси*, а не *браузера*. Кроме того, если обратный прокси-сервер используется для SSL-терминирования (см. главу 28), то запрос, который изначально был сделан с использованием протокола HTTPS, может поступать в ваше приложение в виде HTTP-запроса.

Стандартным решением этих проблем является добавление обратным прокси-сервером дополнительных заголовков перед пересылкой запросов в приложение. Например, заголовок `X-Forwarded-For` идентифицирует IP-адрес исходного клиента, а заголовок `X-Forwarded-Proto` указывает на исходную схему запроса (`http` или `https`).

Чтобы ваше приложение работало правильно, оно должно искать эти заголовки во входящих запросах и при необходимости изменять запрос. Запрос адреса `http://localhost` с заголовком `X-Forwarded-Proto` со схемой `https` должен обрабатываться так же, как если бы это был запрос адреса `https://localhost`.

Чтобы добиться этого, можно использовать компонент `Forwarded-HeadersMiddleware` в конвейере промежуточного ПО. Он переопределяет `Request.Scheme` и другие свойства `HttpContext`, чтобы соответствовать перенаправленным заголовкам. Если вы используете метод по умолча-

нию `Host.CreateDefaultBuilder()` в файле `Program.cs`, считайте, что часть работы уже сделана – промежуточное ПО автоматически добавляется в конвейер в отключенном состоянии. Чтобы активировать его, установите для переменной окружения `ASPNETCORE_FORWARDEDHEADERS_ENABLED` значение `true`.

Если вы используете собственный экземпляр `HostBuilder` вместо построителя по умолчанию, то можете добавить промежуточное ПО в начало конвейера вручную, как показано в листинге 27.1, и сконфигурировать его, используя заголовки, которые нужно искать.

ВНИМАНИЕ Важно, чтобы компонент `ForwardedHeadersMiddleware` был размещён в начале конвейера для исправления `Request.Scheme` перед выполнением любого компонента, зависящего от схемы.

Листинг 27.1 Настройка приложения для использования перенаправленных заголовков в файле Startup.cs

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();           Добавляет компонент  
ForwardedHeadersMiddleware  
на раннем этапе конвейера
app.UseForwardedHeaders(new ForwardedHeadersOptions {           ←
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |           ←
        ForwardedHeaders.XForwardedProto
});           Настраивает  
заголовки, кото-  
рые промежуто-  
чное ПО должно  
искать и исполь-  
зовать
app.UseHttpsRedirection();
app.UseRouting();
app.MapGet("/", () => "Hello world!");
app.Run();
```

ForwardedHeadersMiddleware дол-
жен быть размещён перед всеми
остальными компонентами

ПРИМЕЧАНИЕ Такое поведение требуется для обратных прокси-серверов не только в Linux. Метод расширения `UseIis()` добавляет компонент `ForwardedHeadersMiddleware` под капотом как часть своей конфигурации, когда ваше приложение работает в IIS.

Помимо перенаправленных заголовков, необходимо учесть несколько незначительных моментов при развертывании приложения в Linux, которые могут сбить вас с толку, если вы привыкли к развертыванию в Windows:

- *окончание строк (LF в Linux и CRLF в Windows)* – Windows и Linux используют разные символы в тексте для обозначения конца строки. Часто это не является проблемой для приложений ASP.NET Core, но если вы пишете текстовые файлы на одной платформе, а читаете их на другой, то об этом нужно помнить;
- *разделитель каталогов пути («\» в Windows, «/» в Linux)* – это одна из самых распространенных ошибок, которую я вижу, когда

разработчики на Windows переходят на Linux. Каждая платформа использует свой разделитель в путях к файлам, поэтому если вы загружаете файл с использованием пути "subdir\myfile.json", в Windows это работает, но не в Linux. Вместо этого следует использовать метод `Path.Combine`, чтобы применить соответствующий разделитель для текущей платформы, например `Path.Combine("subdir", "myfile.json")`:

- *переменные окружения не могут содержать символ «:»* – в некоторых дистрибутивах Linux символ двоеточия (:) нельзя использовать в переменных окружения. Как было показано в главе 11, этот символ обычно используется для обозначения различных разделов в ASP.NET Core, поэтому вам часто нужно использовать его в переменных окружения. Вместо этого можно применять двойное подчеркивание (_), и ASP.NET Core будет воспринимать его так же, как если бы это было двоеточие;
- *данные о часовом поясе и данные локализации могут отсутствовать* – дистрибутивы Linux не всегда поставляются с данными о часовом поясе или данными локализации, что может приводить к проблемам с локализацией и вызывать исключения во время выполнения. Можно установить данные часового пояса с помощью диспетчера пакетов своего дистрибутива¹. Также данные могут быть организованы по-другому. Например, в Linux другая иерархия норвежских культур;
- *разная структура каталогов* – в дистрибутивах Linux используется совершенно иная структура папок, чем в Windows, поэтому нужно иметь это в виду, если ваше приложение жестко зашивает пути в код. В частности, рассмотрите различия в папках temporаги или cache.

Этот список ни в коем случае не является исчерпывающим, однако если вы настроили `ForwardedHeadersMiddleware` и позаботились об использовании кросс-платформенных конструкций, таких как `Path.Combine`, у вас не должно возникнуть много проблем с запуском приложений в Linux. Но настройка обратного прокси-сервера – не самое простое занятие, поэтому, где бы вы ни планировали разместить свое приложение, я предлагаю ознакомиться с документацией на странице <https://docs.microsoft.com/aspnet/core/publishing>.

27.4 Настройка URL-адресов приложения

На данный момент вы развернули приложение, но не сделали еще кое-что: не настроили URL-адреса для своего приложения. Когда вы используете IIS в качестве обратного прокси-сервера, вам не нужно беспокоиться об этом. Интеграция IIS с модулем ASP.NET Core работает, динамически создавая URL-адрес, который используется для пересылки запросов между IIS и вашим приложением. Имя хоста, которое вы настраиваете в IIS (на рис. 27.6), является тем URL-адресом, который

¹ Я сам столкнулся с этой проблемой. Подробнее о ней и о том, как я ее решил, можно прочитать в моем блоге: <http://mng.bz/aoem>

видят внешние пользователи; внутренний URL-адрес, который IIS использует при пересылке запросов, не виден.

Если вы не используете IIS в качестве обратного прокси-сервера – вероятно, вы используете NGINX или предоставляете доступ к своему приложению напрямую из сети Интернет, – возможно, вам понадобится настроить URL-адреса, по которым будет вестись прослушивание, вручную.

По умолчанию ASP.NET Core будет прослушивать запросы по URL-адресу `http://localhost:5000`. Есть много способов задать этот адрес, но в этом разделе я опишу два: с помощью переменных окружения или аргументов командной строки. Это два самых распространенных подхода (за пределами IIS) для управления URL-адресами, которые использует ваше приложение.

СОВЕТ Чтобы узнать о других способах установки URL-адреса, см. мой пост «5 способов задать URL-адрес для приложения ASP.NET Core»: <http://mng.bz/go0v>.

В главе 10 вы узнали о конфигурации в ASP.NET Core и, в частности, о концепции сред размещения, чтобы вы могли использовать разные настройки в зависимости от того, о чем идет речь: о запуске в окружении разработки или промышленном окружении. Среду размещения можно задать, установив на компьютере переменную окружения `ASPNETCORE_ENVIRONMENT`. ASP.NET Core волшебным образом подхватывает эту переменную, когда ваше приложение запускается, и использует ее, чтобы определить среду размещения.

Вы можете использовать аналогичную специальную переменную окружения, чтобы указать URL-адрес, который использует ваше приложение; это переменная `ASPNETCORE_URLS`. Когда ваше приложение запускается, оно ищет это значение и использует его в качестве URL-адреса приложения. Изменив его, можно изменить URL-адрес по умолчанию, используемый всеми приложениями ASP.NET Core на компьютере. Например, можно задать временную переменную окружения в Windows из командной оболочки:

```
set ASPNETCORE_URLS=http://localhost:8000
```

Запуск опубликованного приложения с использованием `dotnet <app.dll>` в том же командном окне, как видно на рис. 16.10, показывает, что приложение теперь ведет прослушивание по URL-адресу, предоставленному в переменной `ASPNETCORE_URLS`.

Вы можете указать приложению вести прослушивание по нескольким URL-адресам, разделив их точкой с запятой, или можете слушать определенный порт без указания имени хоста `localhost`. Если вы задаете переменную окружения `ASPNETCORE_URLS`:

```
http://localhost:5001;http://*:5002
```

то ваши приложения ASP.NET Core будут слушать запросы, отправленные на следующие адреса:

- `http://localhost:5001` – этот адрес доступен только на вашем локальном компьютере, поэтому он не будет принимать запросы из интернета;
- `http://*:5002` – любой URL-адрес на порту 5002. Внешние запросы из интернета могут получить доступ к приложению на порту 5002, используя любой URL-адрес, соответствующий вашему компьютеру.

```
C:\web\recipeapp>set ASPNETCORE_URLS=http://localhost:8000
C:\web\recipeapp>dotnet RecipeApplication.dll
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:8000
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:\web\recipeapp
```

Задайте переменную среды `ASPNETCORE_URLS`. Она применяется в течение всего срока жизненного цикла окна консоли

Ваше приложение прослушивает URL-адрес из переменной `ASPNETCORE_URLS`

Рис. 27.10 Измените переменную окружения `ASPNETCORE_URLS`, чтобы изменить URL-адрес, используемый приложениями ASP.NET Core

Обратите внимание, что вы *не можете* указать другое имя хоста, например `iciousrecipes.com`. ASP.NET Core будет слушать все запросы на определенном порту. Исключение составляет `localhost`, которое разрешает только запросы, поступающие с вашего компьютера.

ПРИМЕЧАНИЕ Если вы обнаружите, что переменная `ASPNETCORE_URLS` не работает должным образом, убедитесь, что у вас нет файла `launchSettings.json` в каталоге. Если он там присутствует, то приоритет получают значения из этого файла. По умолчанию файл `launchSettings.json` не включен в вывод `publish`, поэтому обычно в промышленном окружении это не будет проблемой.

Установка URL-адреса приложения с использованием одной переменной окружения отлично подходит для некоторых случаев, особенно когда вы запускаете одно приложение на виртуальной машине или в контейнере Docker.

Если вы не используете контейнеры Docker, то, скорее всего, размещаете несколько приложений рядом на одном компьютере. В этом случае одна переменная окружения не годится для того, чтобы задать URL-адрес, поскольку это приведет к изменению URL-адресов всех приложений.

В главе 11 было показано, что можно задать среду размещения с помощью переменной `ASPNETCORE_ENVIRONMENT`, но это также можно сделать, используя параметр `--environment` при вызове команды `dotnet run`:

```
dotnet run --no-launch-profile --environment Staging
```

СОВЕТ ASP.NET Core хорошо подходит для работы с контейнерами, но работа с контейнерами – это уже отдельная книга. До-

полнительные сведения о размещении и публикации приложений с помощью Docker см. на странице <http://mng.bz/e5GV>.

Можно задать URL-адреса для своего приложения аналогичным образом, применяя параметр `--urls`. Использование аргументов командной строки позволяет иметь несколько приложений ASP.NET Core, работающих на одной машине и слушающих разные порты. Например, следующая команда запустит приложение с рецептами, настроит его на прослушивание на порту 8081 и задаст Staging в качестве среды, как показано на рис. 27.11:

```
dotnet RecipeApplication.dll --urls "http://*:8081" --environment Staging
```

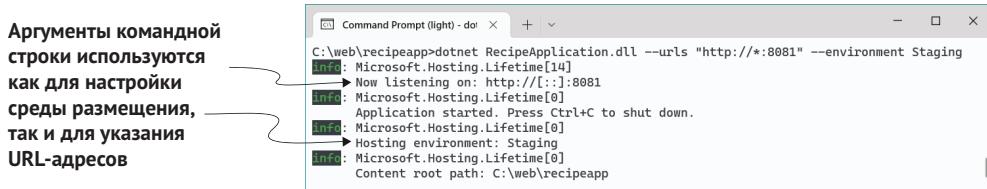


Рис. 27.11 Настройка окружения размещения и URL-адресов для приложения с помощью аргументов командной строки. Значения, переданные в командной строке, переопределяют значения, предоставленные из файла appSettings.json или из переменных окружения

Помните, что не нужно задавать URL-адреса таким образом, если вы используете IIS в качестве обратного прокси-сервера. Интеграция с IIS сделает это за вас. Задавать URL-адреса необходимо только в том случае, если вы вручную настраиваете URL-адрес, по которому ведет прослушивание ваше приложение; например, если вы используете NGINX или предоставляете доступ к Kestrel напрямую клиентам.

ВНИМАНИЕ! Если вы запускаете приложение ASP.NET Core без обратного прокси-сервера, то должны использовать фильтрацию хостов по соображениям безопасности, чтобы приложение отвечало только на запросы от ожидаемых вами имен хостов. Для получения более подробной информации см. мою запись в блоге: <http://mng.bz/pVXK>.

Мы подошли к концу главы о публикации приложения. Эта последняя миля в разработке приложения – развертывание приложения на сервере, где пользователи могут получить к нему доступ, – является общеизвестно сложной проблемой. Опубликовать приложение ASP.NET Core достаточно просто, но множество доступных вариантов хостинга затрудняет предоставление кратких шагов для каждой ситуации.

Какой бы вариант хостинга вы ни выбрали, есть одна важная тема, которую нельзя упускать из виду: безопасность. В следующей главе вы узнаете о HTTPS, о том, как его использовать при локальном тестировании и почему важно, чтобы все ваши рабочие приложения использовали HTTPS.

Резюме

- Приложения ASP.NET Core – это консольные приложения со встроенным веб-сервером. В промышленном окружении обычно используется обратный прокси-сервер, который обрабатывает исходный запрос и передает его вашему приложению. Обратные прокси-серверы могут обеспечить дополнительную безопасность, операции и преимущества в производительности, но также могут усложнить развертывание.
- Платформа .NET состоит из двух частей: .NET SDK (также известного как интерфейс командной строки .NET) и среды выполнения .NET. При разработке приложения вы используете интерфейс командной строки .NET для восстановления, сборки и запуска приложения. Visual Studio использует те же команды интерфейса командной строки .NET из интегрированной среды разработки.
- Если вы хотите развернуть приложение в промышленном окружении, необходимо опубликовать его, используя команду `dotnet publish`. Она создает папку, содержащую ваше приложение в виде файла с расширением .dll вместе со всеми его зависимостями.
- Чтобы запустить опубликованное приложение, вам не нужен интерфейс командной строки .NET, потому что вы не будете собирать приложение. Вам нужна только среда выполнения .NET для запуска опубликованного приложения. Вы можете запустить опубликованное приложение с помощью команды `dotnet app.dll`, где `app.dll` – это приложение в виде файла с расширением .dll, созданное командой `dotnet publish`.
- Приложения в ASP.NET Core IIS можно размещать в одном из двух режимов: внутривнепроцессный и внепроцессный. Во внепроцессном режиме ваше приложение запускается как отдельный процесс, типичный для большинства обратных прокси. Внутривнепроцессный режим запускает ваше приложение как часть процесса IIS. Это дает преимущества в производительности за счет исключения межпроцессного взаимодействия.
- Если вы используете пользовательский конструктор веб-приложений с IIS, убедитесь, что вы вызываете `UseIISIntegration()` и `UseIIS()`, чтобы IIS правильно пересыпал запрос вашему приложению. Если вы используете `WebApplicationBuilder` по умолчанию, эти методы вызываются автоматически.
- Когда вы публикуете свое приложение с помощью интерфейса командной строки .NET, файл `web.config` будет добавлен в папку вывода.
- Важно, чтобы этот файл был развернут при публикации вашего приложения в IIS, поскольку он определяет, как должно запускаться ваше приложение.
- URL-адрес, по которому будет выполнять прослушивание ваше приложение, указывается по умолчанию в переменной окружения `ASPNETCORE_URLS`. Установка этого значения изменит URL-адрес всех приложений на вашем компьютере. В качестве альтернативы можно передать аргумент командной строки `-urls` при запуске вашего приложения, например `dotnet app.dll --urls http://localhost:80`.



Добавляем протокол HTTPS в приложение

В этой главе:

- шифрование трафика между клиентами и приложением с помощью протокола HTTPS;
- использование сертификата HTTPS для локальной разработки;
- настройка Kestrel с использованием специального сертификата HTTPS;
- делаем, так, чтобы протокол HTTPS использовался для всего приложения.

Безопасность веб-приложений в настоящее время является горячей темой. Практически каждую неделю сообщается об очередном взломе или утечке конфиденциальных данных. Может показаться, что ситуация безнадежна, но реальность такова, что подавляющее большинство нарушений можно было предотвратить с минимальными усилиями.

В главе 29 мы рассмотрим ряд распространенных атак и способы защиты от них в приложении ASP.NET Core. В этой главе мы начнем с рассмотрения одной из самых основных мер безопасности: шифрования трафика между клиентом, например браузером, и приложением.

Без шифрования HTTPS вы рискуете, что третьи лица будут шпионить за запросами и ответами или изменять их во время их передачи через интернет. Риски, связанные с незашифрованным трафиком,

означают, что в наши дни HTTPS фактически является обязательным для производственных приложений и его активно поддерживают производители современных браузеров, таких как Chrome и Firefox. В разделе 28.1 вы узнаете больше об этих рисках и некоторых подходах, которые вы можете использовать для защиты своего приложения.

В разделе 28.2 вы увидите, как начать работу с HTTPS локально, используя сертификат разработки ASP.NET Core. Я описываю, что это такое, как доверять этому приложению и что делать, если оно не работает так, как вы ожидаете.

Сертификат разработки отлично подходит для локальной работы, но в рабочей среде вам потребуется настроить настоящий производственный сертификат. Я не описываю процесс получения сертификата в разделе 28.3, так как он зависит от провайдера; вместо этого я покажу, как настроить Kestrel для использования полученного вами пользовательского сертификата.

В разделе 28.4 я описываю некоторые подходы к обеспечению использования HTTPS в вашем приложении. К сожалению, веб-браузеры по-прежнему ожидают, что приложения по умолчанию будут доступны через HTTP, поэтому вам обычно необходимо предоставлять свое приложение как через порты HTTP, так и через HTTPS. Тем не менее есть вещи, которые вы можете сделать, чтобы направить клиентов к конечной точке HTTPS, что в наши дни считается передовой практикой безопасности.

Прежде чем мы рассмотрим HTTPS в ASP.NET Core, мы начнем с рассмотрения HTTPS в целом и почему вы должны использовать его во всех своих приложениях.

28.1 Для чего нужен протокол HTTPS?

В этом разделе вы узнаете о протоколе HTTPS: что это такое и почему нужно о нем знать, когда речь идет о реальных приложениях. Вы увидите два подхода, используемых для того, чтобы добавить его в свое приложение: прямая поддержка HTTPS в приложении и использование SSL/TLS на обратном прокси-сервере.

Затем вы узнаете, как использовать сертификат разработки для работы с HTTPS на локальной машине и как добавить сертификат HTTPS в приложение в промышленном окружении. Наконец, вы узнаете, как сделать так, чтобы протокол HTTPS использовался в приложении, применяя передовые практики, такие как заголовки безопасности и перенаправление с HTTP.

До сих пор в данной книге я показывал, как браузер пользователя отправляет запрос по сети в ваше приложение по протоколу HTTP. Мы не слишком вникали в подробности этого протокола, а только выяснили, что он использует методы для описания типа запроса (например, GET и POST), содержит заголовки с метаданными о запросе и, возможно, включает в себя полезную нагрузку с данными в теле запроса.

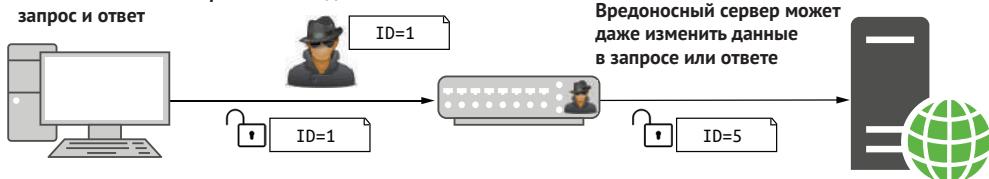
По умолчанию HTTP-запросы не шифруются; это простые текстовые файлы, которые персылаются по сети. Каждый, кто находится

в той же сети, что и пользователь (например, некто, использующий тот же общедоступный Wi-Fi в кафе), может читать запросы и ответы, которые пересылаются туда и обратно. Злоумышленники могут даже изменять запросы или ответы в процессе их передачи.

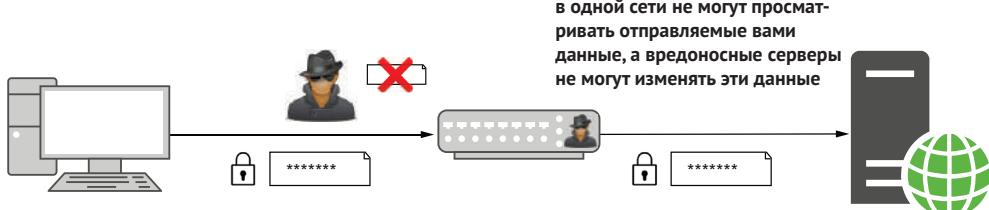
Такое использование незашифрованных веб-приложений представляет угрозу как для конфиденциальности, так и для безопасности ваших пользователей. Злоумышленники могут читать данные, отправляемые в формах и возвращаемые вашим приложением, внедрять вредоносный код в ответы, чтобы совершать атаки или похищать cookie-файлы аутентификации, выдавая себя за пользователей вашего приложения.

Чтобы защитить пользователей, нужно шифровать трафик между браузером пользователя и вашим приложением, когда он передается по сети с использованием протокола HTTPS. Это похоже на HTTP-трафик, но здесь используется сертификат SSL/TLS для шифрования запросов и ответов, поэтому злоумышленники не могут прочитать или изменить содержимое.

При использовании протокола HTTP данные в запросе остаются незашифрованными. Это означает, что любой пользователь в той же сети может прочитать каждый запрос и ответ



Вредоносный сервер может даже изменить данные в запросе или ответе



С помощью HTTPS пользователи в одной сети не могут просматривать отправляемые вами данные, а вредоносные серверы не могут изменять эти данные

Рис. 28.1 Защищенные приложения, использующие протокол HTTPS, и незащищенные приложения, использующие протокол HTTP в браузере Edge. Применяя HTTPS, вы защищаете свое приложение от просмотра или взлома злоумышленниками

ОПРЕДЕЛЕНИЕ Secure Sockets Layer (SSL) – это более старый стандарт, который упрощает HTTPS. Протокол SSL был заменен протоколом Transport Layer Security (TLS), поэтому в этой главе я буду отдавать предпочтение TLS. Обычно если вы слышите, как кто-то говорит об SSL или сертификатах SSL, на самом деле они имеют в виду TLS. Вы можете найти RFC для последней версии протокола TLS на странице <https://www.rfc-editor.org/rfc/rfc8446>.

В браузерах вы можете определить, что сайт использует HTTPS, по префиксу `https://` к URL-адресам (обратите внимание на `s`), а иногда

и по замку, как показано на рис. 28.2. Большинство современных браузеров в наши дни преуменьшают значение того, что сайт использует HTTPS, поскольку большинство сайтов используют HTTPS, и вместо этого подчеркивают, когда вы находитесь на сайте, который не использует HTTPS, помечая его как небезопасный.

Реальность такова, что в наши дни вы всегда должны использовать HTTPS на своих рабочих веб-сайтах. По умолчанию индустрия продвигает HTTPS, при этом большинство браузеров помечают HTTP-сайты как явно небезопасные. Отказ от использования HTTPS в долгосрочной перспективе будет вызывать недоверие у потенциальных пользователей вашего приложения, поэтому, даже если вас не интересуют преимущества безопасности, в ваших интересах настроить HTTPS.

СОВЕТ Вы можете найти хорошую шпаргалку по HTTPS от OWASP по адресу <http://mng.bz/PzxY>. И ASP.NET Core по умолчанию решает проблемы из этого списка, но есть некоторые важные моменты, в частности в разделе «Приложение».

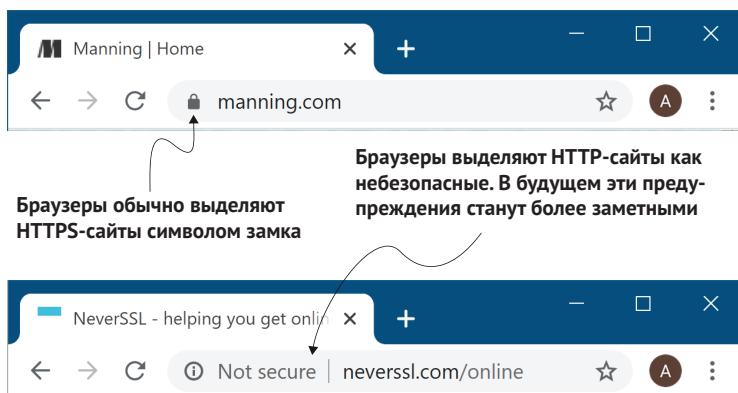


Рис. 28.2 Зашифрованные приложения, использующие протокол HTTPS, и незашифрованные приложения, использующие протокол HTTP в браузере Edge. Применяя HTTPS, вы защищаете свое приложение от просмотра или взлома злоумышленниками

Еще одна причина поддержки HTTPS заключается в том, что многие функции браузера доступны только тогда, когда ваш сайт обрабатывается через HTTPS. Некоторые из этих функций представляют собой API-интерфейсы браузера JavaScript, такие как API-интерфейсы определения местоположения, API микрофона и API хранилища. Они доступны только через HTTPS для защиты пользователей от злоумышленников, которые могут изменить небезопасные HTTP-запросы. Другие функции также применимы к серверным приложениям, например сжатие Brotli и поддержка HTTP/2.

СОВЕТ Подробную информацию о том, как работают протоколы SSL/TLS, см. в главе 9 книги Дэвида Вонга (Manning, 2021), <http://mng.bz/zxz1>.

Чтобы активировать HTTPS, необходимо получить и настроить сертификат TLS для своего сервера. К сожалению, хотя этот процесс намного проще, чем раньше, и сделать это теперь можно практически бесплатно благодаря Let's Encrypt (<https://letsencrypt.org/>), во многих случаях все еще далеко не так просто. Если вы настраиваете рабочий сервер, то рекомендую внимательно следовать руководству на сайте Let's Encrypt. Ошибиться легко, поэтому не торопитесь.

СОВЕТ Если вы размещаете свое приложение в облаке, большинство провайдеров предоставляют сертификаты TLS по одному клику, чтобы вам не нужно было самостоятельно управлять сертификатами. Это чрезвычайно полезно, и я настоятельно рекомендую всем делать так. Чтобы воспользоваться этим, вам даже не обязательно размещать свое приложение в облаке. Cloudflare (<https://www.cloudflare.com>) предоставляет службу CDN, к которой вы можете добавить TLS. Вы даже можете использовать ее бесплатно.

Будучи разработчиками приложений ASP.NET Core, вы часто можете обойтись без *прямой* поддержки HTTPS в приложении, используя архитектуру обратного прокси-сервера, как показано на рис. 28.3, в процессе, называемом SSL/TLS-разгрузкой, или SSL/TLS-терминацией. Вместо того чтобы обрабатывать запросы напрямую, используя HTTPS, ваше приложение продолжает использовать HTTP. Обратный прокси-сервер отвечает за шифрование и дешифровку HTTPS-трафика от браузера. Часто это дает вам лучшее из обоих миров – данные шифруются между браузером пользователя и сервером, но вам не нужно беспокоиться о настройке сертификатов.

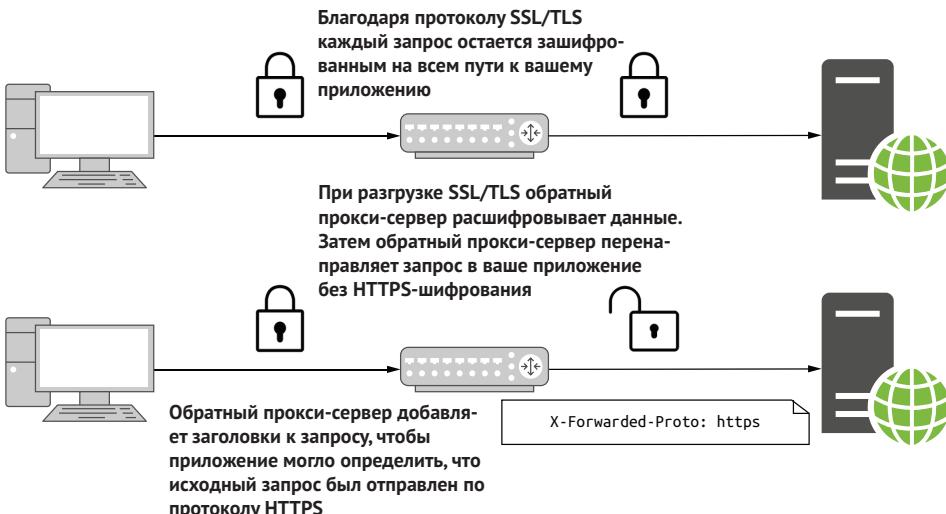


Рис. 28.3 При использовании HTTPS с обратным прокси-сервером у вас есть два варианта: сквозная передача SSL/TLS и разгрузка SSL/TLS. При сквозной передаче SSL/TLS данные шифруются на всем пути до вашего ASP.NET Core приложения. При разгрузке SSL/TLS обратный прокси-сервер или балансировщик выполняет расшифровку данных, поэтому вашему приложению этого делать не нужно

ПРИМЕЧАНИЕ Если вы обеспокоены тем, что трафик между обратными прокси и вашим приложением не зашифрован, рекомендую прочитать пост Троя Ханта «CloudFlare, SSL и не-здоровый абсолютизм безопасности»: <http://mng.bz/eHCl>. В нем обсуждаются плюсы и минусы проблемы, связанные с расшифровкой на обратном прокси-сервере, и почему вы должны учитывать наиболее вероятные атаки на ваш веб-сайт в процессе, называемом моделированием угроз.

В зависимости от конкретной инфраструктуры, в которой вы размещаете свое приложение, обработку SSL/TLS можно перенести на выделенное устройство в вашей сети, сторонний сервис, например Cloudflare, или обратный прокси-сервер (например, IIS, NGINX или HAProxy), работающие на том же или другом сервере. Тем не менее в некоторых ситуациях вам может потребоваться работать с SSL/TLS напрямую в своем приложении:

- если вы предоставляете доступ к Kestrel из сети Интернет напрямую, без обратного прокси-сервера. Подобное стало чаще встречаться с ASP.NET Core 3.0 из-за усиления защиты сервера Kestrel. Такое тоже не редко бывает, когда вы разрабатываете свое приложение локально;
- если наличие протокола HTTP между обратным прокси-сервером и вашим приложением неприемлемо. Когда речь идет о защите трафика внутри вашей сети, это менее критично по сравнению с внешним трафиком, однако, несомненно, безопаснее использовать HTTPS и для внутреннего трафика;
- если вы используете технологию, требующую HTTPS. Некоторые новые сетевые протоколы, такие как gRPC и HTTP/2, требуют соединения по протоколу HTTPS.

В каждом из этих сценариев вам потребуется настроить сертификат TLS для своего приложения, чтобы Kestrel мог получать HTTPS-трафик. В разделе 28.2 вы увидите самый простой способ приступить к использованию протокола HTTPS при локальной разработке, с применением сертификата разработки ASP.NET Core.

28.2 Использование HTTPS-сертификатов для разработки

Работать с сертификатами HTTPS стало проще, чем раньше, но, к сожалению, некоторые вещи по-прежнему могут сбивать с толку, особенно если вы новичок в веб-разработке. Набор средств разработки .NET, Visual Studio и IIS Express пытаются улучшить этот опыт, выполняя значительную часть неблагодарной работы за вас.

При первом запуске команды `dotnet` с помощью набора средств разработки .NET этот набор устанавливает HTTPS-сертификат для разработки на ваш компьютер. Любое приложение ASP.NET Core, которое вы создаете с использованием шаблонов по умолчанию (или для которых вы не настраиваете сертификаты явно), будет использовать этот

сертификат для обработки HTTPS-трафика. Однако сертификат для разработки по умолчанию не является доверенным. Если вы заходите на сайт, использующий ненадежный сертификат, вы получите предупреждение браузера, как показано на рис. 28.4.

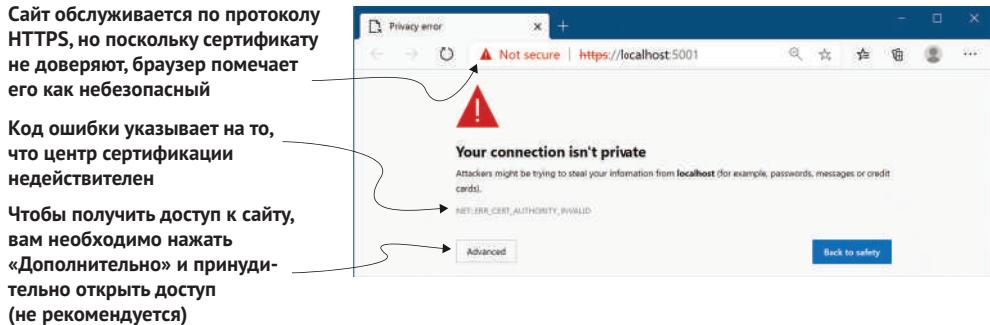


Рис. 28.4 Сертификат разработчика не является доверенным по умолчанию, поэтому приложения, обслуживающие HTTPS-трафик и использующие его, будут помечены браузерами как небезопасные. Хотя при необходимости вы можете обойти эти оповещения, нужно обновить сертификат, чтобы он стал доверенным

Краткое руководство по сертификатам и подписи

HTTPS использует криптографическую систему с открытым ключом как часть процесса шифрования данных. Здесь задействованы два ключа: открытый ключ, который может видеть любой, и закрытый ключ, который видит только ваш сервер. Все, что зашифровано с помощью открытого ключа, можно расшифровать только с помощью закрытого ключа. Таким образом, браузер может что-то зашифровать с помощью открытого ключа вашего сервера, и только ваш сервер может его расшифровать. Полный сертификат TLS состоит из открытых и закрытых частей.

Когда браузер подключается к вашему приложению, сервер отправляет часть с открытым ключом сертификата TLS. Но откуда браузер знает, что именно ваш сервер отправил сертификат? Для этого ваш сертификат TLS содержит дополнительные сертификаты, включая сертификат от третьей стороны, центра сертификации (CA). Этот доверенный сертификат называется корневым сертификатом.

Центр сертификации – это сторона, чья честность неоспорима, а браузеры жестко сконфигурированы, чтобы доверять определенным корневым сертификатам. Чтобы сертификат TLS для вашего приложения был доверенным, он должен содержать (или быть подписан) доверенный корневой сертификат.

Когда вы используете сертификат разработки ASP.NET Core или создаете собственный самоподписанный сертификат, у HTTPS вашего сайта нет этого доверенного корневого сертификата. Это означает, что браузеры не будут доверять вашему сертификату и не будут подключаться к вашему серверу по умолчанию. Чтобы обойти это, нужно дать явное указание своей машине, используемой для разработки, что нужно доверять сертификату.

В промышленном окружении нельзя использовать сертификат разработки или самоподписанный сертификат, поскольку браузер пользователя ему не доверяет. Вместо этого нужно получить подписанный сертификат HTTPS от такого сервиса, как Let's Encrypt, или от облачного провайдера, такого как AWS, Azure или Cloudflare. Эти сертификаты уже будут подписаны доверенным центром сертификации, поэтому браузеры будут автоматически доверять им.

Чтобы устраниТЬ эти оповещения со стороны браузера, необходимо подтвердить доверие такому сертификату. Подтверждение доверия сертификату – это некое сообщение, что-то типа «Я знаю, что этот сертификат выглядит не совсем правильно, но просто проигнорируй это», – поэтому сложно сделать это автоматически. Если вы работаете в Windows или macOS, то можете изменить настройки для сертификата разработки, выполнив команду

```
dotnet dev-certs https --trust
```

Эта команда подтверждает доверие сертификату, регистрируя его в «хранилище сертификатов» операционной системы. После того как вы выполните данную команду, вы сможете получить доступ к своим веб-сайтам без каких-либо предупреждений или ярлыков «небезопасно», как показано на рис. 28.5.

Теперь сертификат является доверенным, поэтому он имеет символ замка, больше не помечен как небезопасный и не отображается красным цветом

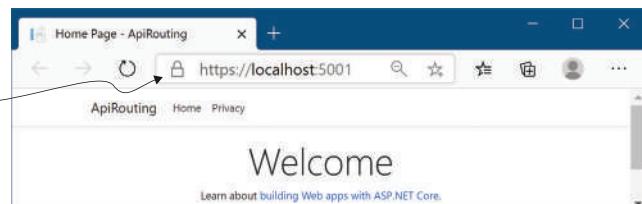


Рис. 28.5 После того как сертификат разработки станет доверенным, вы больше не увидите предупреждения браузера о подключении

СОВЕТ Возможно, вам придется закрыть браузер после того, как вы подтвердили доверие сертификату, чтобы очистить кеш браузера.

Если вы используете Windows, Visual Studio и IIS Express для разработки, то, возможно, вам не потребуется эта процедура подтверждения доверия сертификату разработки. Дело в том, что IIS Express действует как обратный прокси-сервер, когда вы осуществляете разработку локально, поэтому он сам выполняет настройку SSL/TLS. Кроме того, Visual Studio должна подтвердить доверие сертификату разработки IIS во время установки, поэтому вы можете так и не увидеть предупреждений со стороны браузера.

СОВЕТ В macOS до .NET 7 приходилось повторно подтверждать доверие сертификату разработчика для каждого нового приложения. В .NET 7 этот процесс стал намного более бесшовным, поэтому вам не придется так часто подтверждать доверие!

Сертификат разработчика без проблем работает в Windows и macOS. К сожалению, процедура подтверждения доверия сертификату в Linux немного сложнее, и все зависит от конкретного варианта, который вы используете. Кроме того, программное обеспечение в Linux часто использует собственное хранилище сертификатов, поэтому вам, вероятно, потребуется добавить сертификат непосредственно в свой браузер. Если вы используете любой из следующих сценариев, вам придется проделать дополнительную работу:

- браузер Firefox в Windows, macOS или Linux;
- браузеры Edge или Chrome в Linux;
- связь API-API в Linux;
- приложение, работающее в подсистеме Windows для Linux (WSL);
- запуск приложений в Docker.

Каждый из этих сценариев требует немного разного подхода. Во многих случаях это одна или две команды, поэтому я предлагаю внимательно следовать документации для вашего сценария по адресу <http://mng.bz/JgIK>.

СОВЕТ Если вы попытались подтвердить доверие сертификату, но ваше приложение по-прежнему выдает ошибки, попробуйте закрыть все окна браузера и запустить `dotnet dev-certs https --clean`, а затем `dotnet dev-certs https --trust`. Браузеры кешируют подтверждения доверия сертификатам, поэтому этап закрытия и открытия важен!

Сертификаты разработки ASP.NET Core и IIS упрощают использование Kestrel и HTTPS локально, но эти сертификаты не помогут, когда вы перейдете в промышленное окружение. В следующем разделе я покажу вам, как настроить Kestrel для использования сертификата TLS для промышленного окружения.

28.3 Настройка Kestrel для использования сертификата HTTPS в промышленном окружении

Создание сертификата TLS для промышленного окружения часто является трудоемким процессом, поскольку требует подтвердить стороннему центру сертификации (ЦС), что вы являетесь владельцем домена, для которого создаете сертификат. Это важный шаг в процессе подтверждения «доверия», который гарантирует, что злоумышленники не смогут выдавать себя за ваши серверы. Результатом данного процесса является один или несколько файлов, представляющих собой сертификат HTTPS, который необходимо настроить для своего приложения.

СОВЕТ Особенности получения сертификата зависят от провайдера и вашей платформы ОС, поэтому внимательно читайте документацию от своего провайдера. Перипетии и сложности этого процесса – одна из причин, по которой я предпочитаю использование функции завершения SSL/TLS (SSL/TLS-termination), или

подхода «в один клик», описанного ранее. Это означает, что моим приложениям не нужно иметь дело с сертификатами, а мне не нужно использовать подходы, описанные в этом разделе; я делегирую эту ответственность другой части сети или базовой платформе.

Получив сертификат, необходимо настроить Kestrel, чтобы использовать его для обслуживания HTTPS-трафика. В главе 27 вы узнали, как настроить порт, на котором будет выполнять прослушивание ваше приложение, с помощью переменной окружения `ASPNETCORE_URLS` или через командную строку, и вы видели, что можно предоставить URL-адрес с префиксом HTTPS. Поскольку вы не указали конфигурацию сертификата, по умолчанию Kestrel использует сертификат разработки. В промышленном окружении необходимо указать Kestrel, какой сертификат использовать.

Kestrel очень легко настраивать, что позволяет настраивать сертификаты несколькими способами. Вы можете использовать разные сертификаты для разных портов, можете загружать из файла с расширением .pfx или из хранилища сертификатов ОС, или у вас может быть своя конфигурация для каждой конечной точки URL-адреса, которую вы предоставляете. Для получения полной информации см. раздел «Конфигурация конечной точки» в документации Microsoft «Реализация веб-сервера Kestrel в ASP.NET Core»: <http://mng.bz/KMdX>.

В следующем листинге показан один из возможных способов установки специального сертификата HTTPS для приложения в промышленном окружении, сертификат настроен по умолчанию, его Kestrel использует для HTTPS-соединений. Вы можете добавить раздел `"Kestrel:Certificates:Default"` в свой файл `appsettings.json` (или используя любой другой источник конфигурации, как описано в главе 10), чтобы определить файл сертификата с расширением .pfx, который будет использоваться. Вы также должны указать пароль для доступа к сертификату.

Листинг 28.1 Настройка сертификата HTTPS по умолчанию для Kestrel с использованием файла .pfx

```
{
  "Kestrel": {
    "Certificates": {
      "Default": {
        "Path": "localhost.pfx",           | Создайте раздел
        "Password": "testpassword"       | конфигурации
                                            | в Kestrel:Certificates:Default
                                            | Относительный или абсолют-
                                            | ный путь к сертификату
      }
    }
  }
}
```

Пароль для открытия сертифи-ката

Предыдущий пример – самый простой способ заменить сертификат HTTPS, поскольку он не требует изменения каких-либо значений по умолчанию для Kestrel. Вы можете использовать аналогичный подход, чтобы загрузить сертификат HTTPS из хранилища сертификатов

ОС (в Windows или macOS), как показано в документации «Конфигурация конечной точки», упомянутой ранее.

ВНИМАНИЕ В листинге 28.1 имя файла сертификата и пароль указаны явно, только для учебных целей, но вы должны загружать их из хранилища конфигурации, например User Secrets, как было показано в главе 10, или загрузить сертификат из локального хранилища. *Никогда не помещайте рабочие пароли в файлы appsettings.json.*

Все шаблоны ASP.NET Core по умолчанию настраивают ваше приложение на обслуживание HTTP- и HTTPS-трафика, и, используя конфигурацию, которую вы видели до сих пор, вы можете быть уверены, что ваше приложение может работать как с HTTP, так и HTTPS и в окружении разработки, и в промышленном окружении.

Однако используете ли вы HTTP или HTTPS, может зависеть от URL-адреса, по которому пользователи щелкают мышью, когда впервые заходят в ваше приложение. Например, если ваше приложение выполняет прослушивание, используя URL-адреса по умолчанию, `http://localhost:5000` для HTTP-трафика и `https://localhost:5001` для HTTPS-трафика, то когда пользователь переходит по URL-адресу с префиксом HTTP, его трафик шифроваться не будет. И если вы уже настроили HTTPS, то, вероятно, лучше всего заставить пользователей использовать только его.

28.4 Делаем так, чтобы протокол HTTPS использовался для всего приложения

Для современных веб-приложений использование протокола HTTPS является обязательным. Браузеры начинают явно помечать HTTP-страницы как небезопасные; и по соображениям безопасности *нужно* использовать TLS каждый раз, когда передаете конфиденциальные данные по сети. Кроме того, благодаря протоколу HTTP/2 добавление TLS может *улучшить* производительность вашего приложения. В этом разделе вы познакомитесь с тремя методами обеспечения соблюдения HTTPS в вашем приложении.

СОВЕТ HTTP/2 предлагает множество улучшений производительности по сравнению с HTTP/1.x, и всем современным браузерам для его включения требуется HTTPS. Подробное введение в HTTP/2 можно найти в статье Google «Введение в HTTP/2» по адресу <http://mng.bz/9M8j>. ASP.NET Core даже включает поддержку HTTP/3, следующей версии протокола! Вы можете прочитать о HTTP/3 по адресу <http://mng.bz/qrrJ>.

Существует несколько подходов сделать так, чтобы протокол HTTPS использовался для всего приложения. Если вы используете обратный прокси-сервер с функцией завершения SSL/TLS, то все будет сделано за вас, и вам не нужно беспокоиться об этом в своих прило-

жениях. В этом случае вы можете проигнорировать некоторые шаги, описанные в данном разделе.

ВНИМАНИЕ Если вы создаете веб-API, а не приложение Razor Pages, то в API полностью отклоняются небезопасные HTTP-запросы. Вы увидите этот подход в разделе 28.4.3.

Один из подходов к повышению безопасности приложения – использовать заголовки безопасности HTTP. Это HTTP-заголовки, отправляемые как часть вашего HTTP-ответа, которые сообщают браузеру, как он должен себя вести. Есть множество разных заголовков, большинство из которых ограничивают функциональность вашего приложения в обмен на повышенную безопасность. В следующей главе вы увидите, как добавлять собственные заголовки к HTTP-ответам, создавая специальные компоненты промежуточного ПО.

СОВЕТ У Скотта Хельма есть отличные рекомендации по этому и другим заголовкам безопасности, которые вы можете добавить на свой сайт, например по заголовку Content Security Policy (CSP). См. «Безопасность заголовков HTTP-ответов» на его веб-сайте по адресу: <http://mng.bz/7DDe>.

Один из этих заголовков безопасности, заголовок HTTP Strict Transport Security (HSTS), может помочь убедиться, что браузеры используют протокол HTTPS там, где он доступен, вместо использования по умолчанию протокола HTTP.

28.4.1 Принудительное использование протокола HTTPS с помощью заголовков HTTP Strict Transport Security

К сожалению, по умолчанию браузеры всегда загружают приложения по протоколу HTTP, если не указано иное. Это означает, что ваши приложения обычно должны поддерживать и HTTP, и HTTPS, даже если вы не хотите обслуживать трафик через HTTP, как показано на рис. 28.6. Кроме того, если первоначальный запрос выполняется по HTTP, браузер может в конечном итоге отправлять последующие запросы также по HTTP.

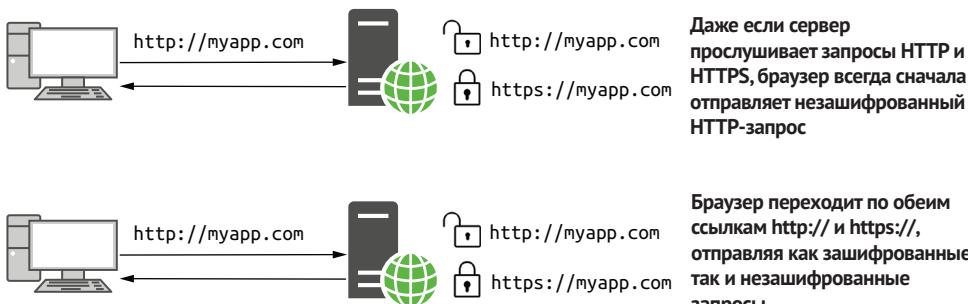


Рис. 28.6 Когда вы вводите URL-адрес, браузеры по умолчанию загружают приложение через HTTP. В зависимости от ссылок, возвращаемых вашим приложением, или введенных URL-адресов браузер может выполнять запросы HTTP или HTTPS

Одной из мер (и лучшей практикой в области безопасности) является добавление заголовков HTTP Strict Transport Security в ваши ответы.

ОПРЕДЕЛЕНИЕ HTTP Strict Transport Security (HSTS) – это спецификация (<https://www.rfc-editor.org/rfc/rfc6797>) для заголовка Strict-Transport-Security, который указывает браузеру использовать HTTPS для всех последующих запросов к вашему приложению. Заголовок HSTS можно отправлять только вместе с ответами на HTTPS-запросы. Это также актуально только для запросов, исходящих из браузера; это не влияет на связь между серверами или на мобильные приложения.

После того как браузер получает валидный заголовок HSTS, он перестает отправлять HTTPS-запросы вашему приложению и вместо этого использует только HTTPS, как показано на рис. 28.7. Даже если в вашем приложении есть ссылка `http://` или пользователь вводит `http://` в строке URL-адреса приложения, браузер автоматически заменяет запрос версией `https://`.

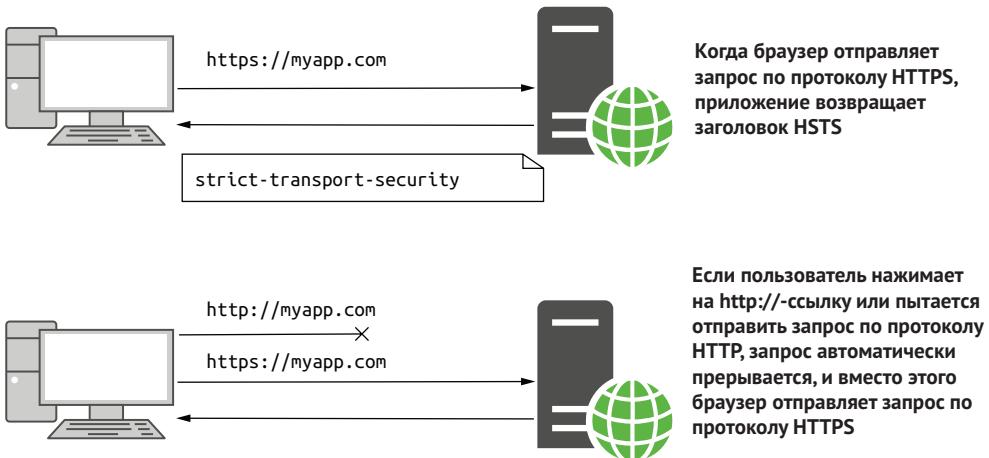


Рис. 28.7 После того как браузер отправляет запрос HTTPS, приложение возвращает заголовок HSTS, указывая браузеру всегда отправлять запросы через HTTPS. В следующий раз, когда пользователь попытается выполнить запрос `http://`, браузер прерывает запрос и вместо этого выполняет запрос `https://`

СОВЕТ Вы можете добиться аналогичного обновления запросов HTTP на HTTPS, используя директиву Upgrade-Insecure-Requests в заголовке Content-Security-Policy (CSP). Она обеспечивает меньшую защиту, чем заголовок HSTS, но может использоваться в сочетании с ним. Более подробную информацию об этой директиве и CSP в целом см. по адресу <http://mng.bz/mWV4>.

Заголовки HSTS настоятельно рекомендуются для приложений в промышленном окружении. Вы вряд ли будете активировать их для локальной разработки, поскольку это будет означать, что вы никогда не сможете запустить приложение, не использующее протокол HTTPS, локально.

Аналогичным образом вы должны использовать HSTS только на сайтах, для которых вы всегда собираетесь применять HTTPS, поскольку непросто (иногда невозможно) отключить его после того, как он был «включен» с помощью HSTS.

ASP.NET Core поставляется со встроенным промежуточным ПО для установки заголовков HSTS, которое автоматически включается в некоторые шаблоны по умолчанию. В следующем листинге показано, как настроить заголовки HSTS для своего приложения, используя `HstsMiddleware` в файле `Program.cs`.

Листинг 28.2 Использование компонента `HstsMiddleware` для добавления заголовков HSTS в приложение

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddHsts(options =>
{
    options.MaxAge = TimeSpan.FromHours(1);
});

WebApplication app = builder.Build();

if(app.Environment.IsProduction()) <-- Этот код изменяет параметры
{                                         настроек в заголовках послед-
    app.UseHsts();                      дующих запросов с использова-
}                                        нием HSTS и изменяет MaxAge
                                         с 30 дней на один час

                                         Не следует использовать HSTS на
                                         локальной машине разработчика
                                         Добавляет
                                         HstsMiddleware

app.UseStaticFiles();
app.UseRouting();

app.MapRazorPages();

app.Run();
```

В этом примере показано, как изменить значение `MaxAge`, отправленное в заголовке HSTS. Лучше всего начать с небольшого значения. И только когда вы убедитесь, что HTTPS в вашем приложении работает правильно, увеличьте значение, чтобы повысить безопасность. Общепринятое значение для развертывания в промышленном окружении – один год.

ВНИМАНИЕ Как только клиентские браузеры получили заголовок HSTS, браузеры по умолчанию будут использовать HTTPS для всех запросов к вашему приложению. Это означает, что вы должны всегда использовать HTTPS, пока не истечет срок, равный значению в `MaxAge`. Но если по каким-то причинам протокол HTTPS будет отключен, то при таких настройках браузеры не будут автоматически изменять запросы на протокол HTTP до истечения срока, указанного в `MaxAge`, поэтому ваше приложение может быть недоступно в этот период до тех пор, пока снова не будет доступен HTTPS! Поэтому для таких ситуаций необходимо изменять настройки HSTS, установив для `MaxAge` значение 0.

Одним из ограничений заголовка HSTS является то, что вы должны сделать первоначальный запрос через HTTPS, прежде чем сможете получить заголовок. Если браузер отправляет только HTTP-запросы, у приложения никогда не будет возможности отправить заголовок HSTS, поэтому браузер никогда не узнает, что нужно использовать HTTPS. Одним из возможных решений такой проблемы является предварительная загрузка HSTS.

Предварительная загрузка HSTS не является частью спецификации HSTS, но поддерживается всеми современными браузерами. Предварительная загрузка добавляет заголовок HSTS в браузер, чтобы браузер знал, что к вашему сайту следует отправлять только HTTPS-запросы. Это решает проблему недоступности сайта при первом запросе по HTTP, но имейте в виду, что предварительная загрузка HSTS исключает использование HTTP для запросов браузеров к вашему серверу и требует установки защищенного соединения по HTTPS.

Настроив использование HTTPS для вашего приложения, далее можно подготовить его к предварительной загрузке HSTS, настроив заголовок с HSTS, в котором можно:

- установить максимальное время использования HSTS не менее одного года, хотя рекомендуется два года;
- включить директиву includeSubDomains;
- включить директиву предварительной загрузки.

В листинге 28.3 показано, как можно настроить эти директивы в приложении. В листинге также показано, как исключить домен типа never-https.com, чтобы, если вы размещаете свое приложение в этом домене, заголовки HSTS не отправлялись. Это может быть полезно для целей тестирования.

Листинг 28.3 Настройка HSTS-заголовка приложения для предварительной загрузки

```
builder.Services.AddHsts(options =>
{
    options.Prefetch = true;
    options.IncludeSubDomains = true; ← Инициализация директивы
    options.MaxAge = TimeSpan.FromDays(365); ← includeSubDomains
    options.ExcludedHosts.Add("never-https.com"); ← Вы должны использовать HSTS-заголовки
}); ← минимум один год
                                                ← Не использовать заголовки
                                                ← HSTS для этого домена
```

Инициализация директивы для предварительной загрузки

Подготовив приложение для предварительной загрузки HSTS, вы можете отправить его для включения в список предварительной загрузки HSTS, который поставляется с современными браузерами. Посетите сайт <https://hstspreload.org>, подтвердите, что ваше приложение соответствует требованиям, и отправьте ссылку на свой домен для проверки. Если все правильно, то ваш домен будет включен в будущие версии всех современных браузеров!

СОВЕТ Более подробную информацию о HSTS и атаках, которые он может предотвратить, см. в статье Скотта Хельма «HSTS – недостающее звено в безопасности транспортного уровня» по адресу <http://mng.bz/5wwa>.

HSTS – отличный вариант, чтобы заставить пользователей использовать HTTPS на вашем сайте. Но есть одна проблема с заголовком, которая заключается в том, что его можно добавлять только в HTTPS-запросы. Это значит, что вы уже должны сделать HTTPS-запрос до того, как HSTS сработает: и если *первоначальный* запрос – это запрос по протоколу HTTP, заголовок HSTS не отправится, и далее будет использоваться HTTP! Это не очень хорошая ситуация, однако вы можете решить проблему, перенаправляя все небезопасные запросы на HTTPS.

28.4.2 Переадресация с HTTP на HTTPS с помощью компонента `HttpsRedirectionMiddleware`

`HstsMiddleware` обычно следует использовать вместе с промежуточным ПО, которое перенаправляет все HTTP-запросы на HTTPS.

СОВЕТ Перенаправление на HTTPS можно применять только к отдельным частям приложения, например к определенным страницам Razor Pages, но я не рекомендую этого делать, поскольку так можно очень легко создать брешь в системе безопасности своего приложения.

ASP.NET Core поставляется с компонентом `HttpsRedirection-Middleware`, который можно использовать для принудительного использования HTTPS во всем приложении. Он добавляется в конвейер промежуточного ПО в файле `Program.cs` и гарантирует, что все проходящие через него запросы безопасны. Если HTTP-запрос доходит до `HttpsRedirectionMiddleware`, то промежуточное ПО тотчас же прервет обработку запроса в конвейере, выполняя перенаправление запроса на HTTPS. Далее браузер повторит запрос, используя вместо HTTP протокол HTTPS.

ПРИМЕЧАНИЕ Даже при использовании заголовков HSTS и `HttpsRedirectionMiddleware` проблема все еще присутствует. По умолчанию браузеры всегда будут делать начальный небезопасный запрос к вашему приложению по протоколу HTTP. Единственный способ избежать этого – предварительно загрузить заголовки HSTS, сообщив браузерам, что для этого сайта всегда нужно использовать HTTPS.

`HttpsRedirectionMiddleware` добавляется в стандартные шаблоны ASP.NET Core. Обычно он помещается после обработки ошибок и `HstsMiddleware`, как показано в следующем листинге. По умолчанию промежуточное ПО перенаправляет все HTTP-запросы на безопасную конечную точку, используя код состояния 307 Temporary Redirect.

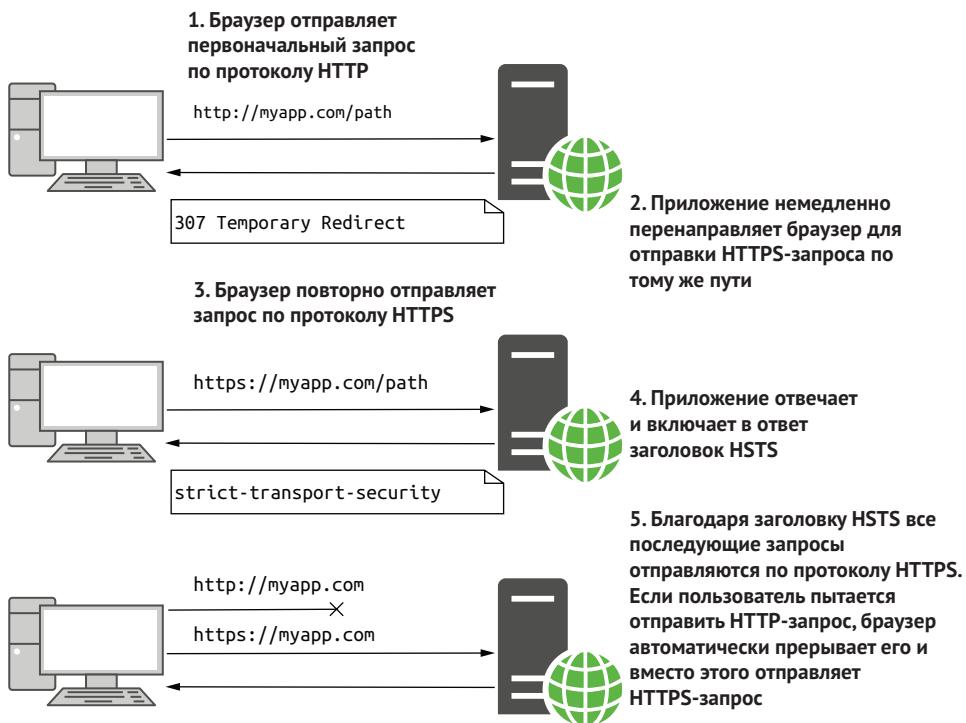


Рис. 28.8 HttpsRedirectionMiddleware работает с HstsMiddleware, чтобы гарантировать, что все запросы после первоначального запроса всегда отправляются через HTTPS

Листинг 28.4 Использование компонента HttpsRedirectionMiddleware

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddHsts(o => options.MaxAge = TimeSpan.FromHours(1));

WebApplication app = builder.Build();

if(app.Environment.IsProduction())
{
    app.UseHsts();
}

app.UseHttpsRedirection(); // Добавляет HttpsRedirectionMiddleware в конвейер. Перенаправляет все HTTP-запросы на HTTPS

app.UseStaticFiles();
app.UseRouting();

app.MapRazorPages();

app.Run();
```

`HttpsRedirectionMiddleware` будет автоматически перенаправлять HTTP-запросы в первую настроенную конечную точку HTTPS вашего приложения. Если оно не настроено на использование HTTPS, то промежуточное ПО *не будет* ничего перенаправлять и вместо этого зарегистрирует предупреждение:

```
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
```

Если вы хотите, чтобы промежуточное ПО выполняло перенаправление на другой порт, о котором знает Kestrel, его можно настроить, задав переменную окружения `ASPNETCORE_HTTPS_PORT`. Иногда это необходимо, если вы используете обратный прокси-сервер, и ее можно задать альтернативными способами, как описано в документации Microsoft: <http://mng.bz/6DDA>.

Использование промежуточного программного обеспечения для перенаправления HSTS и HTTPS рекомендуется при создании серверного приложения, такого как приложение Razor Pages, к которому всегда будет доступ в браузере. Если вы создаете приложение API, то лучшее решение – вообще не слушать небезопасные HTTP-запросы!

Завершение SSL/TLS, пересылка заголовков и обнаружение защищенных запросов

В начале раздела 28.1 я рекомендовал вам рассмотреть завершение HTTPS-запросов на обратном прокси-сервере. Таким образом, пользователь использует HTTPS для связи с обратным прокси-сервером, а обратный прокси-сервер общается с вашим приложением по протоколу HTTP. Благодаря подобной настройке ваши пользователи защищены, но в таком случае ваше приложение не будет связано напрямую с сертификатами TLS.

Для правильной работы `HttpsRedirectionMiddleware` Kestrel требуется какой-то способ узнать, был ли исходный запрос, полученный обратным прокси-сервером, сделан по протоколу HTTP или HTTPS. Обратный прокси-сервер обменивается данными с вашим приложением через HTTP, поэтому Kestrel не может выяснить это без дополнительной настройки.

Стандартный подход, используемый большинством обратных прокси (таких как IIS, NGINX и HAProxy), заключается в добавлении заголовков в запрос перед его пересылкой в приложение. А именно добавляется заголовок `X-Forwarded-Proto`, указывающий на то, какой протокол использовался для исходного запроса: HTTP или HTTPS.

ASP.NET Core включает `ForwardedHeadersMiddleware` для поиска этого заголовка (и других) и обновляет запрос соответствующим образом, чтобы ваше приложение воспринимало запрос, который был изначально защищен протоколом HTTPS как безопасный во всех отношениях.

Если вы используете IIS с методом расширения `UseIisIntegration()`, то пересылка заголовка обрабатывается автоматически. Если вы применяете другой обратный прокси-сервер, например NGINX или HAProxy, то можете

активировать промежуточное ПО, задав для переменной окружения значение true: ASPNETCORE_FORWARDEDHEADERS_ENABLED = true, как было показано в главе 27. В качестве альтернативы можно вручную добавить промежуточное ПО в свое приложение, как показано в разделе 27.3.2.

Когда обратный прокси-сервер пересыпает запрос, ForwardedHeadersMiddleware будет искать заголовок X-Forwarded-Proto и при необходимости обновит детали запроса. Для всех последующих компонентов промежуточного ПО запрос считается безопасным. При добавлении промежуточного ПО вручную важно разместить ForwardedHeadersMiddleware перед вызовом методов UseHsts() или UseHttpsRedirection(), чтобы перенаправленные заголовки были прочитаны и запрос был помечен как безопасный, если необходимо.

28.4.3 Отклонение HTTP-запросов в приложениях API

Браузеры добавляют все больше и больше средств защиты, таких как заголовок HSTS, чтобы защитить пользователей от использования небезопасных HTTP-запросов. Но не все клиенты используют веб-браузер. В этом разделе вы узнаете, почему приложения API обычно должны полностью отключать HTTP.

Если вы создаете приложение API, то зачастую запросы могут поступать не только из браузеров. Ваше приложение API может в первую очередь обслуживать клиентскую платформу в браузере, но оно также может обслуживать мобильные приложения или предоставлять API другим серверным службам. Это означает, что вы не можете рассчитывать только на защиту, встроенную в веб-браузеры, при использовании HTTPS для ваших приложений API.

Кроме того, даже если вы знаете, что все ваши пользователи используют браузер, единственный способ предотвратить отправку запросов через HTTP – это использовать предварительную загрузку HSTS, как мы уже рассмотрели в разделе 28.4.2. Отправка даже одного запроса по HTTP может поставить под угрозу пользователя, поэтому самый безопасный подход – обрабатывать только запросы HTTPS, а не HTTP-запросы. Это лучший вариант для приложений API.

ПРИМЕЧАНИЕ Было бы безопаснее применить тот же подход ко всем браузерным приложениям, но, к сожалению, браузеры в настоящее время по умолчанию используют HTTP-версии приложений.

Вы можете отключить HTTP-запросы для своего приложения, настроив URL-адреса вашего приложения так, чтобы они включали только запросы https://, используя ASPNETCORE_URLS, или применить другой подход, как описано в главе 27. Настройка

```
ASPNETCORE_URLS=https://*:5001
```

будет гарантировать, что ваше приложение будет обслуживать только HTTPS-запросы через порт 5001 и вообще не будет обрабатывать

HTTP-соединения. Это защищает клиентов, поскольку они не смогут отправлять некорректные HTTP-запросы, и это может упростить задачу со стороны сервера, так как не нужно добавлять промежуточное программное обеспечение для перенаправления HTTP.

HTTPS в наши дни является одним из основных требований для повышения безопасности вашего приложения. Его может быть сложно настроить на начальном этапе, но после его корректной настройки можно будет забыть об этом, особенно если вы используете завершение SSL/TLS на обратном прокси-сервере.

К сожалению, большинство других методов обеспечения безопасности требуют гораздо больших усилий, чтобы избежать добавления уязвимости в приложение по мере его роста и развития. В следующей главе мы рассмотрим несколько распространенных атак и узнаем, как ASP.NET Core осуществляет защиту, а также рассмотрим некоторые тонкости, на которые стоит обращать внимание, чтобы защитить чувствительные данные от взлома.

Резюме

- Протокол HTTPS используется для шифрования данных вашего приложения при их передаче от сервера к браузеру и обратно. Это не позволяет третьим лицам видеть или изменять их;
- HTTPS обязателен для приложений в промышленном окружении, поскольку современные браузеры, такие как Chrome и Firefox, явно помечают приложения, не поддерживающие этот протокол HTTPS, как «небезопасные»;
- в промышленном окружении вы можете избежать обработки TLS в своем приложении с помощью завершения SSL/TLS, когда используется обратный прокси-сервер и протокол HTTPS для связи с браузером, но трафик между вашим приложением и обратным прокси-сервером при этом не будет зашифрован. Обратный прокси-сервер может находиться на том же или на другом сервере, например IIS или NGINX, либо это может быть сторонний сервис, такой как Cloudflare;
- можно использовать сертификат разработчика ASP.NET Core или сертификат разработчика IIS Express для активации протокола HTTPS, на локальной машине, во время разработки. Их нельзя использовать в промышленном окружении, но этого достаточно для локального тестирования. Для этого необходимо выполнить команду `dotnet devcerts https --trust` при первой установке .NET SDK, чтобы подтвердить доверие сертификату;
- Kestrel – веб-сервер по умолчанию для ASP.NET Core. Он отвечает за чтение и запись данных из сети и в сеть, анализ байтов на основе базовых HTTP и сетевых протоколов и преобразование необработанных байтов в объекты .NET, которые вы можете использовать в своих приложениях;
- можно настроить сертификат HTTPS для Kestrel и в промышленном окружении, используя раздел конфигурации

`Kestrel:Certificates:Default`. Для этого не требуется никаких изменений в приложении – Kestrel автоматически загрузит сертификат при запуске вашего приложения и будет использовать его для обслуживания HTTPS-запросов;

- можно использовать компонент `HstsMiddleware` для установки HSTS-заголовков для всего приложения, чтобы браузер отправлял HTTPS-запросы вместо HTTP-запросов. Это может быть применено только после того, как в ваше приложение будет отправлен HTTPS-запрос, поэтому лучше всего использовать его вместе с перенаправлением с HTTP на HTTPS;
- вы можете включить предварительную загрузку HSTS для своего приложения, чтобы гарантировать, что HTTP-запросы от браузеров никогда не отправляются и всегда обновляются до HTTPS. Для этого нужно будет настроить свое приложение, как показано в листинге 28.3, с сертификатом TLS, а затем зарегистрировать его по URL-адресу <https://hstspreload.org>. Такая предварительная настройка гарантирует включение вашего приложения во встроенный список сайтов браузера, использующих только HTTPS;
- кроме того, можно обеспечить использование HTTPS для всего приложения с помощью `HttpsRedirectionMiddleware`. Это приведет к перенаправлению любых HTTP-запросов на конечные точки версии HTTPS;
- если вы создаете приложение API, то имеет смысл полностью заблокировать возможность обработки запросов через HTTP и использовать только HTTPS. Мобильные и другие клиенты, работающие за пределами браузера, не имеют такой же защиты, как, например, HSTS, поэтому не существует безопасного способа поддержки одновременно HTTP и HTTPS. Отключение HTTP для приложения прослушивания запросов только по URL-адресам типа `https://` возможно также при использовании настройки значений для `ASPNETCORE_URLS=https://*:5001`.

Повышаем безопасность приложения

В этой главе:

- защита от межсайтового скрипtingа;
- защита от межсайтовой подделки запросов;
- разрешение вызовов API из других приложений с использованием CORS;
- как избежать атак с внедрением вредоносного кода, в частности атак с использованием внедрения SQL-кода.

В главе 28 вы узнали, как и почему вам следует использовать HTTPS в вашем приложении: чтобы защитить HTTP-запросы от злоумышленников. В этой главе мы рассмотрим дополнительные способы защиты вашего приложения и пользователей вашего приложения от злоумышленников. Поскольку безопасность – чрезвычайно обширная тема, охватывающая множество направлений, эта глава ни в коем случае не является исчерпывающим руководством. Она предназначена для того, чтобы познакомить вас с некоторыми из наиболее распространенных угроз вашему приложению и способов противодействия с ним, а также выделить области, в которых вы можете случайно создать уязвимости, если не будете осторожны.

СОВЕТ Я настоятельно рекомендую изучить дополнительные ресурсы по безопасности, после того как вы прочтете эту главу. Открытый проект обеспечения безопасности веб-приложений Open Web Application Security Project (OWASP) ([www.owasp.org](http://www owasp org)) – отличный ресурс, хотя и может показаться немного суховатым. Кроме того, у Троя Ханта есть отличные курсы и семинары по безопасности, ориентированные на разработчиков, работающих с платформой .NET ([www.troyhunt.com](http://www troyhunt com)).

В разделах 29.1 и 29.2 вы узнаете о двух типах потенциальных атак, которые должны быть на вашем радаре: межсайтовый скриптинг (XSS) и межсайтовая подделка запросов (CSRF). Мы изучим, как работают эти атаки и как предотвратить их в своих приложениях. В ASP.NET Core имеется встроенная защита от обоих типов атак, но вы должны не забывать правильно использовать механизмы защиты и не поддаваться искушению обойти их, если только вы не уверены, что это безопасно.

В разделе 29.3 рассматривается распространенный сценарий – у вас есть приложение, которое хочет использовать запросы JavaScript AJAX (Асинхронный JavaScript и XML) для получения данных из второго приложения. По умолчанию веб-браузеры блокируют запросы к другим приложениям, поэтому вам нужно активировать совместное использование ресурсов между разными источниками (CORS) в своем API для достижения этой цели. Мы рассмотрим, как работает CORS, как настроить политику CORS для своего приложения и как применить ее к конкретным конечным точкам.

Последний раздел этой главы, 29.4, охватывает набор распространенных угроз. Каждая из них представляет собой потенциально критическую уязвимость, которую злоумышленник может использовать для взлома вашего приложения. Решение для каждой угрозы в целом относительно простое – важно понять, где в приложениях могут быть эти недостатки, чтобы гарантировать, что вы не подвержены этим уязвимостям.

Как я упоминал в главе 28, вам всегда следует начинать с добавления HTTPS в ваше приложение, чтобы шифровать трафик между браузерами ваших пользователей и вашим приложением. Без HTTPS злоумышленники могут обойти многие меры безопасности, которые вы добавляете в свое приложение, поэтому это важный первый шаг.

К сожалению, большинство других методов обеспечения безопасности требуют гораздо большей бдительности, чтобы случайно не добавить уязвимости в свое приложение по мере его роста и развития. Многие атаки концептуально просты и известны уже много лет, однако они по-прежнему часто встречаются в новых приложениях. В следующем разделе мы рассмотрим одну из таких атак и узнаем, как защититься от нее при создании приложений с использованием Razor Pages.

29.1 Защита от межсайтового скриптинга

В этом разделе я опишу атаки, известные как межсайтовый скриптинг, и как злоумышленники могут использовать их, чтобы скомпромети-

ровать компьютеры ваших пользователей. Я покажу, как фреймворк Razor Pages защищает вас от этих атак, как отключить защиту, когда вам это нужно, и на что обратить внимание. Я также расскажу о разнице между кодировкой HTML и кодировкой JavaScript и какое влияние оказывает использование неправильного кодировщика.

Злоумышленники могут использовать уязвимость в вашем приложении для межсайтового скрипtingа (XSS). Данный тип атаки позволяет запускать код в браузере другого пользователя. Обычно злоумышленники отправляют содержимое, используя легальный ввод, например форму для ввода данных, которая позже отображается где-то на странице. Тщательно поработав с вредоносным вводом, злоумышленник может выполнить вредоносный код JavaScript в браузере пользователя и таким образом может украсть файлы cookie, выдать себя за пользователя и совершить иные нежелательные действия.

СОВЕТ Подробное обсуждение XSS-атак см. в статье «Межсайтовый скрипting (XSS)» на сайте OWASP: <https://owasp.org/www-community/attacks/xss/>.

На рис. 29.1 показан базовый пример такой атаки. Обычные пользователи вашего приложения могут отправить свое имя в ваше приложение, заполнив форму. Затем приложение добавляет имя во внутренний список и отображает весь список на странице. Если имена не отображаются безопасно, злоумышленник может выполнить код JavaScript в браузере любого другого пользователя, который просматривает список.

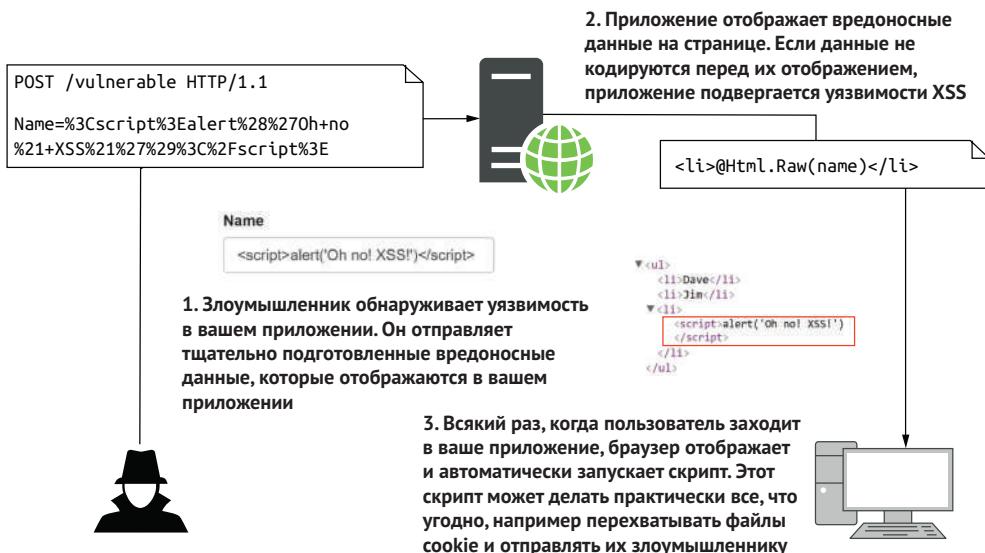


Рис. 29.1 Как используется XSS-уязвимость. Злоумышленник отправляет в ваше приложение вредоносное содержимое, которое отображается в браузерах других пользователей. Если приложение не кодирует содержимое при записи на страницу, ввод становится частью HTML-страницы и может запускать произвольный код JavaScript

На рис. 29.1 пользователь ввел фрагмент HTML, например свое имя. Когда пользователи просматривают список имен, шаблон Razor отображает имена с помощью метода `@Html.Raw()`, который записывает тег `<script>` прямо в документ. Введенные данные любого пользователя становятся частью HTML-разметки страницы. Как только страница загружается в браузере нового пользователя, тег `<script>` выполняется, и компьютер пользователя оказывается скомпрометирован. Как только злоумышленнику удастся выполнить произвольный код JavaScript в браузере обычного пользователя, он сможет делать практически все, что угодно.

СОВЕТ Вы можете снизить вероятность XSS-атак с помощью Content-Security-Policy (CSP). Вы можете прочитать о CSP по адресу <http://mng.bz/nWW2>. У меня есть библиотека с открытым исходным кодом, которую вы можете использовать для интеграции CSP в ваше приложение, доступная на NuGet по адресу <http://mng.bz/vnn4>.

Уязвимость здесь связана с небезопасным отображением пользовательского ввода. Если данные не были закодированы, чтобы сделать их безопасными, прежде чем они будут визуализированы, компьютеры ваших пользователей будут открыты для атаки. *По умолчанию Razor защищает вас от XSS-атак* путем кодирования всех записанных данных в HTML с помощью тег-хелперов, HTML-хелперов или синтаксиса `@`. Поэтому обычно приложения на ASP.NET уже защищены от этого вида атак, как было показано в главе 17.

Однако опасность все же заключается в использовании метода `@Html.Raw()`: если HTML, который вы визуализируете, содержит пользовательский ввод, то у вас может появиться XSS-уязвимость. При выполнении отрисовки пользовательского ввода с символом `@` содержимое кодируется до того, как будет записано в вывод, как показано на рис. 18.6.

В этом примере демонстрируется использование кодирования в HTML для предотвращения непосредственного добавления элементов в HTML DOM, но это не единственный случай, о котором нужно помнить. Если вы передаете недоверенные данные в JavaScript или используете такие данные в значениях URL-запросов, необходимо убедиться, что вы правильно их кодируете.

Распространенный пример – когда вы используете jQuery или JavaScript на страницах Razor и хотите передать значение с сервера клиенту. Если вы используете стандартный символ `@` для отрисовки данных на странице, вывод будет в кодировке HTML. К сожалению, если вы кодируете строку в HTML и вставляете ее непосредственно в код JavaScript, то, вероятно, не получите того, чего ожидаете.

Например, если в вашем файле Razor есть переменная с именем `name` и вы хотите сделать ее доступной в JavaScript, у вас может возникнуть соблазн использовать что-то вроде:

```
<script>var name = '@name'</script>
```



Рис. 29.2 Защита от XSS-атак с помощью HTML-кодирования пользовательского ввода с использованием символа @ в шаблонах Razor. Тег <script> закодирован, поэтому он больше не отображается как HTML и не может быть использован для компрометации приложения

Если имя содержит специальные символы, Razor закодирует их, используя кодировку HTML, а это не то, что нужно в этом контексте JavaScript. Например, если бы у переменной name было значение Arnold "Arnie" Schwarzenegger, тогда при отрисовке, как вы это делали ранее, вы бы получили следующее:

```
<script>var name = 'Arnold &quot;Arnie&quot; Schwarzenegger';</script>
```

Обратите внимание, что двойные кавычки («) были закодированы в HTML как ". Если вы используете это значение в коде JavaScript напрямую, ожидая, что оно будет «безопасным» закодированным значением, то оно будет выглядеть неправильно, как показано на рис. 29.3.

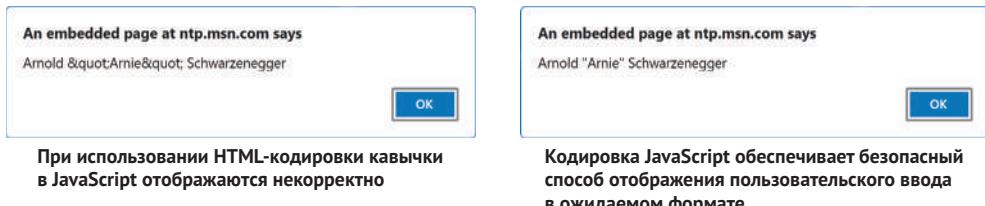


Рис. 29.3 Сравнение предупреждений при использовании кодировки JavaScript и кодировки HTML

Вместо этого нужно закодировать переменную, используя кодировку JavaScript, чтобы двойные кавычки отображались как безопасный символ Юникода, \u0022. Этого можно добиться с помощью внедре-

ния JavaScriptEncoder в представление (как было показано в главе 10) и вызова метода Encode() для переменной name:

```
@inject System.Text.Encodings.Web.JavaScriptEncoder encoder;
<script>var name = '@encoder.Encode(name)';</script>
```

Чтобы избежать необходимости помнить об использовании кодировки JavaScript, рекомендую вам не записывать значения в JavaScript таким образом. Вместо этого запишите значение в атрибуты HTML-элемента, а затем передайте его в переменную JavaScript позже. Это полностью избавляет от необходимости использовать кодировщик JavaScript.

Листинг 29.1. Передача значений в JavaScript путем записи их в атрибуты HTML

```
<div id="data" data-name="@name"></div>
<script>
var ele = document.getElementById('data');
var name = ele.getAttribute('data-name');
</script>
```

Записываем желаемое значение в JavaScript в атрибут data-*
Так вы кодируете данные в HTML
Считывает атрибут data-*
в JavaScript, который преобразует его в кодировку JavaScript

XSS-атаки по-прежнему распространены, и вы легко можете подвергнуться им всякий раз, когда позволяете пользователям вводить данные. Валидация входящих данных иногда может помочь, но часто не все так просто. Например, простой валидатор имени может потребовать, чтобы вы использовали только буквы, что предотвратит большинство атак. К сожалению, здесь не учитываются пользователи, у которых в имени есть дефис или апостроф, не говоря уже о пользователях с незападными именами. Пользователи (по понятным причинам) расстраиваются, когда вы говорите им, что их имя недействительно, поэтому остерегайтесь такого подхода!

Независимо от того, используете вы строгую валидацию или нет, вы всегда должны кодировать данные, когда визуализируете их на странице. Хорошенько подумайте, когда вы пишете @Html.Raw(). Есть ли у пользователя способ ввести вредоносные данные в это поле? Если да, тогда вам нужно будет найти другой способ отображения данных.

XSS-уязвимости позволяют злоумышленникам выполнять код JavaScript в браузере пользователя. Следующая уязвимость, которую мы рассмотрим, позволяет злоумышленнику делать запросы к вашему API, как если бы это был другой пользователь, выполнивший вход, даже если он не использует ваше приложение. Испугались? Надеюсь, что да!

29.2 Защита от межсайтовой подделки запросов (CSRF)

В этом разделе вы узнаете о межсайтовой подделке запросов, а также о том, как злоумышленники могут использовать эти атаки, чтобы вы-

дать себя за пользователя на вашем сайте, и как защититься от них с помощью токенов верификации (*antiforgery tokens*). Razor Pages защищает от этих атак по умолчанию, но вы можете отключить подобные проверки, поэтому важно понимать, к чему это приведет.

Межсайтовая подделка запросов (CSRF) может быть проблемой для веб-сайтов или API, которые используют файлы cookie для аутентификации. Такая атака осуществляется с вредоносного веб-сайта, создающего аутентифицированный запрос к вашему API от имени пользователя, причем пользователь его не инициировал. В этом разделе мы узнаем, как работают эти атаки и как смягчить их последствия с помощью токенов верификации.

Классический пример такой атаки – банковский перевод или вывод средств. Представьте, что у вас есть банковское приложение, которое хранит токены аутентификации в файлах cookie, как это обычно бывает (особенно в традиционных серверных приложениях). Браузеры автоматически отправляют файлы cookie, ассоциированные с доменом, с каждым запросом, чтобы приложение знало, прошел ли пользователь аутентификацию.

Теперь представьте, что в вашем приложении есть страница, которая позволяет пользователю переводить средства со своего счета на другой счет с помощью POST-запроса на страницу Balance. Вам нужно выполнить вход, чтобы получить доступ к форме (вы защитили страницу Razor с помощью атрибута `[Authorize]`), а после этого вы просто отправляете форму, используя POST-запрос, в котором указана сумма, которую вы хотите перевести, с указанием того, куда вы хотите перевести деньги.

Предположим, пользователь заходит на ваш сайт, выполняет вход и совершает транзакцию. Затем он посещает второй сайт, контролируемый злоумышленником. Злоумышленник добавил форму ввода в свой сайт, который выполняет POST-запрос на сайт вашего банка, идентичный тому, что использовался для формы на сайте банка, чтобы перевести средства. Эта форма совершает некие вредоносные действия, например переводит все деньги пользователя злоумышленнику, как показано на рис. 29.4. Браузеры автоматически отправляют файлы cookie для приложения, когда страница выполняет POST-запрос в полной форме, и банковское приложение не может знать, что это злонамеренный запрос. Ничего не подозревающий пользователь отдает все свои деньги злоумышленнику!

Уязвимость здесь вызвана тем, что браузеры автоматически отправляют файлы cookie при запросе страницы (с использованием GET-запроса) или при отправлении содержимого из формы. Нет разницы между легальным POST-запросом для отправки формы в вашем банковском приложении и таким же запросом, исходящим со стороны злоумышленника. К сожалению, подобное поведение встроено в Сеть; это то, что позволяет без проблем путешествовать по сайтам, после того как вы выполнили вход.



Рис. 29.4 CSRF-атака происходит, когда совершивший вход пользователь посещает вредоносный сайт. На этом сайте уже есть форма, которая соответствует форме в вашем приложении, и злоумышленник отправляет ее содержимое в ваше приложение. Браузер отправляет cookie-файл аутентификации автоматически, поэтому ваше приложение воспринимает запрос как действительный запрос от пользователя

Распространенным решением при такой атаке является шаблон *Синхронизирующий токен*, который использует уникальные токены верификации, чтобы обеспечить различие между легитимным POST-запросом и поддельным запросом от злоумышленника. Один токен хранится в файле cookie, а другой добавляется в форму, которую вы хотите защищить. Ваше приложение генерирует токены во время выполнения на основе текущего пользователя, выполнившего вход, поэтому злоумышленник не сможет создать такой токен для своей поддельной формы.

СОВЕТ Браузеры имеют дополнительные средства защиты для предотвращения отправки файлов cookie в этой ситуации, называемые файлами cookie SameSite. По умолчанию большинство браузеров используют SameSite=Lax, что предотвращает возможность такой атаки. Вы можете прочитать о файлах cookie SameSite и о том, как работать с ними в ASP.NET Core, по адресу <http://mng.bz/4DDj>.

Когда со страницы Balance на сервер передан POST-запрос, тогда на сервере сравниваются значения, доступные на сервере, и значения из файла cookie. Если какое-либо значение отсутствует или токены не совпадают, запрос отклоняется. Если злоумышленник создает POST-запрос, браузер отправляет токен из cookie-токена, но в самой форме токена не будет или же он будет недействителен. Страница Razor отклонит запрос, таким образом защищаясь от атаки, как показано на рис. 29.5.



Рис. 29.5 Защита от CSRF-атак с помощью токена верификации. Браузер автоматически пересыпает токен, но вредоносный сайт не может прочитать его и поэтому не может включить в форму. Приложение отклоняет вредоносный запрос, потому что токены не совпадают

Хорошой новостью является то, что Razor Pages автоматически защищает вас от атак подобного рода. Тег-хелпер формы автоматически устанавливает файл cookie токена верификации и отображает токен в скрытое поле `_RequestVerificationToken` для каждого элемента `<form>` в вашем приложении (если вы специально не отключили их). Например, возьмем этот простой шаблон Razor, который отправляется обратно на ту же страницу Razor:

```
<form method="post">
<label>Amount</label>
<input type="number" name="amount" />
<button type="submit">Withdraw funds</button>
</form>
```

При отрисовке в HTML токен верификации хранится в скрытом поле и отсылается обратно с подлинным запросом:

```
<form method="post">
<label>Amount</label>
<input type="number" name="amount" />
<button type="submit">Withdraw funds</button>
<input name="__RequestVerificationToken" type="hidden">
<input value="CfDJ8Daz26qb0hBGsw7QCK" />
</form>
```

ASP.NET Core автоматически добавляет эти токены в каждую форму, а Razor Pages автоматически проверяет их. Фреймворк гарантирует, что токены есть и в файле cookie, и в данных формы, и гарантирует их совпадение и отклоняет любые запросы в случае, если они не совпадают.

Если вы используете контроллеры MVC с представлениями вместо Razor Pages, то ASP.NET Core по-прежнему добавляет токены верификации в каждую форму. К сожалению, он их не проверяет автоматически. Чтобы включить проверку, нужно добавлять в контроллеры специальный атрибут `[ValidateAntiForgeryToken]`. Это гарантирует, что при работе таких контроллеров и представлений для них в браузере соответствие токенов из cookie и на сервере будет проверяться автоматически, и если они не совпадут, тогда сервер отклонит такие запросы.

ПРЕДУПРЕЖДЕНИЕ ASP.NET Core *не* проверяет токены верификации автоматически, если вы используете контроллеры MVC с представлениями. Убедитесь, что вы пометили все уязвимые методы атрибутами `[ValidateAntiForgeryToken]`, как описано в разделе «Предотвращение атак с межсайтовой подделкой запросов (XSRF / CSRF)» в документации по ASP.NET Core: <http://mng.bz/QPPv>. Обратите внимание: если вы не используете файлы cookie для аутентификации, вы не уязвимы для атак CSRF: CSRF-атаки возникают в результате того, что злоумышленники используют тот факт, что браузеры автоматически присоединяют файлы cookie к запросам. Нет cookie – нет проблем!

Как правило, нужно использовать эти токены только для запросов с методами POST, DELETE и других опасных типов запросов, которые используются для изменения состояния. GET-запросы для этой цели не используются, поэтому фреймворк не требует защитных токенов для их вызова. Razor Pages проверяет эти токены, если речь об опасных методах, таких как POST, и игнорирует безопасные методы, такие как GET. Пока вы создаете свое приложение, следя этому шаблону (а вы должны следовать ему!), фреймворк сделает все возможное, чтобы вы были в безопасности.

Если вам по какой-то причине необходимо явно игнорировать эти токены на странице Razor, то можете отключить проверку, применив атрибут `[IgnoreAntiforgeryToken]` в `PageModel` страниц Razor. Это позволяет обойти защиту фреймворка для тех случаев, когда вы уверены, что делаете нечто безопасное, от чего не требуется защита, но в большинстве случаев лучше перестраховаться и все проверить.

Защита от CSRF-атак может быть непростой задачей с технической точки зрения, но по большей части все *должно* работать без особых усилий с вашей стороны. Razor добавит токены в ваши формы, а фреймворк Razor Pages позаботится о валидации.

Все становится сложнее, если вы делаете много запросов к API с помощью JavaScript и отправляете объекты в формате JSON, а не данные формы. В таких случаях вы не сможете отправить токен верификации как часть формы (потому что вы отправляете JSON), поэтому вам нужно будет вместо этого добавить его в качестве заголовка в запрос. Документация Microsoft («Предотвращение атак с межсайтовой подделкой запросов (XSRF / CSRF)» в ASP.NET Core) содержит примеры с использованием JQquery и AngularJS, но вы должны иметь возможность распространить это на выбранный вами фреймворк JavaScript: <http://mng.bz/54Sl>.

СОВЕТ Если вы не используете аутентификацию с файлами cookie и у вас есть одностраничное приложение, отправляющее токены аутентификации в заголовке, то это хорошая новость – вам вообще не нужно волноваться о межсайтовой подделке запросов! Вредоносные сайты могут отправлять вашему API только файлы cookie, но не заголовки, поэтому они не могут выполнять запросы, прошедшие аутентификацию.

Создание уникальных токенов с API для защиты данных

Зашитные токены, используемые для предотвращения межсайтовой подделки запросов, зависят от возможностей фреймворка использовать сильное симметричное шифрование для шифрования и расшифровки данных. Алгоритмы шифрования обычно полагаются на один или несколько ключей, которые используются, чтобы инициализировать шифрование и сделать этот процесс воспроизводимым. Если у вас есть ключ, вы можете шифровать и расшифровывать данные; без этого ключа такие данные невозможна украдь.

В ASP.NET Core шифрование выполняют API защиты данных (data protection API). Они создают защитные токены, шифруют cookie-файлы аутентификации и генерируют безопасные токены. Что особенно важно – они также контролируют управление файлами ключей, которые используются для шифрования. Файл ключа – это небольшой XML-файл, который содержит случайное значение ключа, используемое для шифрования в приложениях ASP.NET Core. Очень важно, чтобы он хранился в надежном месте: если злоумышленник завладеет им, то сможет выдавать себя за любого пользователя вашего приложения!

Система защиты данных хранит ключи в надежном месте, в зависимости от того, как и где вы размещаете приложение:

- веб-приложение размещено в облаке Azure – в специальной синхронизированной папке, доступной всем регионам;
- на сервере IIS без профиля пользователя – зашифрованными в реестре;
- учетная запись с профилем пользователя – в %LOCALAPPDATA%\ASP.NET\DataProtection-Keys в Windows или ~/aspnet/DataProtection-Keys в Linux или macOS;
- во всех остальных случаях – в памяти; при перезапуске приложения ключи будут потеряны.

Почему вас это должно волновать? Чтобы ваше приложение могло читать cookie-файлы аутентификации ваших пользователей, оно должно иметь возможность расшифровать их, используя тот же ключ, который применялся для их шифрования. Если вы работаете с несколькими серверами, то по умолчанию у каждого сервера будет свой ключ, и вы не сможете читать файлы cookie, зашифрованные другими серверами.

Чтобы обойти эту проблему, нужно сконфигурировать свое приложение для хранения ключей защиты данных в едином месте. Это может быть совместно используемая папка на жестком диске, экземпляр Redis или, например, экземпляр хранилища BLOB-объектов Azure.

Документация Microsoft по API защиты данных чрезвычайно подробная, но может показаться сложной. Рекомендую прочитать раздел по настройке защиты данных (<http://mng.bz/d40i>) и настройке провайдера хранилища ключей для использования в том случае, если у вас несколько серверов (<https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/implementation/key-storage-providers?view=aspnetcore-5.0&tabs=visual-studio>).

Стоит пояснить, что уязвимости при межсайтовой подделке запросов, обсуждаемые в этом разделе, требуют, чтобы вредоносный сайт выполнял к вашему приложению POST-запрос в полной форме. Вредоносный сайт не может запрашивать ваш API, используя только код JavaScript *на стороне клиента*, поскольку браузеры будут блокировать запросы к вашему API из другого источника.

Это функция безопасности, но она часто может вызывать проблемы. Если вы создаете одностраничное клиентское приложение, или даже если у вас есть небольшое количество кода JavaScript в приложении с отрисовкой на стороне сервера, то вы можете обнаружить, что вам нужно делать подобные запросы с разных страниц. В следующем разделе я опишу типичный сценарий, с которым вы, вероятно, столкнетесь, и покажу, как изменить приложение, чтобы обойти его.

29.3 Вызов веб-API из других доменов с помощью CORS

В этом разделе вы узнаете о совместном использовании ресурсов между разными источниками (CORS), протоколе, позволяющем JavaScript делать запросы из одного домена в другой. CORS часто смущает многих разработчиков, поэтому в этом разделе описывается, почему он необходим и как работают заголовки CORS. Затем вы узнаете, как добавить CORS для своего приложения и для определенных действий веб-API, а также как настроить несколько политик CORS для своего приложения.

Как вы уже видели, атаки с межсайтовой подделкой запросов могут быть довольно мощными, но они были бы еще опаснее, если бы не браузеры, реализующие *правило ограничения домена*. Данная концепция запрещает приложениям использовать JavaScript для вызова веб-API в другом месте, если только это не разрешено явно веб-API.

ОПРЕДЕЛЕНИЕ Источники считаются одинаковыми, если они соответствуют протоколу (HTTP или HTTPS), домену (example.com) и порту (80 по умолчанию для HTTP и 443 для HTTPS). Если приложение пытается получить доступ к ресурсу с помощью JavaScript и источники не идентичны, браузер блокирует запрос.

Правило ограничения домена является строгим: источники двух URL-адресов должны быть идентичны, чтобы запрос был разрешен. Например, следующие источники одинаковы:

- <http://example.com/home>;
- <http://example.com/site.css>.

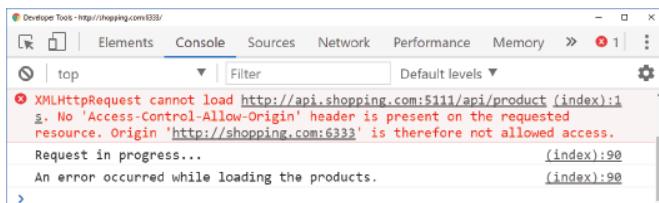
Пути этих двух адресов различаются (`/home` и `/site.css`), но протокол, домен и порт (80) идентичны. Итак, если бы вы находились на главной странице своего приложения, то могли бы без проблем запросить файл `/site.css` с помощью JavaScript.

Напротив, источники следующих сайтов различаются, поэтому вы не можете запросить ни один из этих URL-адресов с помощью JavaScript из источника <http://example.com>:

- `https://example.com` – другой протокол (https);
- `http://www.example.com` – другой домен (включая поддомен);
- `http://example.com:5000` – другой порт.

Что касается простых приложений, например когда вся функциональность сосредоточена в одном веб-приложении, такое ограничение может и не быть проблемой, но очень часто приложение отправляет запросы в иной домен.

Например, у вас может быть сайт для онлайн-торговли, расположенный по адресу <http://shopping.com>, и вы пытаетесь загрузить данные с сайта <http://api.shopping.com>, чтобы отобразить подробные сведения о товарах, имеющихся в продаже. При такой конфигурации вы столкнетесь с правилом ограничения домена. Любая попытка выполнить запрос с помощью JavaScript к домену `api.shopping.com` завершится ошибкой, аналогичной той, что показана на рис. 29.6.



Браузер по умолчанию не разрешает перекрестные запросы и блокирует доступ вашего приложения к ответу

Рис. 29.6 Журнал консоли для неудавшегося запроса. Chrome заблокировал запрос от приложения <http://shopping.com:6333> к API по адресу <http://api.shopping.com:5111>

Необходимость выполнять запросы, используя разные источники, из JavaScript становится все более распространенным явлением с ростом числа клиентских одностраничных приложений и отходом от монолитных приложений. К счастью, есть веб-стандарт, позволяющий безопасно обойти эту проблему, – это совместное использование ресурсов между разными источниками (CORS). Вы можете использовать его, чтобы контролировать, какие приложения могут вызывать ваш API, дабы иметь возможность разрешать сценарии, подобные тому, который я только что описал.

29.3.1 Что такое CORS и как он работает

CORS – это веб-стандарт, позволяющий вашему веб-API делать заявления о том, кто может выполнять к нему запросы из разных источников. Например, можно:

- разрешить запросы от <http://shopping.com> и <https://app.shopping.com>;
- разрешить запросы из разных источников, только если используется метод GET;

- разрешить возвращать заголовок Server в ответах на запросы из разных источников;
- разрешить отправку учетных данных (таких как cookie-файлы аутентификации или заголовки авторизации) с запросами из разных источников.

Можно объединить эти правила в политику и применять разные политики к разным конечным точкам своего API. Вы можете применить политику ко всему приложению или отдельную политику к каждому действию API.

CORS использует HTTP-заголовки. Когда ваш веб-API получает запрос, он задает специальные заголовки для ответа, чтобы указать, разрешены ли запросы из разных источников, что это за источники и какие HTTP-методы и заголовки может использовать запрос, – почти все, что касается запроса.

В некоторых случаях перед отправкой реального запроса к вашему API браузер отправляет *предварительный* запрос. Он отправляется с использованием метода OPTIONS, который браузер использует, чтобы проверить, разрешено ли сделать настоящий запрос. Если API вернет правильные заголовки, браузер отправит настоящий запрос, как показано на рис. 29.7.

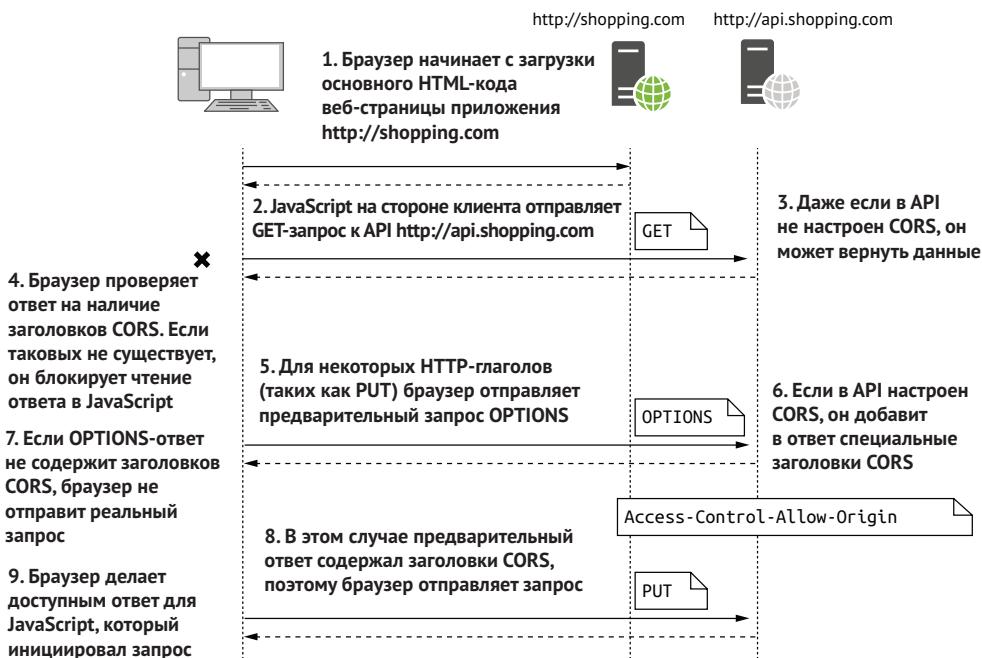


Рис. 29.7 Два запроса из разных источников. Ответ на первый запрос не содержит заголовки CORS, поэтому браузер блокирует их чтение приложением. Второй запрос требует предварительного запроса с помощью метода OPTIONS, чтобы проверить, активирован ли CORS. Поскольку ответ содержит заголовки CORS, можно сделать настоящий запрос и предоставить ответ приложению JavaScript

СОВЕТ Для более подробного обсуждения см. книгу «*CORS в действии*» Монсара Хуссейна (Manning, 2014): <http://mng.bz/aD41>.

Спецификация CORS, как и многие технические документы, довольно сложна и содержит множество заголовков и процессов, с которыми придется столкнуться. К счастью, в ASP.NET Core все эти детали спецификации настраиваются автоматически, поэтому ваша основная задача – точно определить, кому и при каких обстоятельствах нужен доступ к вашему API.

29.3.2 Добавление глобальной политики CORS ко всему приложению

Как правило, не следует настраивать CORS для своих API, пока он вам не понадобится. Браузеры не просто так блокируют обмен данными между источниками – так они закрывают путь для многих атак. И пока у вас не появится API в домене, отличном от приложения, которому необходимо получить к нему доступ, не стоит заморачиваться на эту тему.

Чтобы добавить поддержку CORS в приложение, необходимы четыре вещи:

- добавить сервисы CORS в свое приложение;
- настроить хотя бы одну политику CORS;
- добавить промежуточное ПО CORS в конвейер;
- задать политику CORS по умолчанию для всего приложения либо декорировать действия веб-API атрибутом [EnableCors], чтобы выборочно активировать CORS для определенных конечных точек.

Чтобы добавить службы CORS в приложение, вызовите метод AddCors() для экземпляра WebApplication-Builder из файла Program.cs:

```
builder.Services.AddCors();
```

Большая часть ваших усилий по настройке CORS уйдет на конфигурирование политики. Политика CORS контролирует, как ваше приложение будет отвечать на запросы из разных источников. Она определяет, какие источники разрешены, какие заголовки возвращать, какие HTTP-методы разрешить и т. д. Обычно политики определяются одновременно с добавлением сервисов CORS в свое приложение. Например, рассмотрим предыдущий пример с сайтом для онлайн-торговли.

Вы хотите, чтобы ваш API, размещенный на <http://api.shopping.com>, был доступен из основного приложения через клиентский JavaScript, размещенный на <http://shopping.com>. Поэтому необходимо настроить API, чтобы разрешить запросы из разных источников.

ПРИМЕЧАНИЕ Помните, что именно основное веб-приложение будет получать ошибки при попытке делать запросы из разных источников, однако CORS добавляется к API, к которому вы получаете доступ, а не в веб-приложение, выполняющее запросы.

В следующем листинге показано, как настроить политику "AllowShoppingApp", чтобы разрешить запросы из разных источников от <http://shopping.com> к API. Кроме того, мы явно разрешаем любой тип HTTP-метода; без этого вызова разрешены только простые методы (GET, HEAD и POST). Настройка производится с использованием построителя (fluent builder), который вы встречали на протяжении всей этой книги.

Листинг 29.2 Настройка политики CORS, чтобы разрешить запросы из определенного источника

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options => {
        options.AddPolicy("AllowShoppingApp", policy =>
            policy.WithOrigins("http://shopping.com")
                  .AllowAnyMethod());
    });
    // Другая конфигурация сервисов;
}

Позволяет всем HTTP-методам вызывать API
```

У каждой политики есть уникальное имя

Метод AddCors предоставляет перегруженный вариант Action<CorsOptions>

Метод WithOrigins указывает, какие источники разрешены. Обратите внимание, что URL-адрес не имеет завершающего символа в виде “/”

ВНИМАНИЕ! При перечислении источников в методе `WithOrigins()` убедитесь, что у них нет завершающего символа в виде наклонной черты «/»; в противном случае источники не совпадут и ваши запросы не будут выполнены.

После того как вы определили политику CORS, вы можете применить ее к своему приложению. В следующем листинге применяется политика "AllowShoppingApp" ко всему приложению с помощью `CorsMiddleware` путем вызова метода `UseCors()` в методе `Configure` файла `Startup.cs`.

Листинг 29.3 Добавление промежуточного ПО CORS и настройка политики CORS по умолчанию

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddCors(options => {
    options.AddPolicy("AllowShoppingApp", policy =>
        policy.WithOrigins("http://shopping.com")
              .AllowAnyMethod());
});
var app = builder.Build();
app.UseRouting();
app.UseCors("AllowShoppingApp");
app.UseAuthentication();
app.UseAuthorization();

app.MapGet("/api/products", () => new string[] {});
app.Run();
```

Добавляет промежуточное программное обеспечение CORS и использует AllowShoppingApp в качестве политики по умолчанию

ПРИМЕЧАНИЕ Как и в случае со всеми компонентами промежуточного ПО, важен порядок обработки запроса этим компонентом CORS. Нужно разместить вызов `UseCors()` после метода `UseRouting()` и до метода `UseEndpoints()`. Промежуточное ПО для CORS должно перехватывать запросы из разных источников к вашим веб-API, чтобы иметь возможность генерировать правильные ответы на предварительные запросы и добавлять необходимые заголовки. Обычно промежуточное ПО для CORS размещается перед вызовом `UseAuthentication()`.

Имея такое промежуточное ПО для API, приложение для онлайн-торговли теперь может выполнять запросы из разных источников. Вы можете вызвать API с сайта <http://shopping.com>, и браузер пропускает CORS-запрос, как показано на рис. 29.8. Если вы сделаете тот же запрос из домена, отличного от <http://shopping.com>, запрос останется заблокированным.

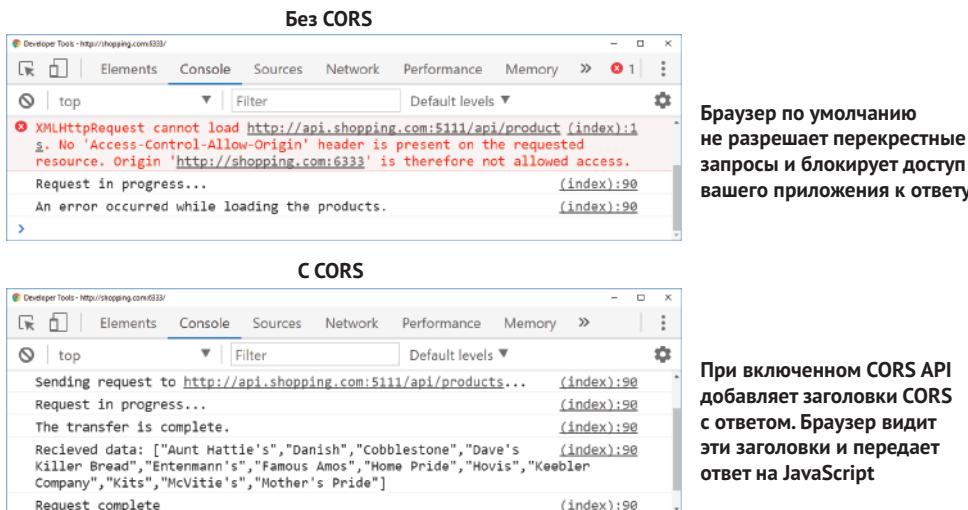


Рис. 29.8 При активации CORS, как показано на изображении ниже, могут выполняться запросы из разных источников, и браузер сделает ответ доступным для JavaScript. Сравните это с верхним изображением, на котором запрос был заблокирован

Глобальное применение политики CORS ко всему приложению может оказаться излишним. Если в вашем API только часть конечных точек могут обрабатывать запросы из разных источников, то нужно будет настраивать CORS лишь для этих конкретных конечных точек. Это можно сделать с помощью атрибута `[EnableCors]`.

29.3.3 Добавление CORS к определенным конечным точкам с помощью метаданных `EnableCors`

Браузеры по умолчанию блокируют запросы между источниками по понятной причине: они могут быть использованы вредоносны-

ми или скомпрометированными сайтами. Включение CORS для всего вашего приложения может не стоить риска, если вы знаете, что только к части логики потребуется доступ из сторонних источников.

В этом случае лучше всего включить политику CORS только для этих конкретных конечных точек. В ASP.NET Core есть метод `RequireCors()`, который можно применить к отдельным конечным точкам или группам маршрутов в минимальных API, а также атрибут `[EnableCors]`, который позволяет выбрать политику для применения к данному контроллеру или методу действия.

ПРИМЕЧАНИЕ Оба этих метода добавляют метаданные CORS в конечную точку, которая используется `CorsMiddleware` для определения применимой политики. Вот почему `CorsMiddleware` следует размещать после `RoutingMiddleware`, чтобы `CorsMiddleware` знал, какая конечная точка была выбрана и какую политику CORS применить.

С помощью метода `RequireCors()` и атрибута `[EnableCors]` вы можете применять разные политики CORS к разным конечным точкам. Например, вы можете разрешить GET-запросы ко всему вашему API из домена <http://shopping.com>, но при этом для других типов HTTP-запросов, настроить доступ на уровне отдельных конечных точек, при этом одновременно позволяя любому источнику получить доступ к конечным точкам списка продуктов.

Политики CORS определяются при вызове метода `AddCors()`, вызывая метод `AddPolicy()` и присваивая политике имя, как вы видели в листинге 29.2. Если вы используете политики, специфичные для конечной точки, вместо вызова `UseCors("AllowShoppingApp")`, как вы видели в листинге 29.3, следует добавить промежуточное ПО без политики по умолчанию, вызвав только `UseCors()`.

Затем вы можете выборочно включить CORS для отдельных конечных точек и указать используемую политику. Чтобы применить CORS к конечной точке минимального API или к группе маршрутов, используйте `RequireCors("AllowShoppingApp")`, как показано в следующем листинге. Чтобы применить политику к контроллеру или методу действия, примените атрибут `[EnableCors("AllowShoppingApp")]`. Вы можете также отключить доступ к конечной точке из других источников, применив атрибут `[DisableCors]`.

Листинг 29.4. Применение политики CORS к конечным точкам минимального API

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddCors(options => { /* Config not shown */});
```

```
var app = builder.Build();
app.UseCors();
```

Добавляет `CorsMiddleware` без настройки политики по умолчанию

```

    app.MapGet("/api/products", () => new string[] {})
        .RequireCors("AllowShoppingApp"); ←

    app.MapGet("/api/products",
        [EnableCors("AllowShoppingApp")] () => new { }); ←

    app.MapGroup("/api/categories")
        .RequireCors("AllowAnyOrigin"); ← | Можно применять политику
                                            CORS для групп маршрутов

    app.MapDelete("/api/products",
        [DisableCors] () => Results.NoContent()); ←

    app.Run(); ← | Атрибут DisableCors полностью от-
                    ключает CORS для метода действия

```

Если вы хотите применить политику CORS к большей части методов действия, но хотите использовать другую политику или полностью отключить CORS для некоторых из них, то можете передать политику по умолчанию при настройке промежуточного ПО, например с помощью `UseCors("AllowShoppingApp")`. Методы действия с атрибутом `[EnableCors("OtherPolicy")]` будут иметь приоритет `OtherPolicy`, а для методов действий с атрибутом `[DisableCors]` CORS вообще не будет активирован.

Независимо от того, хотите ли вы использовать одну политику CORS по умолчанию или несколько, необходимо настроить политики CORS для своего приложения в методе `ConfigureServices`. При настройке CORS доступно множество различных параметров. В следующем разделе я предоставлю их обзор.

29.3.4 Настройка политик CORS

Браузеры реализуют политику CORS из соображений безопасности, поэтому следует внимательно подумать о последствиях ослабления любых налагаемых ими ограничений. Даже если вы активируете возможность отправки запросов из разных источников, то все равно сможете контролировать, какие данные эти запросы могут отправить и что вернет ваш API. Например, можно настроить:

- источники, которые смогут делать запросы к вашему API;
- HTTP-методы (такие как GET, POST и DELETE), которые можно использовать;
- заголовки, которые может отправлять браузер;
- заголовки, которые браузер может прочитать из ответа вашего приложения;
- будет ли браузер отправлять с запросом учетные данные для аутентификации.

Все эти параметры определяются при создании политики CORS в вызове метода `AddCors()`, используя метод `CorsPolicyBuilder`, как вы видели

в листинге 18.5. Политика может задать все эти параметры или ни один из них, чтобы вы могли настроить результаты по своему усмотрению. В табл. 29.1 показаны некоторые доступные параметры и что они делают.

Таблица 29.1 Доступные методы для настройки политики CORS и что они делают

Пример метода CorsPolicyBuilder	Результат
WithOrigins("http://shopping.com")	Разрешает запросы из разных источников от http://shopping.com
AllowAnyOrigin()	Разрешает запросы из любого источника. Это означает, что любой веб-сайт может делать JavaScript-запросы к вашему API
WithMethods() / AllowAnyMethod()	Задает разрешенные HTTP-методы (такие как GET, POST и DELETE), с помощью которых можно выполнять запросы к вашему API
WithHeaders() / AllowAnyHeader()	Задает заголовки, которые браузер может отправлять вашему API. Если вы ограничиваете заголовки, то должны включить по меньшей мере заголовки "Accept", "Content-Type" и "Origin", чтобы разрешить действительные запросы
WithExposedHeaders()	Позволяет вашему API отправлять в браузер дополнительные заголовки. По умолчанию в ответе отправляются только заголовки Cache-Control, Content-Language, Content-Type, Expires, LastModified и Pragma
AllowCredentials()	По умолчанию браузер не отправляет детали аутентификации с запросами из различных источников, только если вы явно не разрешите это. Вы также должны активировать отправку учетных данных на стороне клиента в JavaScript при создании запроса

Один из первых моментов, возникающих при настройке CORS, – осознание того, что у вас могут быть проблемы с доступом из внешних источников. Несколько раз я был в тупике, пытаясь понять, почему запрос не работает, пока не понял, что это кросс-доменный запрос или он идет с HTTP на HTTPS, например.

По возможности я рекомендую полностью избегать запросов из внешних источников. Вы можете столкнуться с небольшими различиями в том, как браузеры обрабатывают их, что может причинить еще больше головной боли. В частности, избегайте кросс-доменных проблем при переходе с HTTP на HTTPS, запуская все свои приложения с использованием протокола HTTPS. Как уже обсуждалось в разделе 18.1, в любом случае это более подходящая практика, и она поможет избежать головной боли, связанной с CORS.

СОВЕТ Другой (часто предпочтительный) вариант – настроить политики CORS в обратном прокси-сервере или шлюзе приложений. Например, вы можете настроить Службу приложений Azure с разрешенными источниками, чтобы вам не приходилось изменять код приложения.

Когда я решаю, что мне определенно нужна политика CORS, то обычно начинаю с метода `WithOrigins()`. Затем при необходимости я расши-

ряю или ограничиваю политику, чтобы обеспечить блокировку моего API для запросов из иных источников, сохранив при этом необходимую функциональность. Обойти CORS может быть сложно, но помните, что эти ограничения существуют для безопасности ваших приложений.

Запросы из разных источников – всего лишь один из множества потенциальных способов, с помощью которых злоумышленники могут скомпрометировать ваше приложение. От многих из них легко защищаться, но нужно знать о них и знать, как смягчить последствия таких атак. В следующем разделе мы рассмотрим распространенные угрозы и способы избежать их.

29.4 Изучение других векторов атак

Пока я описал два потенциальных способа, с помощью которых злоумышленники могут скомпрометировать ваши приложения, – межсайтовый скрипting и межсайтовая подделка запросов – и как их предотвратить. Обе эти уязвимости регулярно входят в десятку самых важных рисков для веб-приложений по версии OWASP, поэтому важно знать о них и избегать их появления в своих приложениях.

СОВЕТ OWASP публикует список в интернете с описанием каждой атаки и способами предотвращения этих атак. Здесь есть шпаргалка по безопасности: <https://cheatsheetseries.owasp.org/>.

В этом разделе я дам обзор других наиболее распространенных уязвимостей и расскажу, как избежать их появления в своих приложениях.

29.4.1 Обнаружение и предотвращение атак с открытым перенаправлением

Одна из распространенных уязвимостей, описанная на сайте OWASP, связана с атаками с *открытым перенаправлением*. Это атака, при которой пользователь щелкает по ссылке, ведущей на другое безопасное приложение, и в конечном итоге попадает на вредоносный веб-сайт, например обслуживающий вредоносное ПО. Безопасное приложение не содержит прямых ссылок на вредоносный сайт, так как же это происходит?

Атаки с открытым перенаправлением происходят, когда следующая страница передается в качестве параметра методу действия. Самый распространенный пример – когда вы входите в приложение. Обычно приложения запоминают страницу, на которой находится пользователь, прежде чем перенаправить его на страницу входа, передавая текущую страницу в качестве параметра строки запроса `returnUrl`. После того как пользователь выполняет вход, приложение перенаправляет пользователя на `returnUrl`, чтобы продолжить с того места, где он остановился.

Представьте, что пользователь просматривает сайт для онлайн-торговли. Он нажимает кнопку «Купить» рядом с продуктом и перенаправляется на страницу входа. Страница продукта, на которой он находился, передается в виде `returnUrl`, поэтому после входа он перенаправляется на страницу продукта вместо переброски на главную страницу.

Атака с открытым перенаправлением использует этот общепринятый шаблон, как показано на рис. 29.9. Злоумышленник создает URL-адрес для входа, где для returnUrl задается значение в виде веб-сайта, на который он хочет отправить пользователя. Злоумышленник убеждает пользователя щелкнуть по ссылке, ведущей на ваше веб-приложение. После того как пользователь выполнит вход, уязвимое приложение перенаправит пользователя на вредоносный сайт.

Простой способ решить такую проблему – всегда проверять, что returnUrl является локальным URL-адресом, принадлежащим вашему приложению, *до* того, как перенаправить пользователей на него. Пользовательский интерфейс Identity по умолчанию уже делает это, поэтому вам не нужно беспокоиться о странице входа, если вы используете эту систему, как описано в главе 23.

Если у вас есть перенаправления в других частях приложения, ASP.NET Core предоставляет несколько вспомогательных методов для обеспечения безопасности, наиболее полезным из которых является `Url.IsLocalUrl()`. В следующем листинге показано, как убедиться, что предоставленный returnUrl безопасен, а если это не так, то выполнить перенаправление на главную страницу приложения. Вы также можете использовать вспомогательный метод `LocalRedirect()` в классах `ControllerBase` и `PageModel`, который возбуждает исключение, если предоставленный URL-адрес не является локальным.



Рис. 29.9 Открытое перенаправление использует распространенный шаблон URL-адреса для возврата. Обычно он используется для страниц входа, но может применяться и в других областях вашего приложения. Если приложение не проверяет, что URL-адрес безопасен для перенаправления пользователя, то может перенаправлять пользователей на вредоносные сайты

Листинг 29.5 Обнаружение атак с открытым перенаправлением путем проверки локальных URL-адресов возврата

```
[HttpPost]
public async Task<IActionResult> Login(
    LoginViewModel model, string returnUrl = null)
{
    // Проверяем пароль и даем пользователю выполнить вход;
    if (Url.IsLocalUrl(returnUrl)) ←
    {
        return Redirect(returnUrl); ←
    }
    else
    {
        // URL-адрес не был локальным и мог представлять собой атаку
        // с открытым перенаправлением, поэтому в целях безопасности
        // выполняем перенаправление на домашнюю страницу
        return RedirectToPage("Index"); ←
    }
}
```

URL-адрес возврата пред-
ставляется в качестве аргу-
мента метода действия

Возвращает true, если URL-адрес воз-
врата начинается с символа / или ~/

URL-адрес является локальным, поэтому
перенаправление на него безопасно

Этот простой шаблон обеспечивает защиту от атак открытого перенаправления, которые в противном случае могли бы предоставить вашим пользователям доступ к вредоносному контенту. Всякий раз, когда вы перенаправляете пользователя на адрес, который идет из строки запроса или другого пользовательского ввода, вы должны использовать этот шаблон.

В некоторых потоках аутентификации, например при аутентификации с помощью OpenID Connect, вы не можете перенаправить на локальный URL-адрес, поэтому вы не можете использовать этот шаблон.

Вместо этого OpenID Connect требует предварительной регистрации разрешенных URL-адресов перенаправления и перенаправления только на зарегистрированный URL-адрес. Вам следует рассмотреть возможность использования этого шаблона, если вы не можете обеспечить локальное перенаправление.

29.4.2 Предотвращение атак с использованием внедрения SQL-кода с помощью EF Core и параметризации

Атаки с открытым перенаправлением представляют риск для ваших пользователей, но не для вашего приложения напрямую, а следующая уязвимость представляет собой опасность уже для самого приложения. Атаки путем внедрения SQL-кода представляют собой одну из самых опасных угроз для приложения. Злоумышленники создают простой вредоносный код, который они отправляют в ваше приложение в виде традиционного ввода на основе форм или путем настройки URL-адресов и строк запроса для выполнения произвольного кода в вашей базе данных. Данная уязвимость может предоставить доступ злоумышленникам ко всей вашей базе данных, поэтому очень важно находить и устранять любые подобные уязвимости в своих приложениях.

Надеюсь, я вас немного напугал этим вступлением, а теперь перейдем к хорошим новостям: если вы используете EF Core (или почти любой другой ORM-инструмент) стандартным образом, то вы в безопасности. EF Core имеет встроенную защиту от внедрения SQL-кода, поэтому если вы не делаете ничего экстравагантного, все будет в порядке. Подобные уязвимости возникают, когда вы сами создаете инструкции SQL и включаете динамический ввод, предоставляемый злоумышленником, пусть даже косвенно. EF Core предоставляет возможность создавать низкоуровневые SQL-запросы с помощью метода `FromSqlRaw()`, поэтому вы должны быть осторожными, когда используете его.

Представьте, что в вашем приложении с рецептами есть форма поиска, позволяющая искать рецепт по названию. Если вы пишете запрос с использованием методов расширения LINQ (как описано в главе 12), то атаки с использованием внедрения SQL-кода вам не грозят. Однако если вы решите написать SQL-запрос вручную, то откроете путь для подобной уязвимости.

Листинг 29.6 Внедрение SQL-кода в EF Core, обусловленное конкатенацией строк

```
Текущий DbContext хранится в поле _context
public IList<User> FindRecipe(string search) {
    return _context.Recipes
        .FromSqlRaw("SELECT * FROM Recipes" +
                    "WHERE Name = '" + search + "'")
        .ToList();
}

Параметр поиска вводится пользователем, поэтому это небезопасно
Вы можете писать запросы вручную, используя метод расширения FromSqlRaw
Внедряет уязвимость, включая небезопасный контент непосредственно в строку SQL
```

В этом листинге пользовательский ввод в параметре `search` включен непосредственно в SQL-запрос. Создавая вредоносный ввод, пользователи потенциально могут выполнять любые операции с вашей базой данных. Представьте, что злоумышленник выполняет поиск по нашему сайту, используя текст

```
'; DROP TABLE Recipes; --
```

Ваше приложение назначает его параметру `search`, и SQL-запрос, выполняемый к вашей базе данных, принимает следующий вид:

```
SELECT * FROM Recipes WHERE Name = ''; DROP TABLE Recipes; --'
```

Просто введя текст в форму поиска вашего приложения, злоумышленник удалил из него всю таблицу рецептов! Это катастрофа, и подобная уязвимость обеспечивает более или менее неограниченный доступ к базе данных. Даже если вы правильно настроили полномочия для базы данных, чтобы предотвратить такого рода деструктивные действия, злоумышленники, вероятно, смогут прочитать все данные из вашей базы, включая данные пользователей.

Самый простой способ избежать этого – не создавать SQL-запросы вручную. Если вам действительно нужно написать собственные SQL-запросы, не используйте конкатенацию строк, как в листинге 29.6. Используйте параметризованные запросы, в которых (потенциально небезопасные) входные данные отделены от самого запроса, как показано здесь.

Листинг 29.7 Как избежать внедрения SQL-кода, используя параметризацию

```
public IList<User> FindRecipe(string search)
{
    return _context.Recipes
        .FromSqlRaw( "SELECT * FROM Recipes WHERE Name = '{0}'",
            search )
        .ToList();
}
```

Параметризованные запросы не уязвимы для атак с использованием SQL-инъекций, поэтому представленная ранее атака не сработает. Если вы используете EF Core или другие ORM для доступа к данным с помощью стандартных запросов LINQ, вы будете неуязвимы для таких атак. EF Core автоматически создает все запросы SQL, используя параметризованные запросы, чтобы защитить вас. Даже если вы используете низкоуровневые API базы данных ADO.NET, придерживайтесь параметризованных запросов!

ПРИМЕЧАНИЕ Я говорил об атаках с использованием внедрения SQL-кода только с точки зрения реляционной базы данных, но эта уязвимость может проявляться и в NoSQL, и в документо-ориентированных базах данных. Всегда используйте параметризованные запросы (или их аналог) и не создавайте запросы, объединяя строки с пользовательским вводом.

Атаки с использованием внедрения SQL-кода были уязвимостью номер один в интернете на протяжении более десятка лет, поэтому очень важно знать о них и о том, как они возникают. Всякий раз, когда вам нужно писать низкоуровневые SQL-запросы, убедитесь, что вы всегда используете параметризованные запросы.

Следующая уязвимость также связана с доступом злоумышленников к данным, к которым они не должны иметь доступа. Эта атака несколько утонченная по сравнению с прямой атакой с использованием внедрения, но выполнить ее несложно: единственное, что нужно злоумышленнику, – умение считать.

29.4.3 Предотвращение небезопасных прямых ссылок на объекты

Небезопасная прямая ссылка на объект – звучит несколько занудно, но это означает, что пользователи получают доступ к вещам, к которым у них

не должно быть доступа, используя шаблоны в URL-адресах. Вернемся к примеру приложения с рецептами. Напомню, что приложение показывает вам список рецептов. Вы можете просмотреть любой из них, но вы можете редактировать только рецепты, которые создали сами. Когда вы просматриваете чужой рецепт, кнопка «Изменить» не отображается.

Например, пользователь нажимает эту кнопку на одном из своих рецептов и замечает URL-адрес: `/Recipes/Edit/120`. «120» – это неуместный идентификатор в базе данных редактируемого объекта. Можно было бы просто изменить этот идентификатор, чтобы получить доступ к другой сущности, к которой у пользователя обычно нет доступа. Пользователь может попробовать ввести `/Recipes/Edit/121`. Если это позволяет ему редактировать или просматривать рецепт, к которому он не должен иметь доступа, значит, у вас небезопасная прямая ссылка на объект.

Решить эту проблему просто: у вас должна быть авторизация и аутентификация на основе ресурсов в методах действий. Если пользователь пытается получить доступ к сущности, доступ к которой ему запрещен, то должен получить ошибку **Permission denied**. У него не должно быть возможности обойти вашу авторизацию, вводя URL-адрес непосредственно в строку поиска браузера.

В приложениях ASP.NET Core эта уязвимость обычно возникает при попытке ограничить действия пользователей, скрывая элементы из пользовательского интерфейса, например кнопку «Изменить». Вместо этого следует использовать авторизацию на основе ресурсов, как описано в главе 24.

ВНИМАНИЕ Вы всегда должны использовать авторизацию на основе ресурсов, чтобы ограничить сущности, к которым пользователь может получить доступ. Скрытие элементов пользовательского интерфейса улучшает опыт взаимодействия пользователя с сайтом, но это не мера безопасности.

Можно обойти эту уязвимость, избегая целочисленных идентификаторов сущностей в URL-адресах, например используя псевдослучайный GUID (допустим, `C2E296BA-7EA8-4195-9CA7-C323304CCD12`) вместо этого. Это усложняет процесс угадывания других сущностей, поскольку нельзя просто добавить единицу к существующему числу, но это всего лишь маскирует проблему, а не исправляет ее. Тем не менее использование GUID может быть полезно, когда вы хотите, чтобы у вас были общедоступные страницы (не требующие аутентификации), но вы не хотите, чтобы их идентификаторы можно было легко обнаружить.

В последнем разделе этой главы речь не идет об одной конкретной уязвимости. В ней мы обсудим отдельный, но связанный с этой темой вопрос: защиту данных пользователей.

29.4.4 Защита паролей и данных пользователей

Для многих приложений самыми конфиденциальными данными, которые вы будете хранить, являются личные данные ваших пользователей. Это может быть адрес электронной почты, пароли, адресные

данные или платежная информация. Будьте осторожны при хранении таких данных. Помимо того что вы предоставляете привлекательную цель для злоумышленников, у вас могут быть юридические обязательства в отношении того, как вы с ними обращаетесь, например законы о защите данных и требования соответствия PCI.

Самый простой способ защитить себя – не хранить ненужные данные. Если вам не нужен адрес вашего пользователя, не спрашивайте его. Таким образом, вы не сможете его потерять! Аналогично, если вы используете стороннюю службу идентификации для хранения сведений о пользователях, как описано в главе 14, вам не придется так усердно работать, чтобы защитить личные данные своих пользователей.

Если вы храните данные пользователя в собственном приложении или создаете собственного поставщика идентификационной информации, тогда вам нужно обязательно следовать лучшим практикам при работе с пользовательской информацией. Новые шаблоны проектов, использующие ASP.NET Core Identity, следуют большинству из этих практик по умолчанию, поэтому я настоятельно рекомендую начать с одного из них. Вам нужно рассмотреть много различных аспектов – их слишком много, чтобы подробно рассматривать их здесь¹, но они включают в себя следующее:

- никогда не храните пароли пользователей где-либо напрямую. Вы должны хранить только криптографические хеши, вычисленные с использованием сложного алгоритма хеширования, такого как BCrypt или PBKDF2;
- не храните больше данных, чем нужно. Никогда не храните данные кредитной карты;
- разрешите пользователям использовать двухфакторную аутентификацию (2FA) для входа на ваш сайт;
- не позволяйте пользователям использовать заведомо ненадежные или скомпрометированные пароли;
- пометьте cookie-файлы аутентификации как http (чтобы их нельзя было прочитать с помощью JavaScript) и «безопасно», чтобы они отправлялись только через HTTPS-соединение, а не через HTTP;
- не показывайте, зарегистрирован пользователь в вашем приложении или нет. Из-за утечки этой информации вы можете подвергнуться атакам перечислением.

СОВЕТ Вы можете узнать больше о перечислении веб-сайтов в этом видеоуроке Троя Ханта: <http://mng.bz/PAAA>.

Эти рекомендации представляют собой минимум, который вы должны делать для защиты своих пользователей. Самое главное – быть в курсе потенциальных проблем безопасности, когда вы создаете приложение. Пытаться использовать механизмы безопасности в конце

¹ В 2020 году Национальный институт стандартов и технологий (NIST) обновил свои рекомендации по цифровой идентификации, касающиеся обработки данных пользователей, которые содержат несколько полезных советов: <http://mng.bz/6gRA>.

всегда труднее, чем подумать об этом с самого начала, поэтому лучше сделать это раньше, чем позже.

Эта глава представляет собой краткий обзор вещей, на которые стоит обратить внимание. Мы коснулись большинства известных уязвимостей в системе безопасности, но я настоятельно рекомендую ознакомиться с другими ресурсами, упомянутыми в данной главе. Они предоставляют более исчерпывающий список вещей, которые следует учитывать, дополняя меры защиты, упомянутые в этой главе. Кроме того, не забывайте о проверке ввода и оверпостинге, о которых шла речь в главе 16. ASP.NET Core включает базовые средства защиты от некоторых наиболее распространенных атак, но вы все равно можете навредить себе. Убедитесь, что ваше приложение не попало в заголовки из-за того, что было взломано!

Резюме

- Межсайтовый скрипting используется злоумышленниками, чтобы внедрять содержимое в ваши приложения, как правило, для запуска вредоносного кода JavaScript, когда пользователи просматривают ваше приложение. Можно избежать подобных атак, если вы будете всегда защищать небезопасный ввод перед записью его на страницу. В Razor Pages это происходит автоматически, если вы не применяете метод `@Html.Raw()`, поэтому используйте его редко и осторожно;
- межсайтовая подделка запросов является проблемой для приложений, использующих проверку подлинности на основе файлов cookie, например ASP.NET Core Identity. Эти атаки полагаются на тот факт, что браузеры автоматически отправляют файлы cookie на веб-сайт. Вредоносный сайт может создать форму, которая отправляет запросы с помощью метода POST на ваш сайт, а браузер будет отправлять cookie-файлы аутентификации с запросом. Это позволяет вредоносным веб-сайтам отправлять запросы, как если бы это был пользователь, выполнивший вход;
- вы можете смягчить последствия таких атак, используя защитные токены. Сюда входит написание скрытого поля в каждой форме, которое содержит случайную строку на основе текущего пользователя. Аналогичный токен хранится в файле cookie. У допустимого запроса будут обе части, но поддельный запрос с вредоносного веб-сайта будет содержать только половину – файл cookie; он не может воссоздать скрытое поле в форме. Выполнив валидацию этих токенов, ваш API может отклонять поддельные запросы;
- фреймворк Razor Pages автоматически добавляет защитные токены в любые формы, которые вы создаете с помощью Razor, и проверяет токены для входящих запросов. При необходимости можно отключить проверку валидации, используя атрибут `[IgnoreAntiForgeryToken]`;
- браузеры не разрешают веб-сайтам выполнять запросы JavaScript AJAX от одного приложения к другому из разных источников.

Чтобы соответствовать источнику, у приложения должны быть такой же протокол, домен и порт. Если вы хотите делать подобные запросы из разных источников, то должны активировать совместное использование ресурсов между источниками (CORS) в своем API;

- CORS использует HTTP-заголовки для обмена данными с браузерами и определяет, какие источники могут вызывать ваш API. В ASP.NET Core можно определить несколько политик, которые могут применяться либо глобально ко всему приложению, либо к конкретным контроллерам и действиям;
- вы можете добавить промежуточное ПО CORS, вызвав метод `UseCors()` класса `WebApplication` и, при необходимости, указав имя применяемой политики CORS по умолчанию. Также можно применить CORS к методу действия или контроллеру веб-API, добавив атрибут `[EnableCors]` и указав имя применяемой политики;
- атаки с открытым перенаправлением используют распространенный механизм `returnURL` после выполнения входа для перенаправления пользователей на вредоносные веб-сайты. Вы можете предотвратить эту атаку, убедившись, что выполняете перенаправление только на локальные URL-адреса, принадлежащие вашему приложению;
- небезопасные прямые ссылки на объект – распространенная проблема, когда вы предоставляете идентификатор сущностей базы данных в URL-адресе. Всегда нужно проверять, есть ли у пользователей полномочия для доступа к запрашиваемому ресурсу или его изменения с помощью авторизации на основе ресурсов в своих методах действия;
- атаки с использованием внедрения SQL-кода являются распространенным вектором атак при создании SQL-запросов вручную. Всегда используйте параметризованные запросы или фреймворк, например EF Core, который неязвим для внедрения SQL-кода;
- самыми конфиденциальными данными в вашем приложении часто являются данные ваших пользователей. Обезопасьте себя, сохранивая только те данные, которые вам нужны. Убедитесь, что вы храните пароли лишь в виде хеша, защитите себя от слабых или скомпрометированных паролей и обеспечьте возможность двухфакторной аутентификации. ASP.NET Core Identity предоставляет все это уже в готовом виде, поэтому если вам нужно будет создать поставщика идентификационной информации, то это отличный выбор.

Часть V

Дальнейшая работа с ASP.NET Core

Части с I по IV затрагивают все аспекты ASP.NET Core, которые необходимо изучить для создания приложения с поддержкой протокола HTTP, будь то приложения, отображаемые на сервере с использованием Razor Pages или JSON API с использованием минимальных API. В части V мы рассмотрим четыре темы, основанные на уже полученных вами знаниях: настройка ASP.NET Core в соответствии с вашими потребностями, взаимодействие со сторонними API по протоколу HTTP, фоновые службы и тестирование.

Главу 30 мы начнем с рассмотрения альтернативного способа загрузки приложений ASP.NET Core, используя обобщенный хост вместо подхода с `WebApplication`, который был показан в этой книге. Обобщенный хост был стандартным способом загрузки приложений до появления .NET 6 (этот подход вы найдете в предыдущих изданиях данной книги), поэтому полезно распознавать этот паттерн, но он также пригодится для создания приложений, не поддерживающих протокол HTTP, как вы увидите в главе 34.

В части I вы узнали о конвейере промежуточного ПО и увидели, насколько он важен для всех приложений ASP.NET Core. В главе 31 вы узнаете, как в полной мере воспользоваться преимуществами конвейера, создавая ветвящиеся конвейеры промежуточного ПО, собственные компоненты и простые конечные точки на основе промежуточ-

ного ПО. Вы также узнаете, как решать сложные проблемы конфигурации вида «курица или яйцо», которые часто возникают в реальных приложениях. Наконец, вы узнаете, как заменить встроенный контейнер внедрения зависимостей сторонней альтернативой.

В главе 32 вы узнаете, как создавать собственные компоненты для работы с Razor Pages и контроллерами API. Вы узнаете, как создавать собственные тег-хелперы и атрибуты валидации, а также я представлю новый компонент – компоненты представления – для инкапсуляции логики с помощью отрисовки представления Razor. Вы также узнаете, как заменить фреймворк валидации на основе атрибутов, используемый по умолчанию в ASP.NET Core, на альтернативный.

Большинство приложений, которые вы создаете, не предназначены для автономной работы. Обычно приложению приходится взаимодействовать с API, будь то API для отправки электронных писем, приема платежей или взаимодействия с вашими собственными внутренними приложениями. В главе 33 вы узнаете, как вызывать эти API, используя абстракцию `IHttpClientFactory` для упрощения настройки, добавить обработку временных сбоев и избежать распространенных ошибок.

В этой книге основное внимание уделяется обслуживанию HTTP-трафика как веб-страниц с отрисовкой на стороне сервера с использованием Razor Pages, так и веб-API, обычно используемых мобильными и одностраничными приложениями. Однако многим приложениям требуются фоновые задачи с длительным временем выполнения, которые выполняют задания по расписанию или обрабатывают элементы из очереди. В главе 34 я покажу, как можно создавать фоновые задачи с длительным временем выполнения в приложениях ASP.NET Core. Я также покажу, как создавать автономные службы, выполняющие только фоновые задачи, без какой-либо обработки HTTP, и устанавливать их как службу Windows или как демон Linux, `systemd`.

Главы 35 и 36, последние главы, посвящены тестированию приложения. Точная роль тестирования в разработке приложений может привести к философским спорам, но в этих главах я сосредоточусь на практических вопросах тестирования приложения с помощью фреймворка тестирования `xUnit`.

Вы увидите, как создавать модульные тесты для приложений, тестировать код, зависящий от EF Core, с помощью поставщика базы данных в памяти, а также писать интеграционные тесты, которые могут тестировать несколько аспектов приложения одновременно.

В быстро меняющемся мире веб-разработки всегда есть чему поучиться, но к концу пятой части у вас должно быть все необходимое для создания приложений с помощью ASP.NET Core, будь то многостраничные приложения с отрисовкой на стороне сервера, API или фоновые службы.

В приложениях к этой книге приводится некоторая информация и ресурсы по теме .NET. В приложении А описывается, как подготовить окружение разработки, путем установки .NET 7 и IDE или редактора. В приложении Б вы найдете список ресурсов, которые я использую, чтобы узнать больше об ASP.NET Core и быть в курсе новейших функций.

30

Создание приложений ASP.NET Core с помощью обобщенного хоста и класса Startup

В этой главе:

- использование обобщенного хоста и класса Startup для загрузки приложения ASP.NET Core;
- чем обобщенный хост отличается от WebApplication;
- создание собственного обобщенного IHostBuilder;
- выбор между обобщенным хостом и минимальным хостингом.

Одними из крупнейших изменений, представленных в ASP.NET Core в .NET 6, были минимальные API, а именно типы `WebApplication` и `WebApplicationBuilder`, которые вы видели в этой книге. Они были добавлены, чтобы значительно сократить объем кода, необходимого для начала работы с ASP.NET Core, и теперь являются способом по умолчанию для создания приложений ASP.NET Core.

До появления .NET 6 в ASP.NET Core использовался другой подход для загрузки приложения: обобщенный хост, `IHost`, `IHostBuilder` и класс

`Startup`. Несмотря на то что этот подход не используется по умолчанию в .NET 7, он по-прежнему существует, поэтому важно, чтобы вы знали о нем, даже если вам не нужно использовать его самостоятельно. В этой главе я познакомлю вас с обобщенным хостом и покажу, как он связан с минимальными API, с которыми вы уже знакомы. В главе 34 вы также узнаете, как использовать обобщенный хост для создания приложений, работающих за пределами браузера.

Я начну с представления двух основных понятий: общих компонентов хоста (`IHostBuilder` и `IHost`) и класса `Startup`. Они разделяют код загрузки вашего приложения между двумя файлами: `Program.cs` и `Startup.cs`, обрабатывая различные аспекты конфигурации вашего приложения. Вы узнаете, почему было введено такое разделение, где настраивается каждый компонент и чем он отличается от минимального хостинга с использованием `WebApplication`.

В разделе 30.4 вы узнаете, как работает вспомогательная функция `Host.CreateDefaultBuilder()`, и используете эти знания для настройки экземпляра `IHostBuilder`. Это может дать больше контроля, чем минимальный хостинг, что может быть полезно в некоторых ситуациях.

В разделе 30.5 мы сделаем шаг назад и рассмотрим некоторые недостатки изученного нами общего кода начальной загрузки хоста, в частности его кажущуюся сложность по сравнению с минимальным хостингом с помощью `WebApplication`.

Наконец, в разделе 30.6 я обсуждаю некоторые причины, по которым вы тем не менее можете выбрать использование общего хоста вместо минимального хостинга в своем приложении .NET 7. В большинстве случаев я предлагаю использовать минимальный хостинг с `WebApplication`, но есть допустимые случаи, когда общий хост имеет смысл.

30.1 Разделение ответственности между двумя файлами

Как вы видели в этой книге, стандартный способ создания приложения ASP.NET Core в .NET 7 – это использование классов `WebApplicationBuilder` и `WebApplication` в файле `Program.cs` с применением операторов верхнего уровня. Однако до .NET 6 в ASP.NET Core использовался другой подход, который при желании вы все еще можете применять в .NET 7.

В этом подходе обычно используется традиционная точка входа `static void Main()` (хотя операторы верхнего уровня поддерживаются) и код начальной загрузки разбивается на два файла, как показано на рис. 30.1:

- *Program.cs* – содержит точку входа для приложения, которое загружает экземпляр `IHost`. Здесь вы настраиваете инфраструктуру своего приложения, такую как Kestrel, интеграцию со службами IIS и источники конфигурации;
- *Startup.cs* – класс `Startup` – это место, где вы настраиваете контейнер внедрения зависимостей, конвейер промежуточного ПО и конечные точки вашего приложения.

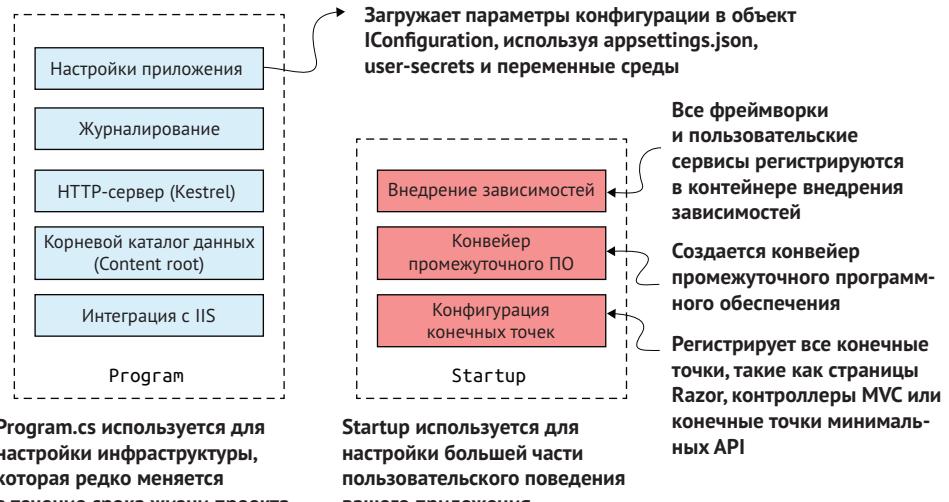


Рис. 30.1 Различные обязанности классов Program и Startup в приложении ASP.NET Core, в котором вместо WebApplication используется обобщенный хост

30.2 Класс Program: сборка веб-хоста

Все приложения ASP.NET по своей сути являются консольными приложениями. В модели размещения на основе класса `Startup` основная точка входа создает и запускает экземпляр `IHost`, как показано в следующем листинге, где используется типичный файл `Program.cs`. `IHost` – это ядро приложения ASP.NET Core: оно содержит HTTP-сервер (Kestrel) для обработки запросов, а также все необходимые службы и настройки для генерации ответов.

Листинг 30.1 Файл Program.cs настраивает и запускает экземпляр IHost

```
public class Program
{
    public static void Main(string[] args)
    {
        Создает и возвращает экземпляр IHost из IHostBuilder
        CreateHostBuilder(args)
            .Build()
            .Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}

Класс Startup определяет большую часть конфигурации

```

Annotations explaining the code flow:

- Создает и возвращает экземпляр IHost из IHostBuilder** (Creates an IHostBuilder instance and returns an IHost instance)
- Создает IHostBuilder с помощью метода CreateHostBuilder** (Creates an IHostBuilder using the CreateHostBuilder method)
- Запускает IHost и начинает прослушивать запросы и генерировать ответы** (Starts the IHost and begins listening for requests and generating responses)
- Создает IHostBuilder, используя конфигурацию по умолчанию** (Creates an IHostBuilder using default configuration)
- Настраивает приложение для использования Kestrel и прослушивания HTTP-запросов** (Configures the application to use Kestrel and listen for HTTP requests)

Функция `Main` содержит весь базовый код инициализации, необходимый для создания веб-сервера и начала прослушивания запросов. Она использует `IHostBuilder`, созданный вызовом `CreateDefaultBuilder`, чтобы определить, как должен быть настроен общий `IHost`, прежде чем создавать экземпляр `IHost` с помощью вызова `Build()`.

СОВЕТ Объект `Host` представляет созданное нами приложение. Тип `WebApplication`, который мы использовали на протяжении всей книги, также реализует `IHost`.

Большая часть настройки вашего приложения происходит в `IHostBuilder`, создаваемом вызовом метода `CreateDefaultBuilder`, при этом часть ответственности делегируется отдельному классу `Startup`. Класс `Startup`, указанный в обобщенном методе `UseStartup<>`, – это место, в котором вы настраиваете сервисы своего приложения и определяете конвейер промежуточного ПО.

ПРИМЕЧАНИЕ Код для сборки `IHostBuilder` вынесен во вспомогательный метод `CreateHostBuilder`. Название этого метода исторически важно, поскольку оно неявно использовалось такими инструментами, как Entity Framework Core (EF Core), о которых я говорю в разделе 30.5.

Вам, наверное, интересно, зачем нужны два класса для настройки: `Program` и `Startup`. Почему бы не включить всю настройку приложения в один из них? Идея состоит в том, чтобы отделить код, который часто меняется, от кода, который меняется редко.

Классы `Program` для двух разных приложений ASP.NET Core обычно будут похожи, а вот классы `Startup` часто будут существенно отличаться (хотя обычно они следуют схожей схеме, в чем вы скоро убедитесь). Вы увидите, что вам редко нужно изменять класс `Program`, по мере того как ваше приложение будет увеличиваться в размерах, тогда как класс `Startup` обычно обновляется всякий раз, когда вы добавляете дополнительные функции. Например, если вы добавите в проект новую зависимость NuGet, то вам зачастую потребуется обновить класс `Startup`, чтобы использовать ее.

В классе `Program` происходит большая часть настройки приложения, но в шаблонах по умолчанию это скрыто внутри метода `CreateDefaultBuilder`. `CreateDefaultBuilder` – это статический вспомогательный метод, упрощающий загрузку вашего приложения путем создания `IHostBuilder` с некой распространенной конфигурацией. Это похоже на то, как мы использовали `WebApplicationBuilder.CreateDefaultBuilder()` на протяжении всей книги.

ПРИМЕЧАНИЕ Вы можете создавать собственные экземпляры `HostBuilder`, если хотите изменить настройки по умолчанию и создать полностью собственный экземпляр `IHost`, как вы увидите в разделе 30.4. Он будет отличаться от `WebApplicationBuilder`, который всегда использует одни и те же значения по умолчанию.

Другой вспомогательный метод, применяемый по умолчанию, – это `ConfigureWebHostDefaults`. Он использует объект `WebHostBuilder`, чтобы настроить Kestrel для обработки HTTP-запросов.

Создание служб с помощью обобщенного хоста

Может показаться странным, что нужно вызывать методы `ConfigureWebHost-Defaults` и `CreateDefaultBuilder`, – разве нельзя использовать только один метод? Разве весь смысл ASP.NET Core – это не обработка HTTP-запросов?

И да, и нет! В ASP.NET Core 3.0 появилась концепция обобщенного хоста. Он позволяет использовать большую часть того же фреймворка, что используют приложения ASP.NET Core, для написания приложений, не поддерживающих протокол HTTP. Их можно запускать как консольные приложения или устанавливать как службы Windows (или как демоны `systemd` в Linux), например для выполнения фоновых задач или чтения из очередей сообщений.

Kestrel и веб-платформа ASP.NET Core строятся поверх функциональности обобщенного хоста, появившейся в ASP.NET Core 3.0. Для настройки типичного приложения ASP.NET Core вы настраиваете функции обобщенного хоста, общие для всех приложений, такие как конфигурация, журналирование и службы зависимостей. Для веб-приложений вы дополнительно настраиваете службы, такие как Kestrel, необходимые для обработки веб-запросов. В главе 34 вы увидите, как создавать приложения, используя обобщенный хост для выполнения запланированных задач и создания сервисов.

Даже в .NET 7 `WebApplication` и `WebApplicationBuilder` скрыто используют обобщенный хост. Вы можете прочитать больше об эволюции кода начальной загрузки ASP.NET Core и взаимосвязи между `IHost` и `WebApplication` в моем блоге по адресу: <http://mng.bz/gBBv>.

Как только настройка `IHostBuilder` будет завершена, вызов метода `Build` создаст экземпляр `IHost`, но приложение еще не начнет обрабатывать HTTP-запросы. Прослушивание сервером HTTP-запросов запускается после вызова `Run`. После этого ваше приложение будет находиться полностью в рабочем состоянии и сможет отвечать на запросы от браузера.

30.3 Класс Startup: настройка приложения

Как вы уже видели, класс `Program` отвечает за настройку большей части инфраструктуры вашего приложения, однако некоторая часть поведения приложения настраивается в классе `Startup`. Класс `Startup` отвечает за настройку двух основных аспектов приложения:

- *регистрация сервисов* – любые классы, от которых зависит ваше приложение для обеспечения функциональности, – как те, что используются фреймворком, так и те, что относятся к вашему приложению, – должны быть зарегистрированы, чтобы можно было корректно создать их экземпляры во время выполнения;

- промежуточное ПО и конечные точки – как ваше приложение обрабатывает запросы и отвечает на них.

Каждый из этих аспектов настраивается в отдельном методе в `Startup`: регистрация сервисов в методе `ConfigureServices`, а конфигурирование промежуточного ПО в методе `Configure`. Типичная структура этого класса показана в следующем листинге.

Листинг 30.2 Схема Startup.cs, показывающая, как настраивается каждый аспект

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services) <-- Настраиваем сервисы, зарегистрировав их в IServiceCollection
    {
        // Детали метода;
    }

    public void Configure(IApplicationBuilder app) <-- Настраиваем конвейер промежуточного ПО для обработки HTTP-запросов
    {
        // Детали метода;
    }
}
```

`IHostBuilder`, созданный в классе `Program`, вызывает метод `ConfigureServices`, а затем метод `Configure`, как показано на рис. 30.2. Каждый вызов настраивает отдельную часть вашего приложения, делая ее доступной для последующих вызовов методов. Все сервисы, зарегистрированные в методе `ConfigureServices`, доступны для метода `Configure`. Как только настройка будет завершена, создается `IHost` путем вызова метода `Build()` объекта `IHostBuilder`.

Что касается класса `Startup`, то здесь есть один интересный момент: он не реализует никаких интерфейсов. Для поиска и вызова методов с предопределенными именами `Configure` и `ConfigureServices` используется *отражение (reflection)*. Это делает класс более гибким и позволяет вам изменить сигнатуру метода `Configure` для внедрения любых служб, зарегистрированных вами в `ConfigureServices`, с помощью DI.

СОВЕТ Если вы не являетесь поклонником подхода гибкой рефлексии, вы можете реализовать интерфейс `IStartup` или наследовать от класса `StartupBase`, который предоставляет сигнатуры методов, показанные ранее в листинге 30.2. Если вы выберете этот подход, вы не сможете использовать внедрение зависимостей (*dependency injection, DI*) для внедрения сервисов в метод `Configuration()`.

В `ConfigureServices` вы добавляете все необходимые и пользовательские сервисы в контейнер DI точно так же, как вы это делаете с `WebApplicationBuilder.Services` в типичном приложении ASP.NET Core. В следующем листинге показано, как можно настроить все сервисы для приложения рецептов Razor Pages, которые вы видели в этой книге. В этом листинге также показано, как вы можете по-

лучить доступ к IConfiguration вашего приложения: путем внедрения в конструктор класса Startup. Вы увидите, как настроить конфигурацию вашего приложения, в разделе 30.4.

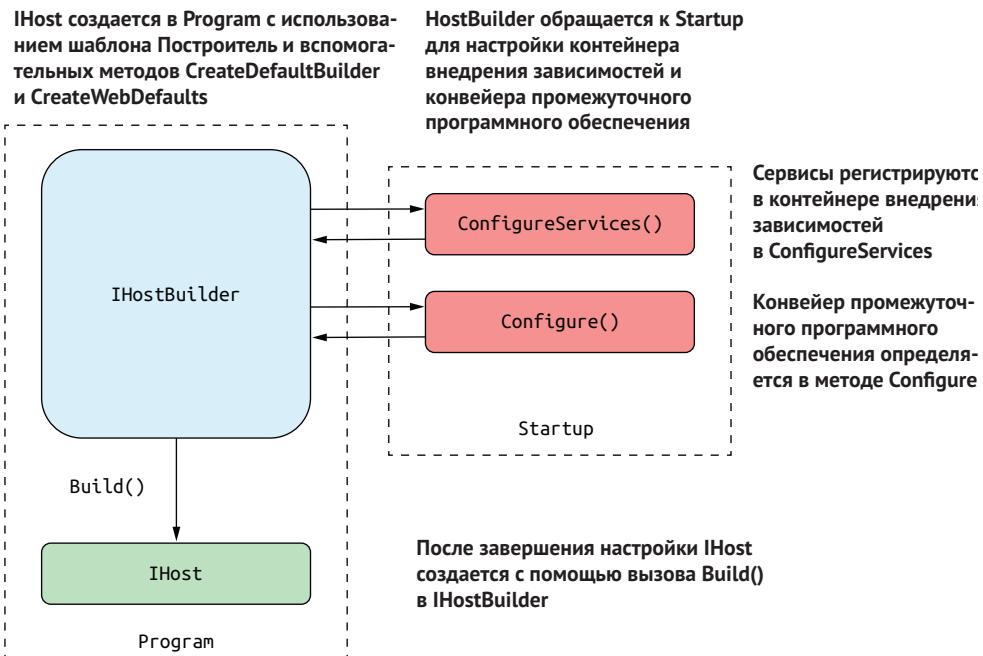


Рис. 30.2 IHostBuilder создается в файле Program.cs и вызывает методы класса Startup для настройки сервисов приложения и конвейера промежуточного ПО. Как только настройка будет завершена, создается IHost путем вызова метода Build() объекта IHostBuilder

Листинг 30.3 Регистрация сервисов с помощью внедрения зависимостей в ConfigureServices

```

public class Startup
{
    public IConfiguration Configuration { get; }
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    public void ConfigureServices(IServiceCollection services)
    {
        var conn = Configuration.GetConnectionString("DefaultConnection");
        services.AddDbContext<AppDbContext>(options =>
            options.UseSqlite(conn));
        services.AddDefaultIdentity<ApplicationUser>(options =>
            options.SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<AppDbContext>();
    }
}

```

Регистрирует все сервисы EF Core и ASP.NET Core Identity

IConfiguration для приложения внедряется в конструктор

Вы должны зарегистрировать свои сервисы в предоставленной коллекции IServiceCollection

**Регистрирует
сервисы
фреймворка**

```

services.AddScoped<RecipeService>(); ← Регистрирует реализации пользовательских сервисов
services.AddRazorPages();

services.AddScoped<IAuthorizationHandler, IsRecipeOwnerHandler>();
services.AddAuthorizationBuilder()
    .AddPolicy("CanManageRecipe",
        p => p.AddRequirements(new IsRecipeOwnerRequirement()));
}

public void Configure(IApplicationBuilder app) => { /* Not shown */ }
}

```

После настройки всех сервисов необходимо настроить конвейер промежуточного ПО и сопоставить конечные точки. Данный процесс аналогичен настройке конвейера промежуточного ПО с помощью `WebApplication`:

- мы добавляем промежуточное ПО в конвейер, вызывая методы расширения `Use*` для экземпляра `IApplicationBuilder`;
- порядок добавления промежуточного ПО в конвейер важен и определяет окончательный порядок конвейера;
- можно добавлять промежуточное ПО условно в зависимости от окружения.

Однако между подходом с `WebApplication`, который вы видели до сих пор, и подходом с `Startup` есть некоторые важные различия:

- `IWebHostEnvironment` для вашего приложения предоставляется непосредственно в `WebApplication.Environment`. Чтобы получить доступ к этой информации внутри `Startup`, вы должны внедрить ее в конструктор или метод конфигурации с помощью внедрения зависимостей;
- как вы видели в главе 4, `WebApplication` автоматически добавляет в ваш конвейер множество промежуточного ПО, такого как промежуточное ПО маршрутизации, промежуточное ПО конечных точек и промежуточное ПО аутентификации. При использовании подхода с `Startup` необходимо добавить это промежуточное ПО вручную;
- `WebApplication` реализует как `IApplicationBuilder`, так и `IEndpointRouteBuilder`, поэтому вы можете добавлять конечные точки непосредственно в `WebApplication`, например вызывая `MapGet()` или `MapRazorPages()`. При использовании подхода с `Startup` вы должны вызвать `UseEndpoints()` и сопоставить все конечные точки с помощью лямбда-метода;
- метод `Configure` не является асинхронным, поэтому выполнять асинхронные задачи затруднительно. Напротив, при использовании `WebApplication` вы можете использовать асинхронные методы между любым вашим общим кодом начальной загрузки.

Несмотря на эти предостережения, во многих случаях ваш метод `Startup.Configure` будет выглядеть почти идентично тому, как вы настраиваете конвейер в `WebApplication`. В следующем листинге показано, как может выглядеть метод `Configure()` для приложения рецептов Razor Pages.

Листинг 30.4 Метод Startup.Configure() для приложения Razor Pages

```

public class Startup
{
    public void Configure(
        IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}

```

Различное поведение в окружении разработки или промышленном окружении

Аналогично вы должны явно вызвать UseRouting

IApplicationBuilder используется для построения конвейера промежуточного программного обеспечения

Другие сервисы могут быть приняты в качестве параметров

WebApplication добавит это автоматически. Вы должны явно добавить его при использовании Startup

Всегда должен размещаться между вызовами UseRouting и UseEndpoints

Добавляет промежуточное программное обеспечение конечной точки, которое выполняет конечные точки

Сопоставляет конечные точки Razor Pages

В этом примере объект `IWebHostEnvironment` внедряется в метод `Configure()` с помощью внедрения зависимостей, чтобы можно было по-разному настроить конвейер промежуточного ПО в окружении разработки и промышленном окружении. В этом случае мы добавляем `DeveloperExceptionPageMiddleware` в конвейер во время разработки.

ПРИМЕЧАНИЕ. Помните, что `WebApplication` добавляет это промежуточное программное обеспечение автоматически, но при использовании `Startup` вы должны добавить его вручную. То же самое касается и всего остального автоматически добавляемого промежуточного ПО.

После добавления всего промежуточного программного обеспечения в конвейер вы переходите к вызову `UseEndpoints()`, который добавляет `EndpointMiddleware` в конвейер. Когда вы используете `WebApplication`, вам редко нужно вызывать его, так как `WebApplication` автоматически добавляет его в конец конвейера, но когда вы используете `Startup`, следует добавить его в конец конвейера.

Также обратите внимание, что при вызове `UseEndpoints()` вы определяете все конечные точки в вашем приложении. Будь то Razor Pages, контроллеры Model-View-Controller (MVC) или минимальные API, вы должны зарегистрировать их в лямбда-методе `UseEndpoints()`.

ПРИМЕЧАНИЕ. Конечные точки должны быть зарегистрированы внутри вызова `UseEndpoints()`, используя экземпляр `IEndpointRouteBuilder` из лямбда-метода.

Помимо отмеченных различий, перемещение вашей службы, промежуточного программного обеспечения и конфигурации конечной точки между подходом на основе `Startup` и `WebApplication` должно быть относительно простым, что может заставить вас задаться вопросом, есть ли веская причина выбирать подход с использованием `Startup` вместо `WebApplication`. Как всегда, ответ: «Все зависит от ситуации», но одна из возможных причин может заключаться в том, что в одном случае вы сможете настроить свой `IHostBuilder`.

30.4 Создание собственного экземпляра `IHostBuilder`

Как вы видели в разделе 30.2, стандартным способом работы с классом `Startup` в ASP.NET Core является использование метода `Host.CreateDefaultBuilder()`. Этот вспомогательный метод устанавливает множество значений по умолчанию для вашего приложения. В этом отношении он аналогичен методу `WebApplicationBuilder.CreateBuilder()`.

Однако вам не обязательно использовать метод `CreateDefaultBuilder` для создания экземпляра `IHostBuilder`: вы можете напрямую создать экземпляр `HostBuilder` и настроить его с нуля, если хотите. Однако прежде чем вы начнете это делать, стоит увидеть некоторые вещи, которые дает вам метод `CreateDefaultBuilder`, и то, для чего они используются. Затем вы можете рассмотреть возможность настройки экземпляра `HostBuilder` по умолчанию вместо создания полностью индивидуального экземпляра.

ПРИМЕЧАНИЕ. Вы можете использовать `Host.CreateDefaultBuilder()` в .NET 7, даже если вы не используете ASP.NET Core, установив пакет `Microsoft.Extensions.Hosting`. Вы узнаете, как создавать приложения, не обрабатывающие HTTP-запросы, используя обобщенный хост, в главе 34.

Значения по умолчанию, выбранные `CreateDefaultBuilder`, идеальны при первоначальной настройке приложения, но по мере его роста может возникнуть необходимость разбить его на части и повозиться с внутренним устройством. В следующем листинге показан приблизительный обзор `CreateDefaultBuilder`, чтобы вы могли увидеть, как создается `HostBuilder`. Он не является исчерпывающим и полным, но должен дать вам представление об объеме работы, которую метод `CreateDefaultBuilder` делает за вас!

Листинг 30.5 Метод Host.CreateDefaultBuilder

```

public static IHostBuilder CreateDefaultBuilder(string[] args)
{
    var builder = new HostBuilder()   ← Создает экземпляр HostBuilder
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureHostConfiguration(IConfigurationBuilder config =>
    {
        config.AddEnvironmentVariables("DOTNET_");
        config.AddCommandLine(args);
    })
    .ConfigureAppConfiguration((hostingContext, config) =>
    {
        IHostEnvironment env = hostingContext.HostingEnvironment;
        config
            .AddJsonFile("appsettings.json")
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json");
        if (env.IsDevelopment())
        {
            config.AddUserSecrets();
        }
        config
            .AddEnvironmentVariables()
            .AddCommandLine();
    })
    .ConfigureLogging((hostingContext, logging) =>
    {
        logging.AddConfiguration(
            hostingContext.Configuration.GetSection("Logging"));
        logging.AddConsole();
        logging.AddDebug();
        logging.AddEventSourceLogger();
        logging.AddEventLog();
    })
    .UseDefaultServiceProvider((context, options) =>
    {
        var isDevelopment = context.HostingEnvironment
            .IsDevelopment();
        options.ValidateScopes = isDevelopment;
        options.ValidateOnBuild = isDevelopment;
    });
}

return builder;   ← Возвращает HostBuilder для дальнейшей конфигурации вызовом дополнительных методов перед вызовом Build()
}

```

Корень содержимого определяет каталог, в котором можно найти файлы конфигурации

Настраивает параметры хостинга, такие как определение среды хостинга

Настраивает параметры приложения

Настраивает инфраструктуру журналирования

Настраивает контейнер DI, при необходимости включая настройки проверки

Возвращает HostBuilder для дальнейшей конфигурации вызовом дополнительных методов перед вызовом Build()

Первый метод, вызываемый HostBuilder, – UseContentRoot(). Он сообщает приложению, в каком каталоге оно может найти любую конфигурацию или файлы Razor, которые ему понадобятся позже. Обычно это папка, в которой запущено приложение, сюда же указывает вызов GetCurrentDirectory.

COBET Помните, что `ContentRoot` – это не то место, где вы храните статические файлы, к которым браузер может получить прямой доступ. Для этой цели предназначен `WebRoot`, обычно `wwwroot`.

Метод `ConfigureHostingConfiguration()` позволяет вашему приложению определить, в какой среде оно в данный момент работает. Фреймворк ищет переменные окружения, начинающиеся с «`DOTNET_`» (например, переменная `DOTNET_ENVIRONMENT`, о которой вы узнали в главе 10), а также аргументы командной строки, чтобы определить, запущено ли приложение в окружении разработки или промышленном окружении. Он применяется для заполнения объекта `IWebHostEnvironment`, который используется во всем вашем приложении.

В методе `ConfigureAppConfiguration()` вы настраиваете основной объект `IConfiguration` для вашего приложения, заполняя его, например, из файлов `appsettings.json`, переменных среды и пользовательских секретов. Компоновщик по умолчанию заполняет конфигурацию, используя все источники, показанные в листинге 30.5, что аналогично конфигурации, которую использует `WebApplicationBuilder`.

COBET Существует несколько важных различий в том, как объект `IConfiguration` создается – с использованием построителя по умолчанию или подхода, используемого `WebApplicationBuilder`. Вы можете прочитать об этих различиях в моем блоге: <http://mng.bz/e11V>.

Следующим после настройки приложения идет функция `ConfigureLogging()`. В `ConfigureLogging` вы указываете параметры ведения журнала и поставщиков журналирования для своего приложения, о которых вы узнали в главе 26. Помимо настройки `ILoggerProviders` по умолчанию, этот метод настраивает фильтрацию журналов, используя `IConfiguration`, подготовленный в `ConfigureAppConfiguration()`.

Последний вызов метода `UseDefaultServiceProvider`, показанный в листинге 30.5, настраивает ваше приложение на использование встроенного DI-контейнера. Он также устанавливает параметры `ValidateScopes` и `ValidateOnBuild` на основе текущего `HostingEnvironment`. Это гарантирует, что при запуске приложения в среде разработки контейнер автоматически проверяет наличие перехваченных зависимостей, о которых вы узнали в главе 9.

Как видите, `CreateDefaultBuilder` многое делает за вас. Во многих случаях эти значения по умолчанию – именно то, что вам нужно, но если это не так, конструктор по умолчанию не является обязательным. Вы можете вызвать `new HostBuilder()` и начать его настройку с нуля, но вам нужно будет настроить все, что делает `CreateHostBuilder`: ведение журнала, конфигурацию хостинга и конфигурацию поставщика сервисов, а также конфигурацию вашего приложения.

Листинг 30.6 Настройка HostBuilder по умолчанию

```

public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logBuilder => logBuilder.AddSeq())
            .ConfigureAppConfiguration((hostContext, config) =>
            {
                config.Sources.Clear();           HostBuilder предоставляет контекст хос-
                                                тинга и экземпляр ConfigurationBuilder
                config.AddJsonFile("appsettings.json"); <-- Добавляет поставщик конфигурации JSON,
                                                предоставляя имя файла конфигурации

                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
            });
}

```

Добавляет поставщи-ка ведения журнала Seq в кон-фигурацию

Очищает по-ставщиков, настроенных по умолчанию в Create-DefaultBuilder

HostBuilder предоставляет контекст хос-тинга и экземпляр ConfigurationBuilder

Новый HostBuilder создается в CreateDefaultBuilder() и выполняет все методы настройки, которые вы видели в листинге 30.5. Затем HostBuilder вызывает дополнительные методы ConfigureLogging() и ConfigureAppConfiguration(), добавленные в листинге 30.6. Вы можете вызвать любой другой метод конфигурации HostBuilder для дальнейшей настройки экземпляра перед вызовом Build().

ПРИМЕЧАНИЕ. Каждый вызов метода Configure*() в HostBuilder добавляет функцию настройки кода установки; эти вызовы не заменяют существующие вызовы Configure*(). Методы конфигурации выполняются в том же порядке, в котором они добавляются в HostBuilder, поэтому они выполняются после методов конфигурации CreateDefaultBuilder().

Одно из критических замечаний в адрес ранних приложений ASP.NET Core заключалось в том, что они были довольно сложны для понимания на начальном этапе, и, прочитав эту главу, вы вполне сможете понять, почему! В следующем разделе мы сравниваем подходы с обобщенным хостом и Startup с новым подходом с минимальным хостингом и обсуждаем, когда вы можете захотеть использовать один из них.

30.5 Сложность обобщенного хоста

До .NET 6 все приложения ASP.NET Core использовали подход с обобщенным хостом и Startup. Многим понравилась добавленная последовательная структура, но у нее также есть некоторые недостатки и сложности:

- 1 конфигурация разделена на два файла;
- 2 разделение Program.cs и Startup довольно произвольно;
- 3 обобщенный IHostBuilder предоставляет новичкам возможность принимать устаревшие решения;
- 4 конфигурацию на основе лямбда-методов может быть сложно применять;
- 5 соглашения Startup, основанные на шаблонах, могут быть трудными для понимания;
- 6 инструментарий исторически основан на определении метода CreateHostBuilder в Program.cs.

Я рассмотрю каждую из этих проблем по очереди, а затем расскажу, как WebApplication попытался улучшить ситуацию.

Пункты 1 и 2 в предыдущем списке касаются разделения Program.cs и Startup. Как вы видели в разделе 30.1, теоретически предполагается, что файл Program.cs определяет хост и редко изменяется, тогда как Startup определяет функции приложения (сервисы, промежуточное ПО и конечные точки). Это кажется разумным решением, но есть один неизбежный недостаток: вам нужно переключаться между как минимум двумя файлами, чтобы понять весь код начальной загрузки.

Кроме того, не обязательно придерживаться этих соглашений. Вы можете зарегистрировать сервисы в Program.cs, например вызвав HostBuilder.ConfigureServices(), или зарегистрировать промежуточное ПО с помощью WebHostBuilder.Configure(). Это относительно редкое явление, но не такое уж и неслыханное, что еще больше стирает границы между файлами.

Пункт 3 относится к тому факту, что вам необходимо вызывать метод ConfigureWebHostDefaults() (который использует IWebHostBuilder), чтобы настроить Kestrel и зарегистрировать свой класс Startup. Этот уровень косвенности (и введение другого типа строителя) является остатком решений, восходящих к ASP.NET Core 1.0. Для людей, знакомых с ASP.NET Core, этот шаблон – всего лишь одна из таких вещей, но он добавляет путаницы, когда вы новичок в этом.

ПРИМЕЧАНИЕ Подробное описание эволюции кода начальной загрузки ASP.NET Core см. в моем блоге: <http://mng.bz/pPPK>.

Аналогичным образом конфигурация на основе лямбда-методов, упомянутая в пункте 4, может быть сложной для понимания новичкам в ASP.NET Core. Если вы новичок в .NET, лямбда-методы – это дополнительная концепция, которую вам необходимо понять, прежде чем вы сможете понять основы кода. Кроме того, выполнение лямбда-выражений не обязательно происходит последовательно; HostBuilder по сути ставит лямбда-методы в очередь, чтобы они выполнялись в нужное время. Рассмотрим следующий фрагмент:

```
public static IhostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging => logging.AddSeq())
        .ConfigureAppConfiguration(config => {})
```

```
.ConfigureServices(s => {})
.ConfigureHostConfiguration(config => {})
.ConfigureWebHostDefaults(webBuilder =>
{
    webBuilder.UseStartup<Startup>();
});
```

Лямбда-методы выполняются в следующем порядке:

- 1 ConfigureWebHostDefaults()
- 2 ConfigureHostConfiguration()
- 3 ConfigureAppConfiguration()
- 4 ConfigureLogging()
- 5 ConfigureServices()
- 6 Startup.ConfigureServices()
- 7 Startup.Configure()

По большей части эти детали не должны иметь значения, но они все же добавляют очевидную сложность для тех, кто плохо знаком с ASP.NET Core.

Пункт 5 в списке проблем относится к классу `Startup` и стандартному подходу, основанному на соглашениях/шаблонах. Пользователи, впервые использующие ASP.NET Core, вероятно, будут знакомы с интерфейсами и базовыми классами, но, возможно, они незнакомы с подходом, основанным на отражении.

Использование соглашений вместо явного интерфейса повышает гибкость, но может усложнить обнаружение. Также следует учитывать различные предостережения и крайние случаи. Например, в конструктор `Startup` можно внедрить только `IWebHostEnvironment` и `IConfiguration`; вы не можете ничего внедрить в метод `ConfigureServices()`, но вы можете внедрить любую зарегистрированную службу в `Configure()`. Это подразумеваемые правила, которые вы обнаруживаете в первую очередь, нарушая их, а затем приложение ругается на это!

ПРИМЕЧАНИЕ Подход на основе шаблонов позволяет использовать в `Configure` гораздо больше, чем просто внедрение зависимостей. Вы также можете создавать методы, специфичные для среды, такие как `ConfigureDevelopmentServices` или `ConfigureProductionServices`, а ASP.NET Core вызывает правильный метод в зависимости от среды. Если хотите, вы даже можете создать целый класс `StartupProduction!` Более подробную информацию об этих соглашениях при запуске см. в документации по адресу <http://mng.bz/Oxxw>.

Класс `Startup` – не единственное место, где ASP.NET Core использует непрозрачные соглашения. Возможно, вы помните, в разделе 30.2 я упоминал, что `Program.cs` намеренно извлекает сборку `IHostBuilder` в метод `CreateHostBuilder`. Название этого метода имело исторически важное значение. К нему подключены такие инструменты, как EF Core, чтобы они могли загружать конфигурацию и службы вашего приложения при выполнении миграции и других функций. В более

ранних версиях ASP.NET Core переименование этого метода привело бы к поломке всех ваших инструментов!

ПРИМЕЧАНИЕ Начиная с .NET 6 вам не нужно создавать метод `CreateHostBuilder`; вы можете создать все свое приложение внутри функции `Main` (или с помощью операторов верхнего уровня), и инструменты EF Core будут работать без ошибок. Это было частично исправлено для добавления поддержки `WebApplication`. Если вас интересует механизм исправления, посетите мой блог <http://mng.bz/Y11z>.

Как только вы освоите ASP.NET Core, большинство этих проблем станут относительно незначительными. Вы быстро привыкаете к стандартным шаблонам и избегаете подводных камней. Но для новых пользователей ASP.NET Core Microsoft хотела обеспечить более приятный опыт, намного ближе к тому, который вы получаете во многих других языках.

API минимального хостинга, предоставляемые `WebApplicationBuilder` и `WebApplication`, в значительной степени решают эти проблемы. Конфигурация происходит в одном файле с использованием императивного стиля с гораздо меньшим количеством методов настройки на основе лямбда-методов или неявной настройки на основе соглашений. Все соответствующие объекты, такие как конфигурация и среда, доступны как свойства типов `WebApplicationBuilder` или `WebApplication`, чтобы было легко получить к ним доступ.

`WebApplicationBuilder` и `WebApplication` также пытаются скрыть от вас большую часть сложностей и устаревших решений. Внутри `WebApplication` используется общий хост, но вам не обязательно знать об этом, чтобы использовать его или работать продуктивно. Как вы видели на протяжении всей книги, `WebApplication` автоматически добавляет в ваш конвейер различное промежуточное программное обеспечение, помогая вам избежать распространенных ошибок, таких как неправильный порядок промежуточного программного обеспечения.

ПРИМЕЧАНИЕ. Если вас интересует, как `WebApplicationBuilder` абстрагируется от обобщенного хоста, см. мой пост на странице <http://mng.bz/GuyD>.

В большинстве случаев минимальный хостинг упрощает загрузку обобщенного хоста и `Startup`, и Microsoft считает его современным способом создания приложений ASP.NET Core. Но есть случаи, когда вместо этого вы можете рассмотреть возможность использования обобщенного хоста.

30.6 Выбор между обобщенным хостом и минимальным хостингом

Введение `WebApplication` и `WebApplicationBuilder` в .NET 6, также известный как минимальный хостинг, было призвано значительно упростить

начало работы для новичков в .NET и ASP.NET Core. Все встроенные шаблоны ASP.NET Core сейчас используют минимальный хостинг, и в большинстве случаев нет смысла оглядываться назад.

В этом разделе я обсуждаю некоторые случаи, в которых вы все равно можете использовать подход обобщенного хоста. В трех основных случаях вы, скорее всего, захотите использовать обобщенный хост вместо минимального хостинга с `WebApplication`:

- если у вас уже есть приложение ASP.NET Core, использующее обобщенный хост;
- когда вам нужен (или вы хотите) точный контроль над созданием объекта `IHost`;
- когда вы создаете приложение, не использующее протокол HTTP.

Первый вариант использования относительно очевиден: если у вас уже есть приложение ASP.NET Core, которое применяет обобщенный хост и `Startup`, вам не нужно его менять. Вы по-прежнему можете обновить свое приложение до .NET 7, и вам не нужно будет менять какой-либо код запуска. Обобщенный хост и `Startup` полностью поддерживаются в .NET 7, но они не используются по умолчанию.

СОВЕТ Во многих случаях обновление существующего проекта до .NET 7 просто требует обновления платформы в файле `.csproj` и обновления некоторых пакетов NuGet. Если вам не повезет, вы можете обнаружить, что некоторые API изменились. Microsoft публикует руководства по обновлению для каждой основной версии, поэтому перед обновлением приложений стоит прочитать их: <http://mng.bz/zXX1>.

Если вы создаете новое приложение, но по какой-то причине вам не нравятся параметры по умолчанию, используемые `WebApplicationBuilder`, лучшим вариантом может быть применение обобщенного хоста. Обычно я бы не советовал этот подход, поскольку он, вероятно, потребует большего обслуживания, чем использование `WebApplication`, но он дает вам полный контроль над вашим кодом запуска, если он вам нужен или вы этого хотите.

Последний случай применяется, когда вы создаете приложение ASP.NET Core, которое в основном запускает службы фоновой обработки, например обрабатывая сообщения из очереди, но не обрабатывает HTTP-запросы. `WebApplication` и `WebApplicationBuilder`, как следует из их названий, ориентированы на создание веб-приложений, поэтому в данной ситуации они не имеют смысла.

ПРИМЕЧАНИЕ В главе 34 вы узнаете, как создавать фоновые задачи и службы с использованием обобщенного хоста. .NET 8 представляет не-HTTP-версию `WebApplicationBuilder`, называемую `HostApplicationBuilder`, цель которой – упростить загрузку приложений для ваших фоновых служб.

Если вы не находитесь ни в одной из этих ситуаций, настоятельно рассмотрите возможность использования подхода с минимальным хостингом веб-приложений и императивной, похожей на сценарий, начальной загрузкой операторов верхнего уровня.

ПРИМЕЧАНИЕ Тот факт, что вы используете `WebApplication`, не означает, что вам нужнобросить всю конфигурацию вашего сервиса и промежуточного программного обеспечения в `Program.cs`. Альтернативные подходы, такие как использование класса `Startup`, который вы вызываете вручную, или локальных функций для разделения вашей конфигурации, см. в моем блоге по адресу <http://mng.bz/OKKJ>.

В этой главе я представил относительно быстрый обзор обобщенного хоста и подхода, основанного на `Startup`. Если вы подумываете о переходе с обобщенного хостинга на минимальный хостинг или если вы знакомы с минимальным хостингом, но вам нужно работать с обобщенным хостингом, вы можете найти эквивалентную функцию в другой модели хостинга. Документация по переходу с .NET 5 на .NET 6 содержит хорошее описание различий между двумя моделями и того, как изменилась каждая отдельная функция. Вы можете найти его по адресу <http://mng.bz/KeeX>.

СОВЕТ В качестве альтернативы у Дэвида Фаулера из команды .NET есть аналогичная шпаргалка, описывающая миграцию. См. <http://mng.bz/9DDj>.

Независимо от того, решите ли вы использовать обобщенный хост или минимальный хостинг, будут использоваться все те же основные концепции ASP.NET: конфигурация, промежуточное программное обеспечение и внедрение зависимостей. В следующей главе вы узнаете о некоторых более продвинутых способах использования каждой из этих концепций, таких как создание ветвящихся конвейеров промежуточного программного обеспечения и пользовательских контейнеров внедрения зависимостей.

Резюме

- До .NET 6 приложения ASP.NET Core разделяли конфигурацию между двумя файлами: `Program.cs` и `Startup.cs`. `Program.cs` содержит точку входа для приложения и используется для настройки и создания объекта `IHost`. При запуске вы настраиваете контейнер внедрения зависимостей, конвейер промежуточного программного обеспечения и конечные точки вашего приложения;
- класс `Program` обычно содержит метод `CreateHostBuilder()`, который создает экземпляр `IHostBuilder`. Точка входа `Main` вызывает `CreateHostBuilder()`, вызывает `IHostBuilder.Build()` для создания экземпляра `IHost` и, наконец, запускает приложение, вызывая `IHost.Run()`;

- вы можете создать `IHostBuilder`, вызвав `Host.CreateDefaultBuilder()`. Это создает экземпляр `HostBuilder`, используя конфигурацию по умолчанию, аналогичную конфигурации, применяемой при вызове `WebApplication.CreateBuilder()`. По умолчанию `HostBuilder` использует поставщиков журналов и конфигурации по умолчанию, настраивает среду хостинга на основе переменных среды и аргументов командной строки и настраивает параметры контейнера внедрения зависимостей;
- приложения ASP.NET Core, использующие обобщенный хост, обычно вызывают `ConfigureWebHostDefaults()` в `HostBuilder`, представляющий лямбда-метод, который вызывает `UseStartup<Startup>()` экземпляра `IWebHostBuilder`. Это сообщает `HostBuilder` настроить контейнер внедрения зависимостей и конвейер промежуточного программного обеспечения на основе класса `Startup`;
- используйте класс `Startup` для регистрации сервисов с помощью внедрения зависимостей, настройте конвейер промежуточного ПО и зарегистрируйте свои конечные точки. Это обычный класс, поскольку он не обязательно реализует интерфейс или базовый класс. Вместо этого `IHostBuilder` ищет методы с конкретными названиями для вызова с использованием отражения;
- зарегистрируйте свои сервисы внедрения зависимостей в методе `ConfigureServices(IServiceCollection)` класса `Startup`. Вы регистрируете сервисы, используя те же методы `Add*`, которые применяете для регистрации сервисов в `WebApplicationBuilder.Services` при использовании минимального хостинга;
- если вам нужен доступ к `IConfiguration` или `IWebHostEnvironment` вашего приложения (доступны через `Configuration` и `Environment` соответственно в `WebApplicationBuilder`), вы можете внедрить их в свой конструктор `Startup`. Вы не можете внедрять какие-либо другие сервисы в конструктор `Startup`;
- настройте конвейер промежуточного программного обеспечения в `Startup.Configure(IApplicationBuilder)`. Используйте те же методы `Use*`, которые вы используете с `WebApplication`, для добавления промежуточного программного обеспечения в конвейер. Что касается `WebApplication`, порядок добавления промежуточного программного обеспечения определяет их порядок в конвейере;
- `WebApplication` автоматически добавляет промежуточное программное обеспечение, например промежуточное программное обеспечение маршрутизации и промежуточное программное обеспечение конечной точки в конвейер, когда вы используете минимальный хостинг. При использовании `Startup` вы должны самостоятельно добавить это промежуточное программное обеспечение;
- чтобы зарегистрировать конечные точки, вызовите `UseEndpoints(endpoints => {})` и вызовите соответствующую функцию `Map*` для предоставленного `IEndpointRouteBuilder` в лямбда-функции. Это существенно отличается от минимального хостинга, на котором можно вызвать `Map*` непосредственно в экземпляре `WebApplication`;

- вы можете настроить экземпляр `IHostBuilder`, добавив вызов методов конфигурации, например `ConfigureLogging()` или `ConfigureAppConfiguration()`. Эти методы выполняются после любых предыдущих вызовов, добавляя дополнительные уровни конфигурации в экземпляре `IHostBuilder`;
- обобщенный хост является гибким, но довольно сложен из-за стиля отложенного выполнения, широкого использования лямбда-методов и интенсивного использования соглашений. Минимальный хостинг был направлен на упрощение кода начальной загрузки, чтобы сделать его более императивным, уменьшая большую часть косвенности. Минимальный хостинг использует больше значений по умолчанию, но, как правило, с ним легче работать новичкам в ASP.NET Core;
- если у вас уже есть приложение ASP.NET Core, использующее `Startup` и обобщенный хост, нет необходимости переключаться на применение `WebApplication` и минимального хостинга; обобщенный хост полностью поддерживается в .NET 7. Кроме того, если вы создаете приложение, не использующее HTTP, обобщенный хост в настоящее время является лучшим вариантом;
- если вы создаете новое приложение ASP.NET Core, минимальный хостинг, скорее всего, обеспечит более приятный опыт. Обычно вам следует отдавать предпочтение этому обобщенному хосту для новых приложений, если только вам не требуется точный контроль над конфигурацией `IHostBuilder`.

31

Расширенная настройка

ASP.NET Core

В этой главе:

- создание собственного промежуточного ПО;
- использование служб внедрения зависимостей в конфигурации `IOptions`;
- замена встроенного контейнера внедрения зависимостей на сторонний контейнер.

Когда вы создаете приложения с помощью ASP.NET Core, большая часть вашей креативности и специализации уходит на сервисы и модели, составляющие бизнес-логику, а также на страницы Razor и API, которые их предоставляют. Однако в конечном итоге вы, скорее всего, обнаружите, что не можете полностью реализовать желаемую функцию, используя компоненты, которые поставляются «из коробки». На этом этапе вам, возможно, придется рассмотреть более сложные варианты использования встроенной функциональности.

В этой главе показаны способы настройки сквозных частей приложения, среди которых – контейнер внедрения зависимостей или конвейер промежуточного ПО. Эти подходы особенно полезны, если вы используете устаревшее приложение или работаете над существующим проектом и хотите продолжать использовать шаблоны и библиотеки, с которыми вы знакомы.

Начнем с рассмотрения конвейера промежуточного программного обеспечения. Вы увидели, как создавать конвейер путем объединения

существующего промежуточного программного обеспечения, в главе 4, но в этой главе вы создадите ваше собственное промежуточное программное обеспечение. Вы изучите базовые конструкции промежуточного программного обеспечения – методы `Map`, `Use` и `Run`, а также узнаете, как создавать автономные классы промежуточного программного обеспечения. Вы будете использовать их для создания компонентов промежуточного программного обеспечения, которые смогут добавлять заголовки ко всем вашим ответам, так же, как промежуточное программное обеспечение, возвращающее ответы. Наконец, вы узнаете, как превратить собственное промежуточное ПО в простую конечную точку с помощью маршрутизации конечных точек.

В главе 10 вы узнали о строго типизированной конфигурации с помощью `IOptions<T>`, а в разделе 31.2 вы углубите свои знания. Вы узнаете, как использовать тип `OptionsBuilder<T>` для быстрого создания объекта `IOptions<T>` с помощью шаблона компоновщика. Вы также увидите, как использовать сервисы внедрения зависимостей при настройке объектов `IOptions` – что невозможно с помощью методов, которые вы видели до сих пор.

Мы остановимся на внедрении зависимостей в разделе 31.3, где я покажу вам, как заменить встроенный контейнер зависимостей сторонней альтернативой. Встроенный контейнер подходит для большинства небольших приложений, но ваша функция `ConfigureServices` может быстро раздуться по мере роста вашего приложения и регистрации большего количества сервисов. Я покажу вам, как интегрировать стороннюю библиотеку Lamar в существующее приложение, чтобы вы могли использовать дополнительные функции, такие как автоматическая регистрация сервиса по соглашению.

Компоненты и методы, показанные в этой главе, являются более продвинутыми, чем большинство функций, которые вы видели до сих пор. Скорее всего, они не понадобятся вам в каждом проекте ASP.NET Core, но их полезно иметь под рукой, если возникнет такая необходимость!

31.1 Настройка конвейера промежуточного ПО

В этом разделе вы узнаете, как создать собственное промежуточное ПО. Вы научитесь использовать методы расширения `Map`, `Run` и `Use` для создания простого промежуточного ПО с использованием лямбда-выражений. Затем вы увидите, как создать эквивалентные компоненты промежуточного программного обеспечения, используя специализированные классы. Вы также узнаете, как разделить конвейер промежуточного ПО на ветви и когда это может пригодиться.

Конвейер промежуточного ПО – один из основных строительных блоков приложений ASP.NET Core, поэтому мы подробно рассмотрели его в главе 4. Каждый запрос проходит через этот конвейер, и каждый компонент, в свою очередь, получает возможность изменить запрос или обработать его и вернуть ответ. ASP.NET Core включает готовое промежуточное ПО для обработки распространенных сценариев. Вы найдете промежуточное ПО для обслуживания статических файлов, обработки ошибок, аутентификации и многого другого.

Во время разработки большую часть времени вы будете проводить, работая с Razor Pages, конечными точками минимальных API и контроллерами веб-API. Они представляются как конечные точки для большей части бизнес-логики вашего приложения и используют методы различных сервисов и модели бизнес-логики вашего приложения. Однако вы также видели промежуточное программное обеспечение, такое как Swagger и WelcomePageMiddleware, которое возвращает ответ без использования системы маршрутизации конечных точек. Благодаря различным улучшениям системы маршрутизации в .NET 7 я редко сталкиваюсь с необходимостью создавать такое «терминальное» промежуточное программное обеспечение, поскольку с маршрутизацией конечных точек легко работать и она расширяема. Тем не менее иногда может быть предпочтительнее создавать небольшие пользовательские терминальные компоненты промежуточного программного обеспечения, подобные этим.

В других случаях у вас могут быть требования, выходящие за рамки компетенции Razor Pages или конечных точек минимальных API. Например, у вас может возникнуть желание убедиться, что все ответы, сгенерированные вашим приложением, включают в себя определенный заголовок. Такого рода сквозная задача идеально подходит для специального промежуточного ПО. Вы можете добавить собственное промежуточное ПО в начало конвейера, чтобы гарантировать, что каждый ответ от вашего приложения включает в себя необходимый заголовок, независимо от того, исходит он от компонента статических файлов, обработки ошибок или страницы Razor.

В этом разделе я также покажу три способа создания собственных компонентов промежуточного ПО и как создать ветви в конвейере, когда запрос может перетекать то в одну ветвь, то в другую. Сочетая методы, продемонстрированные в этом разделе, вы сможете создавать индивидуальные решения для удовлетворения конкретных требований.

Мы начнем с создания компонента промежуточного ПО, который возвращает текущее время в виде обычного текста, когда приложение получает запрос. После этого мы рассмотрим ветвление конвейера, создание компонентов общего назначения и, наконец, увидим, как инкапсулировать промежуточное ПО в отдельные классы. В разделе 31.5 вы увидите альтернативный подход к предоставлению промежуточного программного обеспечения, генерирующего ответы, с использованием маршрутизации конечных точек.

31.1.1 Создание простых приложений с помощью метода расширения Run

Как вы видели в предыдущих главах, вы определяете конвейер промежуточного программного обеспечения для своего приложения в Program.cs, добавляя промежуточное программное обеспечение к объекту WebApplication, обычно используя методы расширения, как в этом примере:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
app.UseExceptionHandler();
```

```
app.UseStaticFiles();
app.Run();
```

Когда ваше приложение получает запрос, он проходит через компоненты, каждый из которых получает возможность изменить запрос или обработать его, сгенерировав ответ. Если компонент генерирует ответ, то, по сути, прерывает выполнение конвейера; последующие компоненты конвейера не увидят запрос. Ответ проходит обратно через более ранние компоненты на пути в браузер.

Для создания простого компонента, который всегда генерирует ответ, можно использовать метод расширения `Run`. Этот метод принимает одну лямбда-функцию, которая выполняется всякий раз, когда запрос достигает компонента. Метод `Run` всегда генерирует ответ, поэтому промежуточное ПО, размещенное после него, никогда не будет выполнено. По этой причине всегда следует размещать данный компонент в конвейере последним.

СОВЕТ Помните: компоненты промежуточного ПО выполняются в том порядке, в котором вы добавляете их в конвейер. Если компонент обрабатывает запрос и генерирует ответ, те компоненты, которые идут после него, не увидят запрос.

Метод расширения `Run` предоставляет доступ к запросу в виде объекта `HttpContext`, который вы видели в главе 4. Он содержит все детали запроса в свойстве `Request`, такие как путь URL-адреса, заголовки и тело запроса, а также свойство `Response`, которое можно использовать для возврата ответа.

В следующем листинге показано, как создать простой компонент промежуточного ПО, который возвращает текущее время. Он использует предоставленный объект `HttpContext` и свойство `Response`, чтобы задать заголовок ответа `Content-Type` (в нашем случае это не является строго необходимым условием, поскольку если альтернативный заголовок не задан, то используется заголовок `text/plain`), и записывает тело ответа с помощью функции `WriteAsync(text)`.

Листинг 31.1 Создание простого компонента промежуточного ПО с использованием метода расширения `Run`

```
Использует метод расширения Run для создания простого промежуточного ПО, которое всегда возвращает ответ
app.Run(async (HttpContext context) => <-----  

{  

    context.Response.ContentType = "text/plain"; <-----|  

    await context.Response.WriteAsync( |  

        DateTimeOffset.UtcNow.ToString()); |  

}); |  

app.UseStaticFiles(); <-----|  

    |  

    | Любое промежу-  

    | точное ПО, до-  

    | бавленное после  

    | метода расшире-  

    | ния Run, не будет  

    | выполнено |  

    |  

    | Вы должны установить  

    | значение заголовка  

    | Content-Type для гене-  

    | рируемого вами ответа;  

    | text/plain – значение  

    | по умолчанию |  

    |  

    | Возвращает время в виде  

    | строки в ответе. Код состояния  

    | 200 OK используется, если не  

    | установлен явно |
```

Метод расширения `Run` полезен для двух разных целей:

- создание простого промежуточного программного обеспечения, которое всегда генерирует ответ;
- создание сложного промежуточного программного обеспечения, которое перехватывает весь запрос для создания дополнительного фреймворка поверх ASP.NET Core.

Независимо от того, используете ли вы расширение `Run` для создания базовых конечных точек или сложного дополнительного уровня фреймворка, промежуточное программное обеспечение всегда генерирует какой-то ответ. Поэтому вы всегда должны размещать его в конце конвейера, так как промежуточное программное обеспечение, размещенное после него, не выполнится.

СОВЕТ Использование расширения «`Run`» для безусловной генерации ответа сейчас встречается редко. Система маршрутизации конечных точек, используемая минимальными API, предоставляет множество дополнительных возможностей, таких как привязка модели, маршрутизация, интеграция с другим промежуточным программным обеспечением, таким как аутентификация и авторизация и т. д.

Могут возникнуть редкие ситуации, когда вы хотите безоговорочно сгенерировать ответ, но более распространенный сценарий – это когда вы хотите, чтобы ваш компонент промежуточного программного обеспечения реагировал только на определенный путь URL-адреса, например как промежуточное программное обеспечение Swagger UI реагирует только на путь `/swagger`. В следующем разделе вы увидите, как можно объединить `Run` с методом расширения `Map` для создания ветвящихся конвейеров промежуточного программного обеспечения.

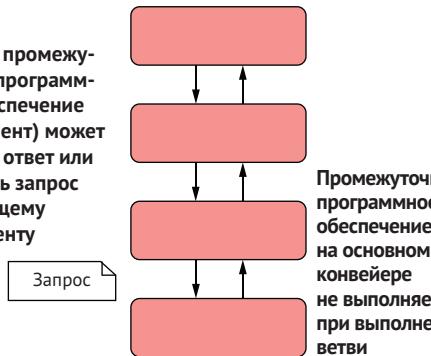
31.1.2 Ветвление конвейера с помощью метода расширения `Map`

До сих пор, обсуждая конвейер промежуточного ПО, мы всегда рассматривали его как единый конвейер, состоящий из последовательных компонентов. Каждый запрос проходит через все компоненты, пока один из них не сгенерирует ответ, который проходит обратно через предыдущие компоненты.

Метод расширения `Map` позволяет преобразовать этот простой конвейер в ветвящуюся структуру. Каждая ветвь конвейера независима; запрос проходит то через одну ветвь, то через другую, но не через обе, как показано на рис. 31.1. Метод расширения `Map` смотрит на путь URL-адреса запроса. Если путь соответствует требуемому шаблону, то запрос перемещается по ветви конвейера; в противном случае он остается в основном конвейере, что обеспечивает совершенно разное поведение в разных ветвях конвейера.

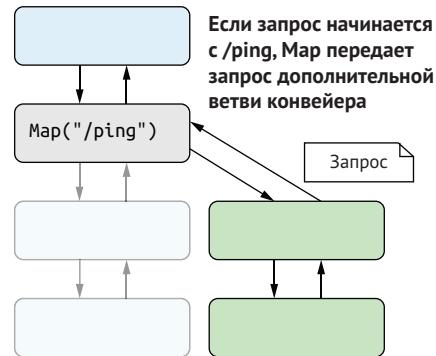
Как правило, конвейеры промежуточного программного обеспечения состоят из последовательных компонентов

Каждое промежуточное программное обеспечение (компонент) может вернуть ответ или передать запрос следующему компоненту



Традиционный последовательный конвейер промежуточного программного обеспечения

Промежуточное программное обеспечение Map() может использоваться для создания разветвленных конвейеров



Разветвляющийся конвейер промежуточного программного обеспечения

Рис. 31.1 Сравнение последовательного конвейера промежуточного ПО и конвейера ветвления, созданного с помощью метода расширения Map. В конвейере ветвления запросы проходят только через одну из ветвей. Компонент другой ветки не видит запрос и не выполняется

ПРИМЕЧАНИЕ Сопоставление URL-адресов, используемое методом Map, концептуально аналогично маршрутизации, как было показано начиная с главы 6, но оно гораздо проще и имеет много ограничений. Например, здесь используется простое сопоставление строки и префикса и нельзя использовать параметры маршрута. Как правило, следует отдавать предпочтение созданию конечных точек, а не ветвлению с использованием метода Map.

Например, представьте, что вы хотите добавить простую конечную точку проверки работоспособности в существующее приложение. Эта конечная точка представляет собой простой URL-адрес, который вы можете вызвать и который указывает, правильно ли работает ваше приложение. Вы можете легко создать компонент для проверки работоспособности, используя метод расширения Run, как было показано в листинге 31.1, но тогда это будет все, что сможет сделать ваше приложение. Вы хотите, чтобы проверка работоспособности отвечала только на определенный URL-адрес, /ping. Razor Pages должен обрабатывать все остальные запросы в обычном режиме.

СОВЕТ Сценарий с проверкой работоспособности – простой пример демонстрации работы метода Map, но ASP.NET Core включает в себя встроенную поддержку конечных точек проверки работоспособности, которые следует использовать вместо того, чтобы создавать собственные. Подробнее о создании проверок работоспособности можно узнать в документе Microsoft «Проверка работоспособности в ASP.NET Core»: <http://mng.bz/nMA2>.

В качестве одного из решений можно было бы создать ветвь с помощью метода расширения `Map` и поместить в нее промежуточное ПО для проверки работоспособности, как показано на рис. 31.1. Только те запросы, которые соответствуют шаблону `/ping`, будут выполнять ветвь; все остальные запросы будут обрабатываться стандартным компонентом маршрутизации и Razor Pages на основном конвейере.

Листинг 31.2 Использование метода расширения `Map` для создания ветвящихся конвейеров

Это промежуточное ПО будет работать только для запросов, соответствующих ветке `/ping`

```
app.UseStatusCodePages(); <-- Каждый запрос будет проходить через это промежуточное ПО
app.Map("/ping", (IApplicationBuilder branch) => <-- Метод расширения Map создаст новую ветвь, если запрос начинается с /ping
{
    branch.UseExceptionHandler();
    branch.Run(async (HttpContext context) =>
    {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync("pong");
    });
});

app.UseStaticFiles();
app.UseRouting();

app.MapRazorPages();
app.Run();
```

Остальная часть конвейера промежуточного ПО будет выполняться для запросов, не соответствующих ветке `/ping`

Метод расширения Run всегда возвращает ответ, но только в ветке `/ping`

Мы создали абсолютно новый объект `IApplicationBuilder` (в листинге он называется `branch`), который можно настроить так же, как и основной конвейер `app`. Промежуточное ПО, добавленное в `branch`, добавляется только в ветвь, а не в основной конвейер.

СОВЕТ Объект `WebApplication`, к которому вы обычно добавляете промежуточное ПО, реализует интерфейс `IApplicationBuilder`. Большинство методов расширения для добавления промежуточного программного обеспечения используют интерфейс `IApplicationBuilder`, поэтому вы можете использовать методы расширения в ветвях, а также в основном конвейере промежуточного программного обеспечения.

В этом примере вы добавляете промежуточное ПО методом `Run` в ветвь, поэтому оно будет выполняться только для запросов, начинающихся с `/ping`, например `/ping`, `/ping/go` или `/ping?id=123`. Все запросы, которые не начинаются с `/ping`, метод `Map` игнорирует. Они остаются в основном конвейере и выполняют следующие компоненты в конвейере после `Map` (в данном случае `StaticFilesMiddleware`).

ВНИМАНИЕ! Существует несколько перегрузок метода расширения `Map`. Некоторые из них являются методами расширения `IApplicationBuilder` и используются для ветвления конвейера, как вы видели в листинге 31.2. Другие перегрузки являются расширениями `IEndpointRouteBuilder` и используются для создания минимальных конечных точек с использованием системы маршрутизации конечных точек. Если у вас возникли проблемы с компиляцией приложения, убедитесь, что вы случайно не используете неправильную перегрузку метода `Map`!

При необходимости с помощью метода `Map` можно создавать раскидистые ветви в конвейере, где ветви будут независимы друг от друга. Кроме того, можно вкладывать вызовы в `Map`, чтобы у вас были ветви, которые отходят от других ветвей.

Метод расширения `Map` может быть полезным, но если вы переусердствуете, то быстро можете запутаться. Помните, что нужно использовать промежуточное ПО для реализации сквозных задач или очень простых конечных точек. Механизм маршрутизации конечных точек контроллеров и Razor Pages лучше подходит для более сложных требований к маршрутизации, поэтому не бойтесь его использовать.

Одна из ситуаций, в которой метод `Map` может быть полезен, – когда вам нужно два «независимых» более мелких приложения, но вы хотите избавить себя от хлопот, связанных с несколькими развертываниями. Можно использовать этот метод, чтобы конвейеры были разделены, с отдельной маршрутизацией и конечными точками внутри каждой ветви.

СОВЕТ Этот подход может быть полезен, например, если вы встраиваете сервер OpenID Connect, такой как `IdentityServer`, в ваше приложение. Путем сопоставления `IdentityServer` в ветвь вы гарантируете, что конечные точки и контроллеры в вашем основном приложении не помешают конечным точкам, предоставляемым `IdentityServer`.

Просто имейте в виду, что обе эти ветви будут иметь одну и ту же конфигурацию и контейнер внедрения зависимостей, поэтому они независимы только с точки зрения конвейера промежуточного ПО.

ПРИМЕЧАНИЕ Создание по-настоящему независимых ветвей в одном и том же приложении требует гораздо больше усилий. См. пост в блоге Филипа В. «Запуск нескольких независимых конвейеров ASP.NET Core бок о бок в одном приложении»: <http://mng.bz/vzA4>.

Последний момент, о котором следует помнить при использовании метода расширения `Map`, заключается в том, что он изменяет фактическое свойство `Path`, которое видит промежуточное ПО в ветви. Если оно совпадает с префиксом URL-адреса, метод `Map` отсекает совпадающий сегмент от пути, как показано на рис. 31.2. Удаленные сегменты хранятся в свойстве `PathBase` объекта `HttpContext`, чтобы быть доступными, когда они вам понадобятся.

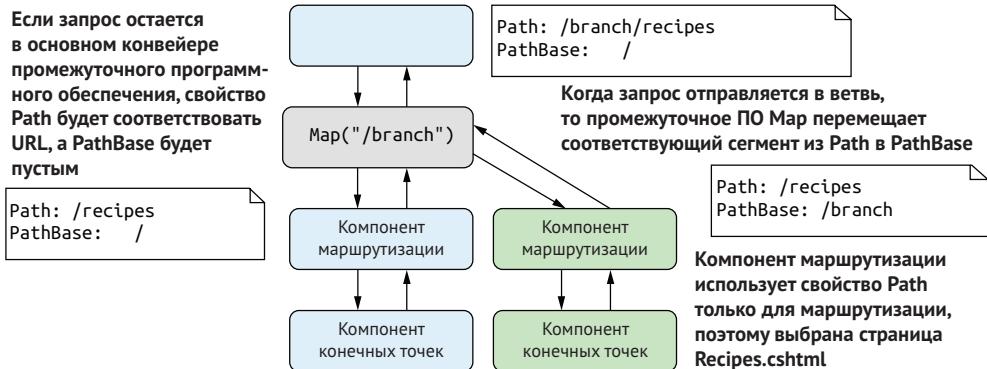


Рис. 31.2 Когда метод расширения `Map` перенаправляет запрос в ветвь, он удаляет совпадающий сегмент из свойства `Path` и добавляет его в свойство `PathBase`

ПРИМЕЧАНИЕ Генератор ссылок ASP.NET Core (используемый, например, в Razor, как обсуждалось в главе 5) использует свойство `PathBase`, чтобы гарантировать, что он генерирует URL-адреса, которые включают `PathBase` в качестве префикса. Вы уже видели метод расширения `Run`, который всегда возвращает ответ, и метод расширения `Map`, который создает ветвь в конвейере. Следующий метод расширения, который мы рассмотрим, – это обобщенный метод `Use`.

31.1.3 Добавление в конвейер с помощью метода расширения `Use`

Метод расширения `Use` можно использовать для добавления компонентов общего назначения. Его можно использовать для просмотра или изменения запросов по мере их поступления, для генерации ответа или передачи запроса следующему компоненту в конвейере.

Как и в случае с методом расширения `Run`, когда вы добавляете метод `Use` в конвейер, то указываете лямбда-функцию, которая выполняется, когда запрос доходит до компонента. Приложение передает в эту функцию два параметра:

- *HttpContext*, представляющий текущий запрос и ответ. Вы можете использовать его, чтобы проверить запрос или сгенерировать ответ, как было показано в случае с расширением `Run`;
- *указатель на остальную часть конвейера в виде `Func<Task>`*. Выполняя эту задачу, вы можете выполнить остальную часть конвейера.

Предоставляя указатель на остальную часть конвейера, вы можете использовать метод расширения `Use`, чтобы точно контролировать, как и когда будет выполняться остальная часть конвейера, как показано на рис. 31.3. Если вы вообще не вызываете предоставленный `Func<Task>`, остальная часть конвейера не выполняется, таким образом вы получаете полный контроль.

Предоставляя остальную часть конвейера в виде `Func<Task>`, вы условно упрощаете прерывание выполнения конвейера, что открывает мно-

жество различных сценариев. Вместо ветвления конвейера для реализации компонента проверки работоспособности с помощью методов `Map` и `Run`, как вы это делали в листинге 31.2, можно использовать единственный экземпляр метода расширения `Use`. Это обеспечивает ту же необходимую функциональность, что и раньше, но без ветвления конвейера.

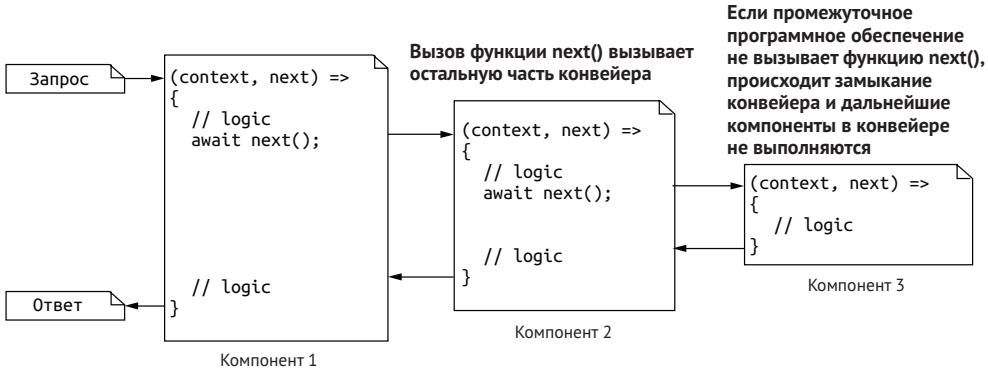


Рис. 31.3 Три компонента промежуточного ПО, созданных с помощью метода расширения `Use`. Вызывая `Func<Task>` с помощью метода `next()`, вы вызываете остальную часть конвейера. Каждый компонент может выполнять код до и после вызова остальной части конвейера или может вообще не вызывать метод `next()`, чтобы прервать выполнение конвейера

Листинг 31.3 Использование метода расширения `Use` для создания компонента для проверки работоспособности

```
Метод расширения Use принимает
лямбда-функцию с параметрами
HttpContext(context) и Func<Task>(next)

app.Use(async (HttpContext context, Func<Task> next) =>
{
    Если путь
    совпадает,
    генериру-
    ем ответ и
    прерываем
    выполне-
    ние кон-
    вейера
    {
        if (context.Request.Path.StartsWithSegments("/ping"))
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("pong");
        }
        else
        {
            await next();
        }
    });
    Если путь не совпадает, вызываем следующий
    компонент промежуточного ПО в конвейере,
    в данном случае метод UseStaticFiles()
});

app.UseStaticFiles();
```

Если входящий запрос начинается с необходимого сегмента пути (`/ping`), компонент отвечает и не вызывает остальную часть конвейера. В противном случае расширение вызывает следующий компонент в конвейере, и ветвление не требуется.

С помощью метода расширения `Use` можно контролировать, когда вызывать остальную часть конвейера. Но важно отметить, что обычно не

следует изменять объект `Response` после вызова метода `next()`. Вызов метода `next()` запускает остальную часть конвейера, поэтому последующие компоненты могут начать потоковую передачу ответа в браузер. Если вы попытаетесь изменить ответ *после выполнения* конвейера, то в конечном итоге можете повредить ответ или отправить неверные данные.

ВНИМАНИЕ! Не изменяйте объект `Response` после вызова метода `next()`. Кроме того, не вызывайте его, если вели запись в `Response.Body`; запись в этот поток может инициировать Kestrel начать потоковую передачу ответа браузеру, а это может привести к отправке недействительных данных. Обычно можно безопасно считывать данные из объекта `Response`; например, чтобы проверить окончательный код состояния `StatusCode` или `ContentType` ответа.

Еще один распространенный вариант использования метода расширения `Use` – изменение каждого запроса или ответа, который проходит через него. Например, существуют различные HTTP-заголовки, которые вы должны отправлять со всеми своими приложениями из соображений безопасности. Эти заголовки часто отключают небезопасные и устаревшие варианты поведения браузеров или ограничивают функции, активированные браузером. В главе 28 вы узнали о заголовке HSTS, но есть и другие заголовки, которые вы можете добавить для дополнительной безопасности.

СОВЕТ Можно проверить заголовки безопасности для своего приложения на сайте <https://securityheaders.com>, который также предоставляет информацию о том, какие заголовки нужно добавить в свое приложение и почему.

Представьте, что вам было поручено добавить один такой заголовок, `X-Content-Type-Options: nosniff` (который обеспечивает дополнительную защиту от XSS-атак), в каждый ответ, создаваемый вашим приложением. Такой вид сквозной задачи идеально подходит для промежуточного ПО. Вы можете использовать метод расширения `Use`, чтобы перехватывать каждый запрос, задать заголовок ответа, а затем выполнить остальную часть конвейера. Не важно, какой ответ генерирует конвейер, будь то статический файл, ошибка или страница Razor. Ответ всегда будет иметь заголовок безопасности.

В листинге 31.4 показан надежный способ сделать это. Когда промежуточное ПО получает запрос, оно регистрирует функцию обратного вызова, которая выполняется до того, как Kestrel начнет отправлять ответ обратно в браузер. Затем вызывается метод `next()` для запуска остальной части конвейера. Когда конвейер генерирует ответ, вероятно в каком-либо компоненте, который идет позже, Kestrel выполняет функцию обратного вызова и добавляет заголовок. Такой подход гарантирует, что заголовок не окажется случайно удален другим компонентом в конвейере, а также что вы не пытаетесь изменить заголовки после того, как ответ начал потоковую передачу в браузер.

Листинг 31.4 Добавление заголовков к ответу с методом расширения Use

```

    app.Use(async (HttpContext context, Func<Task> next) =>
{
    context.Response.OnStarting(() => {
        context.Response.Headers["X-Content-Type-Options"] = "nosniff";
        return Task.CompletedTask;
    });
    await next();
}

app.UseStaticFiles();
app.UseRouting();
app.MapRazorPages();

```

Добавляет промежуточное ПО в начало конвейера

Вызывает остальную часть конвейера промежуточного ПО

Задает функцию, которая должна вызываться перед отправкой ответа в браузер

Добавляет заголовок в ответ для дополнительной защиты от XSS-атак

Функция, переданная в OnStarting, должна возвращать задачу

Независимо от того, какой ответ будет генерирован, в него будет добавлен заголовок безопасности

Такие простые компоненты, которые вы видели в примере с заголовком безопасности, – обычное явление, но они могут быстро загромоздить метод `Configure` и затруднить понимание конвейера с первого взгляда. Вместо этого обычно принято инкапсулировать промежуточное ПО в класс, функционально эквивалентный методу расширения `Use`, но который легко протестировать и использовать повторно.

31.1.4 Создание пользовательского компонента промежуточного программного обеспечения

Удобно создавать промежуточное ПО, используя метод расширения `Use`, как вы это делали в листингах 31.3 и 31.4, но его нелегко протестировать, и вы несколько ограничены в своих возможностях. Например, нельзя просто использовать внедрение зависимостей для внедрения сервисов с жизненным циклом `Scoped` внутри этих базовых компонентов. Обычно вместо того, чтобы напрямую вызвать метод расширения `Use`, промежуточное ПО инкапсулируется в функционально эквивалентный класс.

Специальные компоненты промежуточного ПО не обязательно должны наследовать от определенного базового класса или реализовывать интерфейс, но они имеют определенную форму, как показано в листинге 31.5. ASP.NET Core использует отражение для вызова метода во время исполнения приложения. У классов промежуточного ПО должен быть конструктор, принимающий объект `RequestDelegate`, который представляет остальную часть конвейера, и функция `Invoke` с сигнатурой, похожей на эту:

```
public Task Invoke(HttpContext context);
```

Функция `Invoke()` эквивалентна лямбда-функции из метода расширения `Use` и вызывается при получении запроса. Вот как можно преобразовать класс `HeadersMiddleware` из листинга 31.4 в отдельный класс промежуточного ПО:

Листинг 31.5 Добавление заголовков к объекту Response с использованием специального компонента промежуточного ПО

```
public class HeadersMiddleware
{
    private readonly RequestDelegate _next;
    public HeadersMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    public async Task Invoke(HttpContext context)
    {
        context.Response.OnStarting(() =>
        {
            context.Response.Headers["X-Content-Type-Options"] =
                "nosniff";
            return Task.CompletedTask;
        });
        await _next(context); ←
    }
}
```

Метод `Invoke` вызывается с `HttpContext` при получении запроса

RequestDelegate представляет остальную часть конвейера промежуточного ПО

Добавляет заголовок в ответ, как и раньше

Вызывает остальную часть конвейера промежуточного ПО. Обратите внимание, что вы должны передать предоставленный `HttpContext`

ПРИМЕЧАНИЕ Использование такого подхода делает промежуточное программное обеспечение более гибким. В частности, это означает, что вы можете легко использовать DI для внедрения сервисов в метод `Invoke`. Это было бы невозможно, если бы метод `Invoke` был переопределенным методом базового класса или интерфейсом. Однако при желании вы можете реализовать интерфейс `IMiddleware`, который определяет базовый метод `Invoke`.

Данный компонент фактически идентичен примеру из листинга 31.4, но он инкапсулирован в классе `HeadersMiddleware`. Вы можете добавить этот компонент в свое приложение в `Startup.Configure` путем вызова `app.UseMiddleware<HeadersMiddleware>();`

Существует распространенный шаблон, который состоит в том, чтобы создавать вспомогательные методы расширения, дабы упростить потребление вашего метода расширения из `Startup.Configure` (чтобы Intelli-Sense отображал его как параметр в экземпляре `IApplicationBuilder`). Вот как можно создать простой метод расширения для `HeadersMiddleware`.

Листинг 31.6 Создание метода расширения для предоставления `HeadersMiddleware`

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSecurityHeaders(
        this IApplicationBuilder app)
```

По соглашению, метод расширения должен возвращать экземпляр `IApplicationBuilder`, чтобы разрешить построение цепочки

```
{
    return app.UseMiddleware<HeadersMiddleware>(); ←
}
} Добавляет промежуточное  
ПО в конвейер
```

С помощью этого метода расширения вы теперь можете добавить компонент заголовков в свое приложение, используя

```
app.UseSecurityHeaders();
```

СОВЕТ Существует пакет NuGet, SecurityHeaders, который упрощает добавление заголовков безопасности с помощью промежуточного ПО без необходимости писать свои собственные. Он предоставляет удобный интерфейс для добавления рекомендуемых заголовков безопасности в свое приложение. Инструкции по его установке можно найти на GitHub: <https://github.com/andrewlock/NetEscapades.AspNetCore.SecurityHeaders>.

Листинг 31.5 представляет собой простой пример, но вы можете создать промежуточное ПО для множества различных целей. В некоторых случаях вам может потребоваться использовать внедрение зависимостей для внедрения сервисов и их использования для обработки запроса. Вы можете внедрять сервисы с моделью жизненного цикла Singleton в конструктор своего компонента или сервисы с любым типом цикла в метод Invoke, как показано в следующем листинге.

Листинг 31.7 Использование внедрения зависимостей в компонентах промежуточного ПО

```
public class ExampleMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ServiceA _a;
    public HeadersMiddleware(RequestDelegate next, ServiceA a)
    {
        _next = next;
        _a = a;
    }
    public async Task Invoke(
        HttpContext context, ServiceB b, ServiceC c) ←
    {
        // Используем сервисы a, b и c
        // и/или вызываем _next.Invoke(context);
    }
}
```

Вы можете внедрить в конструктор дополнительные сервисы. Это должны быть синглтоны

Вы можете внедрить сервисы в метод Invoke. У них может быть любой жизненный цикл

ВНИМАНИЕ! ASP.NET Core создает промежуточное ПО только один раз за весь жизненный цикл вашего приложения, поэтому любые зависимости, внедряемые в конструктор, должны иметь жизненный цикл Singleton. Если вам нужно использовать зависимости с жизненными циклами Scoped или Transient, внедрите их в метод Invoke.

Помимо сквозных сценариев, хорошим применением промежуточного программного обеспечения является создание простых обработчиков с как можно меньшим количеством зависимостей, которые обрабатывают запросы на фиксированный URL-адрес, аналогично методу расширения «Use», о котором вы узнали в разделе 31.1.3. Эти простые обработчики могут быть встроены в несколько приложений, независимо от того, как настроена маршрутизация приложения.

Так называемые общеизвестные унифицированные идентификаторы ресурсов (URI) являются хорошим сценарием использования простых обработчиков промежуточного программного обеспечения для общеизвестных URI security.txt (<https://www.rfc-editor.org/rfc/rfc9116>) и URI OpenID Connect (<http://mng.bz/wj2>). Эти обработчики всегда отвечают на один путь, поэтому они могут аккуратно инкапсулировать всю логику без риска вмешательства в любую другую конфигурацию маршрутизации.

В листинге 31.8 показан простой пример обработчика security.txt, реализованного как промежуточное программное обеспечение. Он всегда отвечает на известный путь с фиксированным значением, и его легко добавить в любое приложение, вызвав `app.UseMiddleware<SecurityTxtHandler>`.

Листинг 31.8 Обработчик Security.txt, реализованный как промежуточное ПО

```
public class SecurityTxtHandler
{
    private readonly RequestDelegate _next;
    public SecurityTxtHandler(RequestDelegate next)
    {
        _next = next;
    }
    public Task Invoke(HttpContext context)
    {
        var path = context.Request.Path;
        if(path.StartsWithSegments("/.well-known/security.txt"))
        {
            context.Response.ContentType = "text/plain";
            return context.Response.WriteAsync(
                "Contact: mailto:security@example.com");
        }
        return _next.Invoke(context);
    }
}
```

Промежуточное программное обеспечение ищет фиксированный, хорошо известный путь

Если путь совпадает, промежуточное программное обеспечение возвращает ответ

Если путь не совпал, вызывается следующее промежуточное программное обеспечение в конвейере

Здесь мы рассмотрели практически все, что вам нужно, чтобы начать создавать свои собственные компоненты промежуточного программного обеспечения. Инкапсулируя промежуточное программное обеспечение в специальные классы, вы можете легко протестировать его поведение или распространить его в пакетах NuGet, поэтому я на-

стоятельно рекомендую использовать этот подход. Помимо всего прочего, это сделает файл Program.cs менее загроможденным и более понятным.

31.1.5 Преобразование промежуточного ПО в конечные точки

В этом разделе вы узнаете, как создавать специальные конечные точки из промежуточного ПО с использованием маршрутизации конечных точек. Мы возьмем простые ветви конвейера, использованные в разделе 31.1.2, и преобразуем их для применения маршрутизации конечных точек. Я также продемонстрирую дополнительные функции, такие как маршрутизация и авторизация.

В разделе 31.1.2 я описал создание простой конечной точки с помощью методов расширения `Map` и `Run`, которые возвращают ответ `pong` в виде простого текста всякий раз, когда вы получаете запрос `/ping` путем ветвления конвейера промежуточного ПО. Это прекрасно, потому что все так просто, но что, если у вас более сложные требования?

Усовершенствуем пример с отправкой запроса и получением ответа (`ping-pong`): как бы вы добавили авторизацию в запрос? Компонент `AuthorizationMiddleware`, добавленный в конвейер с помощью метода `UseAuthorization()`, ищет метаданные конечных точек, таких как `Razor Pages`, чтобы узнать, есть ли там атрибут `[Authorize]`, но он не умеет работать с методом расширения `Map`.

Аналогично, что, если вы захотите использовать более сложную маршрутизацию? Возможно, вы хотите иметь возможность вызывать `/ping/3` и получать от своего промежуточного программного обеспечения ответ «`pong-pong-pong`». (Нет, я не могу представить, зачем вам это!) Теперь вам нужно попытаться проанализировать это целое число из URL-адреса, убедиться, что оно допустимо, и т. д. Это звучит как гораздо больше работы и, по-видимому, является четким показателем того, что вам следует создать конечную точку минимального API с использованием маршрутизации конечных точек!

Для нашей простой конечной точки «пинг-понг» это будет несложно сделать, но что, если у вас есть более сложный компонент промежуточного программного обеспечения, который вы не хотите полностью переписывать?

Есть ли способ преобразовать промежуточное программное обеспечение в конечную точку? Давайте представим, что вам нужно применить авторизацию к простой конечной точке пинг-понга, которую вы создали в разделе 31.1.2. Этого гораздо проще добиться с помощью маршрутизации конечных точек, чем с помощью простых ветвей промежуточного программного обеспечения, таких как `Map` или `Use`, но давайте представим, что вы хотите придерживаться использования промежуточного программного обеспечения вместо традиционной минимальной конечной точки API. Первый шаг – создание автономного компонента промежуточного программного обеспечения для этой функциональности, используя подход, который вы видели в разделе 31.1.4, как показано в следующем листинге.

Листинг 31.9 Класс PingPongMiddleware, реализованный в виде компонента промежуточного ПО

```
public class PingPongMiddleware
{
    public PingPongMiddleware(RequestDelegate next)
    {
    }

    public async Task Invoke(HttpContext context)
    {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync("pong");
    }
}
```

Даже если он не используется в данном случае, вы должны добавить RequestDelegate в конструктор

Метод Invoke вызывается для выполнения промежуточного ПО

Промежуточное ПО всегда возвращает ответ «pong»

Обратите внимание, что этот компонент всегда возвращает ответ "pong", независимо от URL-адреса запроса – мы настроим путь "/ping" позже. Можно использовать этот класс для преобразования конвейера промежуточного ПО, чтобы вместо версии с ветвлением, показанной на рис. 31.1, получить версию с конечной точкой, показанной на рис. 31.4.

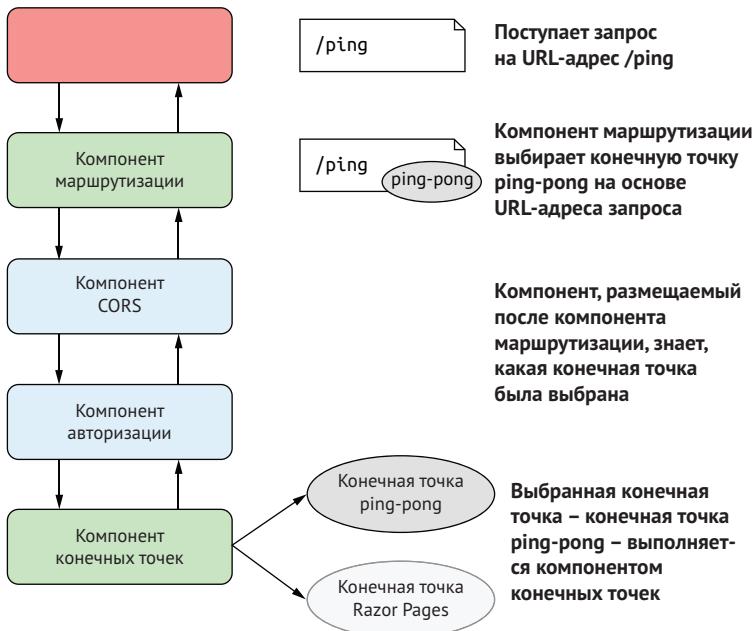


Рис. 31.4 Маршрутизация конечной точки отделяет выбор конечной точки от выполнения конечной точки. Компонент маршрутизации выбирает конечную точку на основе входящего запроса и предоставляет метаданные о конечной точке. Компонент, размещенный перед компонентом конечной точки, может действовать в зависимости от выбранной конечной точки, например завершать неавторизованные запросы. Если запрос авторизован, компонент конечной точки выполняет выбранную точку и генерирует ответ

Преобразование компонента ping-pong в конечную точку не требует никаких изменений в самом промежуточном ПО. Вместо этого нужно создать мини-конвейер, где содержится только компонент ping-pong.

СОВЕТ По существу, преобразование промежуточного ПО, генерирующего ответы, в конечную точку требует преобразовать его в его же собственный мини-конвейер, чтобы при желании вы могли включить в него дополнительные компоненты.

Чтобы создать мини-конвейер, вы вызываете `CreateApplicationBuilder()` в экземпляре `IEndpointRouteBuilder`, который создает новый `IAApplicationBuilder`. Существует два способа доступа к `IEndpointRouteBuilder`: вызвать `UseEndpoints(endpoints => {})` и использовать переменную `endpoints` или явно привести тип `WebApplication` к `IEndpointRouteBuilder`.

ПРИМЕЧАНИЕ Хотя `WebApplication` реализует интерфейс `IEndpointRouteBuilder`, оно намеренно скрывает от вас расширенный метод `CreateApplicationBuilder()!` Это хороший признак того, что вы находитесь на продвинутой территории, и, вероятно, вам следует вместо этого рассмотреть возможность использования минимальных конечных точек API.

В следующем листинге мы создаем новый `IAApplicationBuilder`, добавляем к нему промежуточное программное обеспечение, составляющее конечную точку, а затем вызываем `Build()` для создания конвейера. Если у вас есть конвейер, вы можете связать его с заданным маршрутом, вызвав `Map()` в экземпляре `IEndpointRouteBuilder` и передав шаблон маршрута.

Листинг 31.10 Отображение конечной точки пинг-понга в `UseEndpoints`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
var endpoint = ((IEndpointRouteBuilder)app) <-- 
    .CreateApplicationBuilder() <-- 
    .UseMiddleware<PingPongMiddleware>()
    .Build(); <-- 

    app.Map("/ping", endpoint); <-- 
    app.MapRazorPages();
    app.MapHealthChecks("/healthz");
    app.Run(); <-- 

Добавляем новую конечную точку
в коллекцию конечных точек, ассоциируя с шаблоном маршрута "/ping"
```

Преобразуем `WebApplication` к `IEndpointRouteBuilder`, чтобы можно было вызвать `CreateApplicationBuilder`

Создаем миниатюрный автономный экземпляр `IAApplicationBuilder` для построения конечной точки

Добавляем промежуточное ПО и создаем окончательную конечную точку. Выполнение происходит при вызове конечной точки

СОВЕТ Обратите внимание, что функция `Map()` в `IEndpointRouteBuilder` создает новую конечную точку (состоящую из вашего микронавигатора) с ассоциированным маршрутом. Хотя у нее то же имя, она концептуально отличается от функции `Map` для `IApplicationBuilder` из раздела 31.1.2, которая используется для ветвления конвейера. Он аналогичен методам `MapGet` (и родственным), которые вы используете для создания минимальных конечных точек API.

Как это обычно бывает с ASP.NET Core, вы можете извлечь эту несколько многословную функциональность в метод расширения, чтобы облегчить чтение и обнаружение вашей конечной точки. Следующий листинг извлекает код для создания конечной точки из листинга 31.10 в отдельный класс, используя шаблон маршрута для использования в качестве параметра метода.

Листинг 31.11 Метод расширения для использования `PingPongMiddleware` в качестве конечной точки

```
Создаем метод расширения для
регистрации PingPongMiddleware
в качестве конечной точки

public static class EndpointRouteBuilderExtensions
{
    public static IEndpointConventionBuilder MapPingPong(
        this IEndpointRouteBuilder endpoints,
        string route)    ← Позволяет вызывающей стороне передать
    {                                шаблон маршрута для конечной точки
        var pipeline = endpoints
            .CreateApplicationBuilder()
            .UseMiddleware<PingPongMiddleware>()
            .Build();

        return endpoints
            .Map(route, pipeline)
            .RequireAuthorization();   ← Вы можете добавить сюда дополнительные метаданные напрямую, либо вызывающая сторона может добавить метаданные
    }                                самостоятельно
}
```

Теперь, когда у вас есть метод расширения `MapPingPong()`, можете обновить свой код сопоставления конечных точек, чтобы он стал проще и понятнее:

```
app.MapPingPong("/ping");
app.MapRazorPages();
app.MapHealthChecks("/healthz");
```

Поздравляю, вы создали свою первую конечную точку! Превратив промежуточное программное обеспечение в конечную точку, вы теперь можете добавлять дополнительные метаданные, как показано

в листинге 31.11. Ваше промежуточное программное обеспечение подключено к системе маршрутизации конечных точек и извлекает выгоду из всего, что она предлагает.

В примере в листинге 31.11 использовался базовый шаблон маршрута «/ping», но вы также можете использовать шаблоны в пути, содержащие параметры маршрута, например «/ping/{count}», как и в случае с минимальными API. Большая разница в том, что вы не получите преимуществ привязки модели, которые вы получаете от минимальных API, и это явно требует больше усилий, чем просто использование минимальных API!

СОВЕТ Примеры доступа к данным маршрута из вашего промежуточного программного обеспечения, а также рекомендации по передовому опыту см. в моей записи в блоге под названием «Доступ к значениям маршрута в промежуточном программном обеспечении конечной точки в ASP.NET Core 3.0» по адресу <http://mng.bz/4ZRj>.

Преобразование существующего промежуточного ПО, такого как PingPongMiddleware, для работы с маршрутизацией конечных точек может быть полезно, если вы уже реализовали это ПО, но вам придется писать большое количество шаблонного кода, если вы хотите создать новую простую конечную точку. Почти во всех случаях вместо этого следует использовать конечные точки минимальных API. Но если вам когда-нибудь понадобится повторно использовать существующее промежуточное программное обеспечение в качестве конечной точки, теперь вы знаете, как это сделать!

В следующем разделе мы отойдем от конвейера промежуточного программного обеспечения и рассмотрим, как удовлетворить общие требования к конфигурации: использование сервисов внедрения зависимостей для создания строго типизированных объектов IOptions.

31.2 Использование внедрения зависимостей с OptionsBuilder и IConfigureOptions

В этом разделе я описываю, как действовать в распространенном сценарии: вы хотите использовать сервисы, зарегистрированные в контейнере внедрения зависимостей, для настройки объектов IOptions<T>. Есть несколько способов добиться этого, но в этом разделе я представляю OptionsBuilder<T> как один из возможных подходов и выделяю некоторые другие возможности, которые он обеспечивает.

В главе 10 мы подробно обсудили систему конфигурации ASP.NET Core. Вы видели, как объект IConfiguration строится из нескольких уровней, где последующие уровни могут добавлять или заменять значения конфигурации из предыдущих уровней. Каждый уровень добавляется поставщиком конфигурации, который считывает значения из файла, из переменных среды, из пользовательских секретов или из любого количества возможных мест.

Распространенной и рекомендуемой практикой является привязка вашего объекта конфигурации к строго типизированным объектам `IOptions<T>`, как вы видели в главе 10. Обычно вы настраиваете эту привязку в `Program.cs`, вызывая `builder.Services.Configure<T>()` и предоставляемый объект `IConfiguration` или раздел конфигурации для привязки. Например, чтобы привязать строго типизированный объект `CurrencyOptions` к разделу «`Currencies`» объекта `IConfiguration`, вы можете использовать следующее:

```
builder.Services.Configure<CurrencyOptions>(
    Configuration.GetSection("Currencies"));
```

СОВЕТ Вы увидите пример типа `CurrencyOptions` и связанный с ним раздел «`Currencies`» файла `appsettings.json` в исходном коде данной главы.

При этом свойства объекта `CurrencyOptions` устанавливаются на основе значений из раздела «`Currencies`» вашего объекта `IConfiguration`. Простая привязка, подобная этой, является общепринятой, хотя иногда вам может не потребоваться настройка объектов `IOptions<T>` через систему конфигурации; вместо этого вы можете настроить их в коде. Паттерн `IOptions` требует только настройки строго типизированного объекта перед его внедрением в зависимый сервис; это не требует привязки его к разделу `IConfiguration`.

СОВЕТ Технически, даже если вы вообще не настраиваете `IOptions<T>`, вы можете все равно внедрить его в свои сервисы. В этом случае объект `T` просто создается с использованием конструктора по умолчанию.

Метод `Configure<T>()` имеет перегруженный метод, принимающий лямбда-функцию. Фреймворк выполняет лямбда-функцию для настройки объекта `CurrencyOptions`, когда он внедряется с помощью DI. В следующем листинге показан пример, в котором используется лямбда-функция для установки фиксированного массива строк для свойства `Currencies` в настроенном объекте `CurrencyOptions`.

Листинг 31.12 Настройка объекта `IOptions` с помощью лямбда-функции

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<CurrencyOptions>(
    builder.Configuration.GetSection("Currencies"));

builder.Services.Configure<CurrencyOptions>(options =>
    options.Currencies = new string[] { "GBP", "USD" });

WebApplication app = builder.Build();
app.MapGet("/", (IOptions<CurrencyOptions> opts) => opts.Value);
app.Run();

Внедренное значение IOptions создается путем привязки к конфигурации, а затем применения лямбда-выражения
```

Настраивает объект `IOptions` путем привязки к разделу `IConfiguration`

Настраивает объект `IOptions`, выполняя лямбда-функцию

Каждый вызов `Configuration<T>()` (как привязка к `IConfiguration`, так и лямбда-функция) добавляет еще один шаг настройки к объекту `CurrencyOptions`. Когда контейнеру внедрения зависимостей впервые требуется экземпляр `IOptions<CurrencyOptions>`, шаги выполняются по очереди, как показано на рис. 31.5.

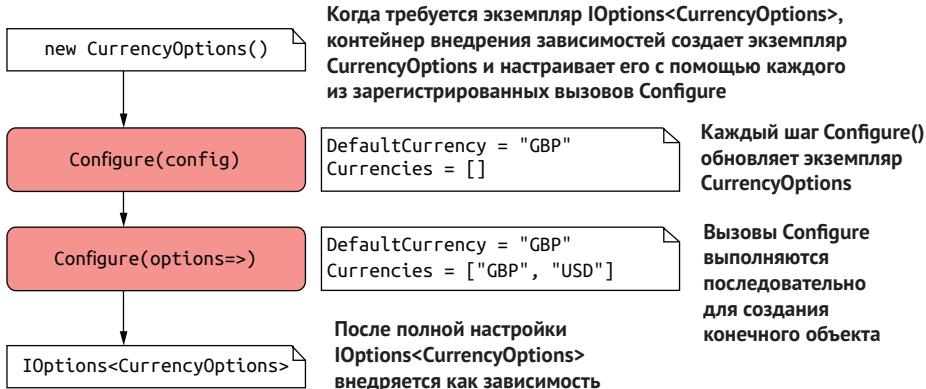


Рис. 31.5 Настройка объекта `CurrencyOptions`. Когда контейнеру внедрения зависимостей требуется экземпляр `IOptions<T>` строго типизированного объекта, контейнер создает объект, а затем использует каждый из зарегистрированных методов `Configure()`, чтобы задать свойства объекта

В предыдущем фрагменте кода мы задали для свойства `Currencies` статический массив строк в лямбда-функции. Но что, если вы заранее не знаете правильные значения? Вам может потребоваться загрузить доступные валюты из базы данных или, например, из какого-либо удаленного сервиса.

К сожалению, такую ситуацию, когда вам нужен сконфигурированный сервис для настройки `IOptions<T>`, трудно разрешить. Помните, что вы объявляете конфигурацию объекта `IOptions<T>` как часть конфигурации внедрения зависимостей. Как получить полностью настроенный экземпляр сервиса обмена валют, если вы еще не зарегистрировали его в контейнере?

Решение состоит в том, чтобы отложить настройку объекта `IOptions<T>` до последнего момента, непосредственно перед тем, как контейнер внедрения зависимостей должен создать его для обработки входящего запроса.

У этой циклической проблемы есть несколько потенциальных решений, но самый простой подход – использовать альтернативный API для настройки экземпляров `IOptions` с использованием типа `OptionsBuilder<T>`. Этот тип по сути является оболочкой некоторых основных интерфейсов `IOptions`, но он часто приводит к более краткому и удобному синтаксису по сравнению с подходом, который вы видели до сих пор.

СОВЕТ Еще одна полезная функция `OptionsBuilder<T>` – добавление проверки к вашим объектам `IOptions`. Это гарантирует, что ваша конфигурация будет правильно загружена и привязана при

запуске приложения, и, например, у вас не будет опечаток в именах разделов конфигурации. Подробнее о добавлении проверки к вашим объектам `IOptions` можно прочитать в моем блоге по адресу <http://mng.bz/qrjJ>.

В следующем листинге показан эквивалент листинга 31.12, но с использованием `OptionsBuilder<T>`. Вы создаете экземпляр `OptionsBuilder<T>`, вызывая `AddOptions<T>()`, а затем связываете дополнительные методы, такие как `BindConfiguration()` и `Configure()`, для настройки окончательного объекта `IOptions<T>`, создавая уровни конфигурации параметров, как показано ранее на рис. 31.5.

Листинг 31.13 Настройка объекта `IOptions<T>` с помощью `OptionsBuilder<T>`

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddOptions<CurrencyOptions>()
    .BindConfiguration("Currencies")
    .Configure(opts =>
        opts.Currencies = new string[] { "GBP", "USD" });

WebApplication app = builder.Build();
app.MapGet("/", (IOptions<CurrencyOptions> opts) => opts.Value);
app.Run();
```

Creates object `OptionsBuilder<CurrencyOptions>`

Привязывается к разделу «Currencies» `IConfiguration`

Настраивает объект `IOptions`, выполняя лямбда-функцию

Вы неоднократно видели шаблон «Построитель» на протяжении всей книги, и в данном случае он ничем не отличается. Конструктор предоставляет методы, которые вы можете легко объединить.

Одним из преимуществ шаблона построителя является то, что все методы, которые он предоставляет, легко обнаружить. В этом случае, если вы исследуете тип в своей интегрированной среде разработки (IDE), вы можете заметить, что `OptionsBuilder<T>` предоставляет несколько перегрузок, таких как

- `Configure<TDeps>(Action<T, TDeps> config);`
- `Configure<TDeps1, TDeps2>(Action<T, TDeps1, TDeps2> config);`
- `Configure<TDeps1, TDeps2, TDeps3>(Action<T, TDeps1, TDeps2, TDeps3> config).`

Эти методы позволяют вам указывать зависимости, которые автоматически извлекаются из контейнера DI и передаются в действие `config`, когда объект `IOptions` извлекается из контейнера, как показано на рис. 31.6. Пять перегруженных вариантов для `Configure<TDeps>` позволяют внедрять до пяти зависимостей с помощью этих методов.

Применяя этот шаблон, мы можем обновить код из листинга 31.13, чтобы использовать `ICurrencyProvider` всякий раз, когда нашему приложению необходимо создать объект `CurrencyOptions`. Мы можем зарегистрировать сервис в контейнере внедрения зависимостей и знать, что он позаботится о предоставлении ее лямбда-функции во время выполнения, как показано в следующем листинге.

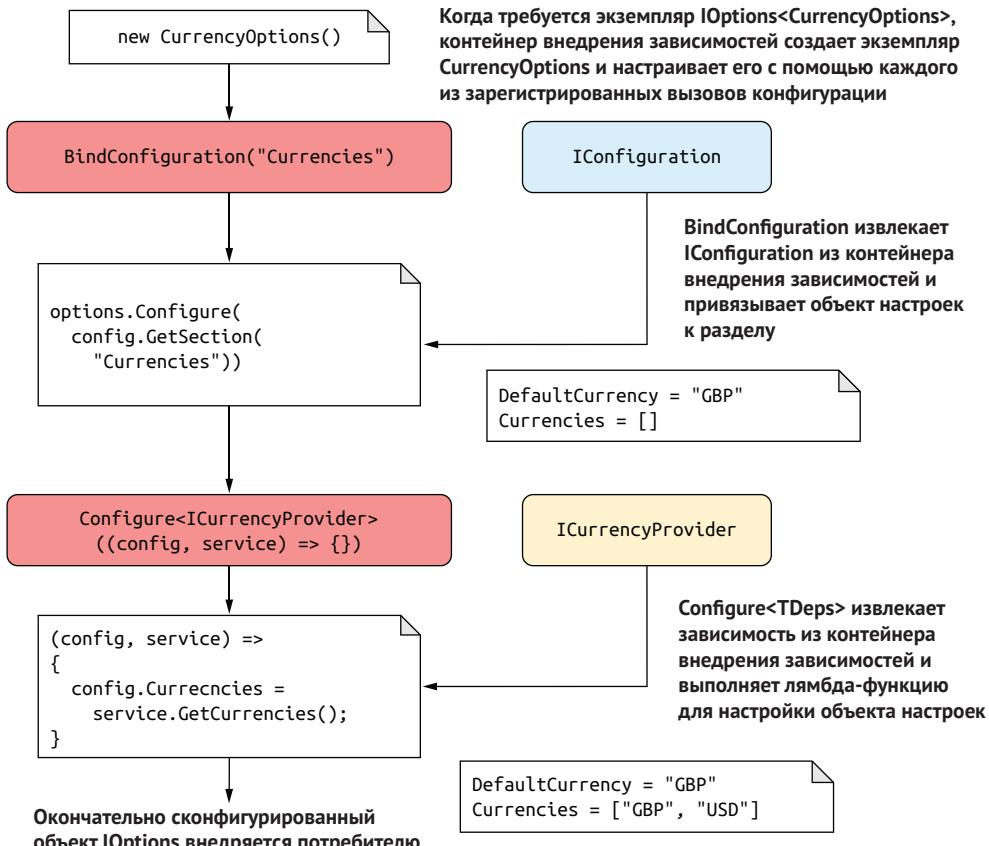


Рис. 31.6 Использование OptionsBuilder для создания объекта `IOptions`. Зависимости, запрашиваемые с помощью методов `Configure<TDeps>`, автоматически извлекаются из контейнера и используются для выполнения лямбда-функции

Листинг 31.14 Использование сервиса из контейнера внедрения зависимостей

```

Регистрирует сервис в контейнере внедрения зависимостей
builder.Services.AddOptions<CurrencyOptions>()
    .BindConfiguration("Currencies")
    .Configure<ICurrencyProvider>((opts, service) =>
        opts.Currencies = service.GetCurrencies());
    builder.Services.AddSingleton<ICurrencyProvider, CurrencyProvider>();

WebApplication app = builder.Build();
app.MapGet("/", (IOptions<CurrencyOptions> opts) => opts.Value);
app.Run();

```

Настраивает объект `IOptions` с помощью сервиса из контейнера внедрения зависимостей

Получает объект `IOptions`, который извлекает сервис из контейнера и запускает лямбда-метод

В конфигурации из листинга 31.14, когда `IOptions<CurrencyOptions>` впервые внедряется в конечную точку минимального API, создается объект `IOptions<CurrencyOptions>`, как описано в `OptionsBuilder`. Во-первых, раздел «Currencies» `IConfiguration` приложения привязывается к новому объекту `CurrencyOptions`. Затем `ICurrencyProvider` извлекается из контейнера и передается в лямбда-функцию `Configure<TDeps>` вместе с объектом параметров. Наконец, объект `IOptions` внедряется в конечную точку.

ВНИМАНИЕ Необходимо внедрять только сервисы с жизненным циклом `Singleton` с помощью методов `Configure<TDeps>`. Если вы попытаетесь внедрить сервис с циклом `Scoped`, например `DbContext`, вы получите сообщение об ошибке в процессе разработки, предупреждающее вас о захваченной зависимости. Я описываю, как обойти эту проблему, в своем блоге <http://mng.bz/7Dve>.

`OptionsBuilder<T>` – это удобный способ настройки объектов `IOptions` с использованием зависимостей, но вы можете использовать альтернативный подход: реализацию интерфейса `IConfigureOptions<T>`. Вы реализуете этот интерфейс в классе конфигурации и используете его для настройки объекта `IOptions<T>` любым необходимым вам способом, как показано в следующем листинге. Этот класс может использовать внедрение зависимостей, поэтому вы можете легко использовать любые другие необходимые сервисы.

Листинг 31.15 Реализация `IConfigureOptions<T>` для настройки объекта параметров

```
Вы можете внедрить службы, доступные только после полной настройки DI
public class ConfigureCurrencyOptions : IConfigureOptions<CurrencyOptions>
{
    private readonly ICurrencyProvider _currencyProvider; ←
    public ConfigureCurrencyOptions(ICurrencyProvider currencyProvider)
    {
        _currencyProvider = currencyProvider; ←
    }                                              Настройка вызывается, когда требуется
                                                    экземпляр IOptions<CurrencyOptions>
    public void Configure(CurrencyOptions options) ←
    {
        options.Currencies = _currencyProvider.GetCurrencies(); ←
    }                                              Использует внедренный сервис
                                                    для загрузки значений
}
```

Остается только зарегистрировать реализацию в DI-контейнере. Как всегда, порядок важен, поэтому если вы хотите, чтобы `ConfigureCurrencyOptions` запускался после привязки к конфигурации, вы должны добавить его после настройки вашего `OptionsBuilder<T>`:

```
builder.Services.AddOptions<CurrencyOptions>()
    .BindConfiguration("Currencies");
builder.AddSingleton
    <ICConfigureOptions<CurrencyOptions>, ConfigureCurrencyOptions>();
```

СОВЕТ Порядок, в котором вы настраиваете параметры, имеет значение. Если вы хотите всегда запускать конфигурацию последней, то есть после всех других методов настройки, вы можете использовать метод `PostConfigure()` в `OptionsBuilder` или интерфейс `IPostConfigureOptions`. Подробнее об этом подходе можно прочитать в моем блоге <http://mng.bz/mVj4>.

В этой конфигурации, когда `IOptions<CurrencyOptions>` внедряется в конечную точку или сервис, объект `CurrencyOptions` сначала привязывается к разделу «`Currencies`» вашего `IConfiguration`, а затем настраивается с помощью класса `ConfigureCurrencyOptions`.

ВНИМАНИЕ Объект `ConfigureCurrencyOptions` регистрируется как одиночка, поэтому он будет захватывать любые внедренные сервисы с циклами `Transient` или `Scoped`.

Независимо от того, используете вы подход с `OptionsBuilder<T>` или `IConfigureOptions<T>`, вам необходимо зарегистрировать зависимость `ICurrencyProvider` в контейнере внедрения зависимостей. В примере кода для этой главы я создал простой сервис `CurrencyProvider` и зарегистрировал его в контейнере внедрения зависимостей, используя

```
builder.Services.AddSingleton<ICurrencyProvider, CurrencyProvider>();
```

По мере роста вашего приложения и добавления дополнительных функций и сервисов вы, вероятно, будете писать больше таких простых регистраций, где вы регистрируете реализацию `Service` для интерфейса `IService`. Встроенный DI-контейнер ASP.NET Core требует явной регистрации каждого из этих сервисов вручную. Если вас это требование разочаровывает, то, возможно, пришло время взглянуть на сторонние DI-контейнеры, которые могут позаботиться о некоторых шаблонах за вас.

31.3 Использование стороннего контейнера внедрения зависимостей

В этом разделе я покажу, как заменить контейнер внедрения зависимостей, используемый по умолчанию сторонней альтернативой, Lamar. Сторонние контейнеры часто предоставляют дополнительные функции по сравнению со встроенным контейнером, такие как сканирование сборок, автоматическая регистрация сервисов и внедрение свойств. Замена встроенного контейнера также может быть полезна при переносе существующего приложения, использующего сторонний контейнер, в ASP.NET Core.

Сообщество .NET использовало контейнеры внедрения зависимостей в течение многих лет, прежде чем ASP.NET Core решил использовать встроенный контейнер. Команда ASP.NET Core хотела найти способ использовать внедрение зависимостей в собственных библиотеках фрейм-

ворка, и они хотели создать общую абстракцию¹, позволяющую заменить встроенный контейнер любимой сторонней альтернативой, например Autofac, StructureMap/Lamar, Ninject, Simple Injector, или Unity.

Встроенный контейнер намеренно ограничен в возможностях, которые он предоставляет, и вряд ли позволит получить больше в будущем. Сторонние контейнеры, напротив, могут предоставлять множество дополнительных функций. Вот некоторые из функций, доступных в Lamar (<https://jasperfx.github.io/lamar/documentation/ioc/>), наследнике StructureMap:

- сканирование сборок для пар «интерфейс/реализация» на основе соглашений;
- автоматическая регистрация конкретных классов;
- внедрение свойств и выбор конструктора;
- автоматическое разрешение Lazy<T>/Func<T>;
- инструменты отладки и тестирования, позволяющие заглянуть внутрь контейнера.

Ни одна из этих функций не является обязательной для запуска приложения, поэтому использование встроенного контейнера имеет смысл, если вы создаете небольшое приложение или новичок в работе с контейнерами внедрения зависимостей. Но если в какой-то переломный момент простота встроенного контейнера становится слишком обременительной, возможно, его стоит заменить.

СОВЕТ Промежуточный подход – использовать NuGet-пакет Scrutor, который добавляет ряд функций во встроенный контейнер внедрения зависимостей, не заменяя его полностью. Введение и примеры см. в посте моего блога «Использование Scrutor для автоматической регистрации своих сервисов в контейнере внедрения зависимостей ASP.NET Core»: <http://mng.bz/MX7B>.

В этом разделе я покажу, как настроить приложение ASP.NET Core, чтобы использовать Lamar для разрешения зависимостей. Это не будет сложный пример или подробное обсуждение самой этой библиотеки. Я приведу минимум информации, чтобы вы могли приступить к работе.

Какой бы сторонний контейнер вы ни выбрали для установки в существующее приложение, общий процесс почти одинаков:

- 1 Установите пакет NuGet.
- 2 Зарегистрируйте сторонний контейнер в `WebApplicationBuilder` в файле `Program.cs`.
- 3 Настройте сторонний контейнер, чтобы зарегистрировать свои сервисы.

¹ Хотя продвижение внедрения зависимостей в качестве основной практики приветствовалось, эта абстракция вызвала ряд противоречий. В посте под названием «Что не так с внедрением зависимостей в ASP.NET Core?» от одного из разработчиков библиотеки SimpleInjector описываются многие аргументы и проблемы: <http://mng.bz/yYAd>. Дополнительную информацию также можно найти на GitHub: <https://github.com/aspnet/DependencyInjection/issues/433>.

Большинство основных .NET-контейнеров включают адаптер, позволяющий добавлять их в приложения ASP.NET Core. Для получения более подробной информации стоит ознакомиться с конкретным руководством для используемого вами контейнера. В случае с Lamar процесс выглядит так:

- 1 Установите пакет `Lamar.Microsoft.DependencyInjection`, используя диспетчер пакетов NuGet, запустив команду `dotnet add package`:

```
dotnet add package Lamar.Microsoft.DependencyInjection
```

Или добавив `<PackageReference>` в файл с расширением `.csproj`:

```
<PackageReference  
Include="Lamar.Microsoft.DependencyInjection" Version="8.1.0" />
```

- 2 Вызовите метод `UseLamar()` для `WebApplicationBuilder.Host` в файле `Program.cs`:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);  
builder.Host.UseLamar(services => {})  
WebApplication app = builder.Build();
```

- 3 Настройте Lamar `ServiceRegistry` в лямбда-методе, переданном методу `UseLamar()`, как показано в следующем листинге. Это базовая конфигурация, но более сложный пример вы можете увидеть в исходном коде данной главы.

Листинг 31.16 Настройка Lamar в качестве стороннего контейнера внедрения зависимостей

```
builder.Host.UseLamar(services => {  
    services.AddAuthorization();  
    services.AddControllers()  
        .AddControllersAsServices();  
  
    services.Scan(_ => {  
        _AssemblyContainingType(typeof(Program));  
        _WithDefaultConventions();  
    });  
})
```

**Требуетс-
я, чтобы
Lamar
испольzo-
валась для
создания
контролле-
ров веб-API**

**Настраиваем сервисы в UseLamar()
вместо builder.Services**

**Вы можете (и должны) добавлять
сервисы платформы ASP.NET Core
в ServiceRegistry, как обычно**

**Lamar может авто-
матически сканиро-
вать ваши сборки
в поисках сервисов,
которые нужно заре-
гистрировать**

В этом примере я использовал соглашения по умолчанию для регистрации сервисов. Вы автоматически регистрируете конкретные классы и сервисы, имена которых соответствуют ожидаемым соглашениям (например, `Service` реализует `IService`). Вы можете изменить эти соглашения или добавить другие регистраций в методе `UseLamar()`.

Класс `ServiceRegistry`, переданный в метод `UseLamar()`, реализует интерфейс `IServiceCollection`. Это означает, что вы можете использовать все встроенные методы расширения, такие как `AddControllers()` и `AddAuthorization()`, для добавления сервисов фреймворка в свой контейнер.

ВНИМАНИЕ! Если вы используете внедрение зависимостей в своих контроллерах MVC (почти наверняка!) и регистрируете эти зависимости в Lamar, а не во встроенным контейнере, вам может потребоваться вызвать метод `AddControllersAsServices()`, как показано в листинге 19.16. Это связано с деталями реализации того, как ваши контроллеры MVC создаются фреймворком. Для получения дополнительной информации см. запись в моем блоге под названием «Активация контроллера и внедрение зависимостей в ASP.NET Core MVC»: <http://mng.bz/aogm>.

При такой конфигурации всякий раз, когда вашему приложению требуется создать сервис, оно будет запрашивать его из контейнера Lamar, который создаст экземпляр класса и дерево его зависимостей. Данный пример не демонстрирует всю мощь Lamar, поэтому обязательно ознакомьтесь с документацией (<https://jasperfx.github.io/lamar/>) и соответствующим исходным кодом для этой главы, чтобы увидеть дополнительные примеры. Даже в скромных приложениях Lamar может значительно упростить код регистрации сервисов, но его дополнительное преимущество состоит в том, чтобы показать все сервисы, которые вы зарегистрировали, и все связанные с ними проблемы.

СОВЕТ Сторонние контейнеры обычно добавляют подходы к настройке, но не изменяют основы работы внедрения зависимостей в ASP.NET Core. Все методы, которые вы видели в этой книге, будут работать независимо от того, используете ли вы встроенный контейнер или сторонний контейнер, поэтому вы можете использовать, например, подход `IConfigureOptions<T>` из раздела 31.2, независимо от того, какой контейнер вы выберете.

На этом мы подошли к концу главы, посвященной продвинутой конфигурации. В этой главе я сосредоточился на некоторых основных компонентах любого приложения ASP.NET Core: промежуточном программном обеспечении, конфигурации и внедрении зависимостей. В следующей главе вы узнаете о дополнительных настраиваемых компонентах с акцентом на Razor Pages и контроллерах веб-API.

Резюме

- Используйте метод расширения `Run` для создания компонентов промежуточного ПО, которые всегда возвращают ответ. Вы всегда должны размещать их в конце конвейера или ветви, поскольку компоненты, размещенные после них, не будут выполнены;
- вы можете создавать ветви в конвейере промежуточного ПО с помощью метода расширения `Map`. Если входящий запрос соответствует указанному префиксу пути, запрос пройдет ветвь конвейера; в противном случае он останется в основном конвейере;
- когда метод расширения `Map` соответствует сегменту пути запроса, он удаляет сегмент из объекта запроса `HttpContext.Path` и пере-

мещает его в свойство `PathBase`, что гарантирует правильную работу маршрутизации в ветвях;

- вы можете использовать метод расширения `Use` для создания компонентов промежуточного ПО общего назначения, которые могут генерировать ответ, изменять запрос или передавать запрос следующему компоненту в конвейере. Это полезно для сквозных задач, таких как добавление заголовка ко всем ответам;
- вы можете инкапсулировать промежуточное ПО в повторно используемый класс. Этот класс должен принимать объект `RequestDelegate` в конструкторе и иметь открытый метод `Invoke()`, который принимает объект `HttpContext` и возвращает `Task`. Для вызова следующего компонента в конвейере вызовите объект `RequestDelegate` с предоставленным объектом `HttpContext`;
- чтобы создать конечные точки, которые генерируют ответ, создайте миниатюрный конвейер, содержащий промежуточное ПО для генерации ответа, и вызовите `endpoints.Map(route, pipeline)`. Маршрутизация конечных точек будет использоваться для сопоставления входящих запросов с вашими конечными точками;
- вы можете настроить объекты `IOptions<T>` с помощью гибкого интерфейса компоновщика. Вызовите `AddOptions<T>()`, чтобы создать экземпляр `OptionsBuilder<T>`, а затем создайте цепочку вызовов конфигурации. `OptionsBuilder<T>` обеспечивает легкий доступ к зависимостям для настройки, а также к таким функциям, как проверка;
- вы также можете использовать сервисы из контейнера DI для настройки объекта `IOptions<T>`, создав отдельный класс, реализующий `IConfigureOptions<T>`. Этот класс может использовать DI в конструкторе и используется для ленивого создания запрошенного объекта `IOptions<T>` во время выполнения;
- вы можете заменить встроенный DI-контейнер сторонним контейнером. Сторонние контейнеры часто предоставляют дополнительные функции, такие как регистрация зависимостей на основе соглашений, сканирование сборок и внедрение свойств.

39

Создание собственных компонентов MVC и Razor Pages

В этой главе:

- создание собственных тег-хелперов Razor;
- использование компонентов представления для создания сложных представлений Razor;
- создание собственного атрибута валидации `DataAnnotations`;
- замена фреймворка валидации `DataAnnotations` альтернативным вариантом.

В предыдущей главе вы узнали, как настроить и расширить некоторые основные системы в ASP.NET Core: конфигурацию, внедрение зависимостей и конвейер промежуточного ПО. Эти компоненты составляют основу всех приложений ASP.NET Core. В данной главе мы сосредоточимся на Razor Pages и контроллерах MVC/API. Вы узнаете, как создавать специальные компоненты, которые работают с представлениями Razor, и компоненты, которые работают с фреймворком валидации, используемым как Razor Pages, так и контроллерами API.

Мы начнем с рассмотрения тег-хеллеров. В разделе 32.1 я покажу вам, как создать два разных тег-хелпера: один, который генерирует

HTML для описания текущей машины, и второй, который позволяет писать инструкции if в шаблонах Razor без использования C#. Все это предоставит вам сведения, необходимые для создания собственных тег-хелперов в ваших приложениях, если возникнет такая необходимость.

В разделе 32.2 вы узнаете о новой концепции Razor: компонентах представления. Компоненты представления немного похожи на частичные представления, но могут содержать бизнес-логику и доступ к базе данных. Например, на сайте для онлайн-торговли у вас может быть корзина покупок, динамически заполняемое меню и виджет входа – и все это на одной странице. Каждый из этих разделов не зависит от содержимого главной страницы и имеет свою логику и потребности в доступе к данным. В приложении ASP.NET Core, использующем Razor Pages, каждый из них реализуется в виде компонента представления.

В разделе 32.3 я покажу, как создать специальный атрибут валидации. Как было показано в главе 6, валидация является важной обязанностью обработчиков страниц Razor и методов действий, а атрибуты `DataAnnotations` обеспечивают ясный декларативный способ сделать это. Ранее мы рассматривали только встроенные атрибуты, но часто вы будете сталкиваться с тем, что вам нужно добавить атрибуты, адаптированные к предметной области вашего приложения. В разделе 32.3 вы увидите, как создать простой атрибут валидации и расширить его для использования сервисов, зарегистрированных в контейнере внедрения зависимостей.

В этой книге я уже упоминал, что при желании можно легко заменить основные части фреймворка ASP.NET Core. В разделе 32.4 вы сделаете это, заменив встроенный фреймворк валидации на основе атрибутов популярной альтернативой `FluentValidation`. Эта библиотека с открытым исходным кодом позволяет отделить модели привязки от правил проверки, что упрощает создание определенной логики валидации. Многие предпочитают данный подход разделения проблем декларативному подходу с использованием `DataAnnotations`.

Когда вы создаете страницы с помощью Razor Pages, одной из лучших функций повышения производительности являются тег-хелперы, и в следующем разделе вы увидите, как создать собственные тег-хелперы.

32.1 Создание собственного тег-хелпера Razor

В этом разделе вы узнаете, как создавать собственные тег-хелперы, позволяющие настраивать вывод HTML. Вы узнаете, как создавать тег-хелперы, которые добавляют новые элементы в разметку HTML, а также тег-хелперы, которые можно использовать для удаления или настройки существующей разметки. Вы также увидите, как ваши тег-хелперы интегрируются с инструментами интегрированной среды разработки, чтобы предоставить богатые возможности IntelliSense таким же образом, что и встроенные тег-хелперы.

На мой взгляд, тег-хелперы – одно из лучших дополнений к языку шаблонов Razor в ASP.NET Core. Они позволяют писать шаблоны

Razor, которые легче читать, поскольку требуют меньше переключения между C# и HTML, и они *дополняют* ваши HTML-теги, а не заменяют их (в отличие от HTML-хелперов, широко использовавшихся в предыдущей версии ASP.NET).

ASP.NET Core обладает широким спектром тег-хелперов (см. главу 8), которые удовлетворят многие из ваших повседневных требований, особенно когда речь идет о создании форм. Например, можно использовать тег-хелпер ввода, добавив атрибут `asp-for` в тег `<input>` и передав ссылку на свойство `PageModel`, в данном случае `Input.Email`:

```
<input asp-for="Input.Email" />
```

Тег-хелпер активируется наличием атрибута и получает возможность дополнить тег `<input>` при отрисовке в HTML. Тег-хелпер использует имя свойства, чтобы задать свойства `name` и `id` тега `<input>`, значение модели, чтобы задать свойство `value`, и наличие таких атрибутов, как `[Required]` или `[EmailAddress]`, чтобы добавить атрибуты для валидации:

```
<input type="email" id="Input_Email" name="Input.Email"
       value="test@example.com" data-val="true"
       data-val-email="The Email Address field is not a valid e-mail address."
       data-val-required="The Email Address field is required."/>
/
```

Тег-хелперы помогают уменьшить дублирование кода или могут упростить распространенные шаблоны. В этом разделе я покажу, как создать собственные тег-хелперы.

В разделе 32.1.1 вы создадите тег-хелпер системной информации, который выводит сведения об имени и операционной системе сервера, на котором работает ваше приложение. В разделе 32.1.2 вы создадите тег-хелпер, который можно использовать для условного отображения или скрытия элемента на основе логического свойства C#. В разделе 32.1.3 вы создадите тег-хелпер, который считывает содержимое Razor, записанное *внутри* тег-хелпера, и преобразовывает его.

32.1.1 Вывод информации об окружении с помощью собственного тег-хелпера

Распространенная проблема, с которой вы можете столкнуться, когда начинаете запускать свои веб-приложения в промышленном окружении, особенно если вы используете группу серверов, – это определить, на каком компьютере отображается страница, которую вы просматриваете в данный момент. Точно так же при частом развертывании может быть полезно знать, какая *версия* приложения работает. При разработке и тестировании мне иногда нравится добавлять небольшой «информационный дамп» в нижней части своих макетов, чтобы можно было с легкостью определить, какой сервер сгенерировал текущую страницу, в каком окружении работает приложение и т. д.

В этом разделе я покажу вам, как создать специальный тег-хелпер для вывода системной информации в свой макет. Вы сможете пере-

ключать отображаемую информацию, но по умолчанию будет отображаться имя компьютера и операционная система, на которой запущено приложение, как показано на рис. 32.1.

Вы можете вызвать этот тег-хелпер из Razor, создав элемент `<system-info>` в своем шаблоне:

```
<footer>
    <system-info></system-info>
</footer>
```

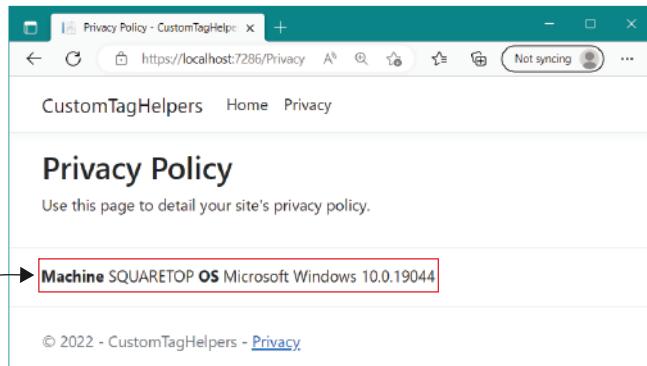


Рис. 32.1 SystemInfoTagHelper отображает имя компьютера и операционную систему, на которой работает приложение. Это может быть полезно, чтобы определить, какой экземпляр приложения обработал запрос, если вы работаете с группой серверов

COBET Вероятно, вы не захотите раскрывать такую информацию в промышленном окружении, поэтому вы также можете заключить ее в тег-хелпер `<environment>`, как было показано в главе 18.

Самый простой способ создать специальный тег-хелпер – наследовать от базового класса `TagHelper` и переопределить функцию `Process()` или `ProcessAsync()`, которая описывает, как класс должен отображать себя.

В следующем листинге показан полный тег-хелпер, `SystemInfoTagHelper`, который отображает системную информацию в тег `<div>`. Вы можете легко расширить этот класс, если хотите отобразить дополнительные поля или добавить дополнительные параметры.

Листинг 32.1 SystemInfoTagHelper для отрисовки системной информации в представление

<p>▷ public class SystemInfoTagHelper : TagHelper Наследуется от базового класса TagHelper</p> <pre> { private readonly HtmlEncoder _htmlEncoder; public SystemInfoTagHelper(HtmlEncoder htmlEncoder) { _htmlEncoder = htmlEncoder; } }</pre>	<p>При записи HTML-содержимого на страницу необходим HtmlEncoder</p>
--	---

```

[HtmlAttributeName("add-machine")]
public bool IncludeMachine { get; set; } = true;

[HtmlAttributeName("add-os")]
public bool IncludeOS { get; set; } = true;

public override void Process(
    TagHelperContext context, TagHelperOutput output)
{
    output.TagName = "div";
    output.TagMode = TagMode.StartTagAndEndTag;
    var sb = new StringBuilder();

    if (IncludeMachine)
    {
        sb.Append(" <strong>Machine</strong> ");
        sb.Append(_htmlEncoder.Encode(Environment.MachineName));
    }
    if (IncludeOS)
    {
        sb.Append(" <strong>OS</strong> ");
        sb.Append(
            _htmlEncoder.Encode(RuntimeInformation.OSDescription));
    }
    output.Content.SetHtmlContent(sb.ToString());
}

```

Основная функция, вызываемая при отрисовке элемента

Заменяет элемент <systeminfo> на элемент <div>

При необходимости добавляет элемент и имя ОС в кодировке HTML

Декорирование свойств с помощью атрибута `[HtmlAttributeName]` позволяет задавать их значения из разметки Razor

Отображает как начальный, так и конечный тег <div> </div>

При необходимости добавляет элемент и имя компьютера в кодировке HTML

Задает внутреннее содержимое тела <div>, используя значение в кодировке HTML, хранящееся в построителе строк

В этом примере много нового кода, поэтому мы будем работать над ним построчно. Вначале имя класса тег-хелпера определяет имя элемента, который вы должны создать в своем шаблоне Razor. У него будет удален суффикс и будет использоваться дефис. Поскольку этот тег-хелпер называется `SystemInfoTagHelper`, то будет создан элемент `<system-info>`.

СОВЕТ Если вы хотите изменить имя элемента, например на `<env-info>`, но хотите сохранить то же имя класса, то можно применить атрибут `[HtmlTargetElement]` с желаемым именем, например `[HtmlTargetElement("Env-Info")]`. HTML-теги не чувствительны к регистру, поэтому можно использовать "Env-Info" или "env-info".

Внедрите `HtmlEncoder` в свой тег-хелпер, чтобы иметь возможность кодировать в HTML любые данные, которые вы пишете на страницу. Как было показано в главе 18, вы всегда должны кодировать в HTML данные, которые пишете на страницу, чтобы избежать XSS-языковых атак (и обеспечить правильное отображение данных).

Вы определили два свойства в тег-хелпере: `IncludeMachine` и `IncludeOS`, которые вы будете использовать, чтобы управлять, какие данные будут записываться на страницу. Они декорированы соответствующим атрибутом `[HtmlAttributeName]`, что позволяет задавать свойства из шаблона

Razor. В Visual Studio вы даже получаете IntelliSense и проверку типов для этих значений, как показано на рис. 32.2.

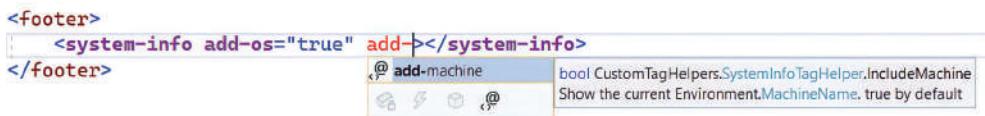


Рис. 32.2 В Visual Studio тег-хелперы выделены фиолетовым цветом, и вы получаете IntelliSense для свойств, декорированных атрибутом [HtmlAttributeName]

Наконец, мы подошли к методу `Process()`. Движок Razor вызывает этот метод, чтобы выполнить тег-хелпер, когда он идентифицирует целевой элемент в шаблоне представления. Метод `Process()` определяет тип тега для отрисовки (`<div>`), нужно ли визуализировать начальный и конечный теги (или самозакрывающийся тег – это зависит от типа тега, который вы визуализируете) и HTML-содержимое тега `<div>`. Вы задаете HTML-содержимое для отрисовки внутри тега, вызывая метод `Content.SetHtmlContent()` в предоставленном экземпляре класса `TagHelperOutput`.

ВНИМАНИЕ Всегда кодируйте свой вывод в HTML перед записью в тег с помощью метода `SetHtmlContent()`. В качестве альтернативы можно передать незакодированный ввод в метод `SetContent()`, и вывод будет автоматически закодирован в HTML.

Прежде чем вы сможете использовать новый тег-хелпер в шаблоне Razor, необходимо зарегистрировать его. Это можно сделать в файле `_ViewImports.cshtml`, используя директиву `@addTagHelper` и указав полное имя тег-хелпера и сборки. Например:

```
@addTagHelper CustomTagHelpers.SystemInfoTagHelper, CustomTagHelpers
```

Кроме того, можно добавить все тег-хелперы из определенной сборки с помощью синтаксиса подстановочных знаков `*` и указав имя сборки:

```
@addTagHelper *, CustomTagHelpers
```

Создав и зарегистрировав собственный тег-хелпер, вы можете использовать его в любом представлении Razor, частичном представлении или макетах.

СОВЕТ Если вы не видите автодополнение для своего тег-хелпера в Visual Studio, а тег-хелпер не отображается полужирным шрифтом, используемым Visual Studio, вероятно, вы неправильно зарегистрировали тег-хелперы в файле `_ViewImports.cshtml`, используя директиву `@addTagHelper`.

`SystemInfoTagHelper` – это пример тег-хелпера, который генерирует содержимое, но вы также можете использовать тег-хелперы для управления отрисовкой существующих элементов. В следующем разделе

вы создадите простой тег-хелпер, который может контролировать, визуализируется элемент или нет, на основе HTML-атрибута.

32.1.2 Создание специального тег-хелпера для условного скрытия элементов

Если вы хотите контролировать, отображается ли элемент в шаблоне Razor на основе переменной C#, то обычно заключаете этот элемент в инструкцию if:

```
@{  
    var showContent = true;  
}  
@if(showContent)  
{  
    <p>The content to show</p>  
}
```

Возврат к подобным конструкциям C# может быть полезным, поскольку позволяет создавать любую разметку, которая вам нравится. К сожалению, переключение между C# и HTML может быть утомительным, а также усложняет использование редакторов HTML, которые не понимают синтаксис Razor.

В этом разделе вы создадите простой тег-хелпер, чтобы избежать подобной проблемы. Вы можете применить его к существующим элементам, чтобы добиться того же результата, как было показано ранее, но без необходимости возвращаться к C#:

```
@{  
    var showContent = true;  
}  
<p if="showContent" >  
    The content to show  
</p>
```

При визуализации во время выполнения этот шаблон Razor будет возвращать HTML-код.

```
<p>  
    The content to show  
</p>
```

Вместо того чтобы создавать новый элемент, как вы это делали для `SystemInfoTagHelper(<systeminfo>)`, вы создадите тег-хелпер, который примените в качестве атрибута к существующим HTML-элементам. Этот тег-хелпер делает одно: контролирует видимость элемента, к которому прикреплен. Если значение, переданное в атрибуте if, истинно, то элемент и его содержимое отображаются как обычно. Если переданное значение ложно, тег-хелпер удаляет элемент и его содержимое из шаблона. Вот как это сделать.

Листинг 32.2 Создание IfTagHelper для условной отрисовки элементов

```
[HtmlTargetElement(Attributes = "if")]
public class IfTagHelper : TagHelper
{
    [HtmlAttributeName("if")]
    public bool RenderContent { get; set; } = true;

    public override void Process(
        TagHelperContext context, TagHelperOutput output)
    {
        if(RenderContent == false)
        {
            output.TagName = null;
            output.SuppressOutput();
        }
    }

    public override int Order => int.MinValue;
}
```

Задавая значение свойства Attributes, вы указываете, что тег-хелпер запускается атрибутом if

Привязывает значение атрибута if к свойству RenderContent

Движок Razor вызывает метод Process() для выполнения тег-хелпера

Если свойство RenderContent имеет значение false, удаляет элемент

Задает для элемента, где находится тег-хелпер, значение null, удаляя его со страницы

Гарантирует, что этот тег-хелпер запускается раньше всех остальных, прикрепленных к элементу

Не отображает и не вычисляет внутреннее содержимое элемента

Вместо отдельного элемента `<if>` движок Razor выполняет `IfTagHelper` всякий раз, когда находит элемент с атрибутом `if`. Его можно применить к любому HTML-элементу: `<p>`, `<div>`, `<input>` и т. д. Вы должны определить логическое свойство, указав, следует ли отображать содержимое, привязанное к значению в атрибуте `if`.

Здесь функция `Process()` намного проще. Если `RenderContent` имеет значение `false`, она задает для `TagHelperOutput.TagName` значение `null`, в результате чего элемент удаляется со страницы. Вы также вызываете метод `SuppressOutput()`, который запрещает отрисовку содержимого *внутри* элемента с атрибутом. Если значение `RenderContent` равно `true`, вы пропускаете эти шаги, и содержимое отображается как обычно.

Еще одно замечание – переопределенное свойство `Order`. Оно контролирует порядок, в котором запускаются тег-хелперы, когда к элементу применено несколько тег-хелперов. Задавая для `Order` значение `int.MinValue`, вы гарантируете, что `IfTagHelper` будет запущен первым, удаляя элемент, если это необходимо, до выполнения других тег-хелперов. Как правило, нет смысла запускать другие тег-хелперы, если элемент все равно будет удален со страницы.

ПРИМЕЧАНИЕ Не забудьте зарегистрировать свои тег-хелперы в файле `_ViewImports.cshtml` с директивой `@addTagHelper`.

С помощью простого HTML-атрибута теперь можно условно отображать элементы в шаблонах Razor без необходимости возвращаться к C#. Этот тег-хелпер может показывать и скрывать содержимое, и ему не нужно знать, что оно из себя представляет. В следующем разделе мы создадим тег-хелпер, которому *необходимо* знать содержимое.

32.1.3 Создание тег-хелпера для преобразования Markdown в HTML

Показанные до сих пор два тег-хелпера не зависят от содержимого, написанного внутри тег-хелпера, но также может быть полезно создавать тег-хелперы, которые проверяют, извлекают и изменяют это содержание. В этом разделе вы увидите пример такого тег-хелпера, который преобразует содержимое Markdown, записанное внутри него, в HTML.

ОПРЕДЕЛЕНИЕ *Markdown* – широко используемый текстовый язык разметки, который легко читается, и его также можно преобразовать в HTML. Этот распространенный формат используется файлами README на GitHub, а я использую его, например, для написания статей в блоге. Для получения более подробной информации о Markdown см. руководство на GitHub: <https://guides.github.com/features/mastering-markdown>.

Мы будем применять популярную библиотеку Markdig (<https://github.com/xoofx/markdig>) для создания тег-хелпера Markdown. Эта библиотека преобразует строку, содержащую Markdown, в строку HTML. Установить Markdig можно с помощью Visual Studio, выполнив команду `dotnet add package Markdig` или добавив `<PackageReference>` в файл с расширением .csproj:

```
<PackageReference Include="Markdig" Version="0.30.4" />
```

Тег-хелпер Markdown, который мы вскоре создадим, можно использовать, добавив элементы `<markdown>` на страницу Razor, как показано в следующем листинге.

Листинг 32.3 Использование тег-хелпера Markdown на странице Razor

```
@page
@model IndexModel
 @{
     var showContent = true;
 }
<markdown>    ← Тег-хелпер Markdown
                  добавляется с помощью
                  элемента <markdown>
## This is a markdown title ← Заголовки можно создать
                                в Markdown, используя символ
                                # для обозначения h1, ## для
                                обозначения h2 и т. д.

This is a markdown list:   ← Markdown преоб-
                           разует простые
                           списки в HTML-
                           элементы <ul>
* Item 1
* Item 2

<div if="showContent">
    Content is shown when showContent is true
</div>
</markdown>   ← Содержимое Razor
                           может быть вло-
                           жено в другие тег-
                           хелперы
```

Тег-хелпер Markdown визуализирует содержимое с помощью следующих шагов.

- 1 Визуализирует любое содержимое Razor внутри тег-хелпера. Сюда входит выполнение любых вложенных тег-хелперов и код C# внутри тег-хелперов. В листинге 32.3, например, используется IfTagHelper.
- 2 Преобразовывает полученную строку в HTML с помощью библиотеки Markdig.
- 3 Заменяет содержимое визуализированным HTML и удаляет элемент <markdown>.

В следующем листинге показан простой подход к реализации тег-хелпера Markdown с использованием Markdig. Markdig поддерживает множество дополнительных расширений и функций, которые вы можете использовать, но общий шаблон тег-хелпера будет таким же.

Листинг 32.4 Реализация класса MarkdownTagHelper с использованием Markdig

```
public class MarkdownTagHelper : TagHelper ← Тег-хелпер Markdown буд-
{                                            дет использовать элемент
    public override async Task ProcessAsync(←
        TagHelperContext context, TagHelperOutput output) ←
    {
        TagHelperContent markdownRazorContent = await ← Получаем содер-
            output.GetChildContentAsync();                жимое элемента
        string markdown = ← <markdowm>
            markdownRazorContent.GetContent();           <markdowm>

        string html = Markdig.Markdown.ToHtml(markdown); ← Преобразовываем
                                                               строку Markdown
                                                               в HTML с помощью
                                                               Markdig

        output.Content.SetHtmlContent(html); ← Пишем содер-
        output.TagName = null; ← жимое HTML
    }                                     в вывод
    }                                     Удаляем элемент <markdowm>
                                         из содержимого
```

При отрисовке в HTML содержимое Markdown из листинга 32.3 (когда значение переменной showContent равно true) становится таким:

```
<h2>This is a markdown title</h2>
<p>This is a markdown list:</p>
<ul>
<li>Item 1</li>
<li>Item 2</li>
</ul>
<div>
    Content is shown when showContent is true
</div>
```

ПРИМЕЧАНИЕ В листинге 32.4 мы реализовали метод ProcessAsync() вместо метода Process(), потому что вызываем асинхронный метод GetChildContentAsync(). Вы должны вызывать асинхронные методы

только из других асинхронных методов; в противном случае вы можете получить такие проблемы, как нехватка потоков. Дополнительные сведения см. в документе Microsoft «Лучшие практики по производительности ASP.NET Core»: <http://mng.bz/KM7X>.

Тег-хелперы из этого раздела представляют собой небольшую выборку возможных путей, которые вы могли бы изучить, но они охватывают две обширные категории: тег-хелперы для отрисовки нового содержимого и тег-хелперы для управления отрисовкой других элементов.

СОВЕТ Дополнительные сведения и примеры см. в документации Microsoft: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/authoring?view=aspnetcore-5.0>.

Тег-хелперы могут быть полезны для предоставления небольших частей изолированной, многократно используемой функциональности, подобной этой, но они не предназначены для предоставления крупных разделов приложения или для выполнения вызовов сервисов бизнес-логики. Вместо этого нужно использовать компоненты представления, как будет показано в следующем разделе.

32.2 Компоненты представления: добавление логики в частичные представления

В этом разделе вы познакомитесь с *компонентами представления*. Компоненты представления работают независимо от основной страницы Razor и могут использоваться для инкапсуляции сложной бизнес-логики. Вы можете использовать компоненты представления, чтобы ваша основная страница Razor была сосредоточена на одной задаче, отрисовке основного содержимого, вместо того чтобы нести ответственность за другие разделы страницы.

Если представить себе типичный веб-сайт, то можно заметить, что часто у него имеется несколько независимых динамических разделов в дополнение к основному содержимому. Рассмотрим, например, сайт Stack Overflow, показанный на рис. 32.3. Помимо основной части страницы, содержащей вопросы и ответы, здесь есть раздел, показывающий пользователя, выполнившего вход, панель для статей в блогах и связанных с этим материалов, а также раздел для вакансий.

Каждый из этих разделов фактически не зависит от основного содержимого. Каждый раздел содержит бизнес-логику (решая, какие статьи или рекламные сообщения нужно показывать), доступ к базе данных (загрузка деталей поста) и логику отрисовки для отображения данных.

В главе 7 вы видели, что можно использовать макеты и частичные представления, чтобы разделить отрисовку шаблона представления на похожие разделы, но частичные представления не подходят для этого примера. Частичные представления позволяют инкапсулировать логику *отрисовки представления*, но не бизнес-логику, независимую от содержимого главной страницы.

Компоненты представления предоставляют эту функциональность, инкапсулируя и бизнес-логику, и логику отрисовки для отображения небольшого раздела страницы. Вы можете использовать внедрение зависимостей для предоставления доступа к контексту базы данных и можете тестировать компоненты представления независимо от представления, которое они генерируют, во многом подобно контроллерам MVC и API. Рассматривайте их как своего рода мини-контроллеры MVC или мини-страницы Razor Pages. Они вызываются непосредственно из представления Razor, а не в ответ на HTTP-запрос.

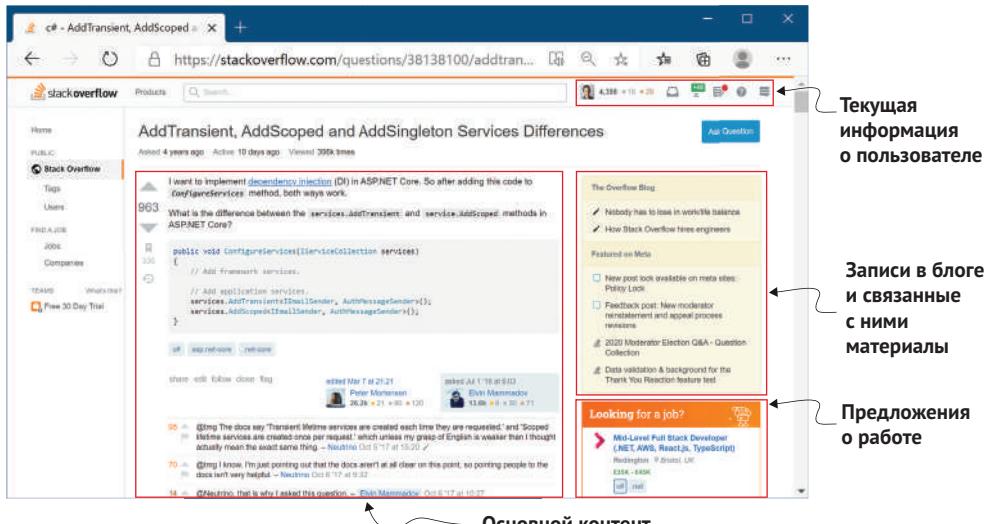


Рис. 32.3 Веб-сайт Stack Overflow имеет несколько разделов, которые не зависят от основного содержимого, но содержат бизнес-логику и сложную логику отрисовки. Каждый из них может отображаться в виде компонента представления в ASP.NET Core

СОВЕТ Компоненты просмотра сопоставимы с *дочерними действиями* из предыдущей версии ASP.NET, поскольку они представляют аналогичную функциональность. В ASP.NET Core до-
нерных действий нет.

Компоненты представления в сравнении с Razor Components и Blazor

В этой книге основное внимание уделено приложениям с отрисовкой на стороне сервера с использованием Razor Pages, и приложениям API, использующим контроллеры веб-API. .NET Core 3.0 представил совершенно новый подход к созданию приложений ASP.NET Core: Blazor. В этой книге он не рассматривается, поэтому я рекомендую прочитать книгу «*Blazor в действии*» Криса Сэйнти (Manning, 2021).

В Blazor есть две модели программирования: на стороне клиента и на стороне сервера, – и там, и там используются компоненты *Blazor* (которые, как ни странно, официально называются компонентами *Razor*).

Компоненты Blazor имеют много параллелей с компонентами представления, но находятся они в совершенно разных мирах. Компоненты Blazor могут легко взаимодействовать друг с другом, но их нельзя использовать с тег-хелперами или компонентами представления, и их сложно комбинировать с формами Razor Page.

Тем не менее если вам нужен «островок» богатой интерактивности на стороне клиента на одной странице Razor, то можно встроить компонент Blazor внутрь страницы Razor, как показано в разделе «Отрисовка компонентов на странице или в представлении с помощью тег-хелпера компонента»: <http://mng.bz/PPen>. Вы также можете использовать компоненты Blazor как способ заменить вызовы AJAX на своих страницах Razor, как показано в записи моего блога: <http://mng.bz/9Mjj>.

Если вам не нужна интерактивность Blazor на стороне клиента, компоненты представления по-прежнему являются лучшим вариантом для изолированных разделов в Razor Pages. Они четко взаимодействуют с вашими страницами, не имеют дополнительных операционных издержек и используют знакомые концепции, такие как макеты, частичные представления и тег-хелперы. Для получения дополнительной информации о том, почему следует продолжать использовать компоненты представления, см. запись в моем блоге «Не заменяйте компоненты представления на компоненты Razor»: <http://mng.bz/1rKq>.

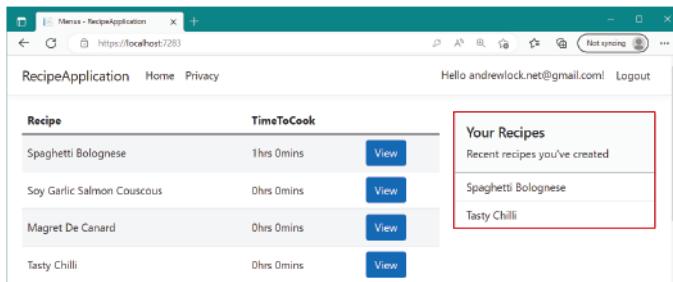
В этом разделе вы узнаете, как создать специальный компонент представления для приложения с рецептами, которое вы создали в предыдущих главах, как показано на рис. 32.4. Если текущий пользователь выполнил вход, компонент представления отображает панель со списком ссылок на недавно созданные рецепты пользователя. Для пользователей, не прошедших аутентификацию, компонент представления отображает ссылки для входа и регистрации.

Этот компонент является отличным кандидатом на роль компонента представления, поскольку он содержит доступ к базе данных и бизнес-логику (выбирая, какие рецепты отображать), а также логику отрисовки (решая, как должна отображаться панель).

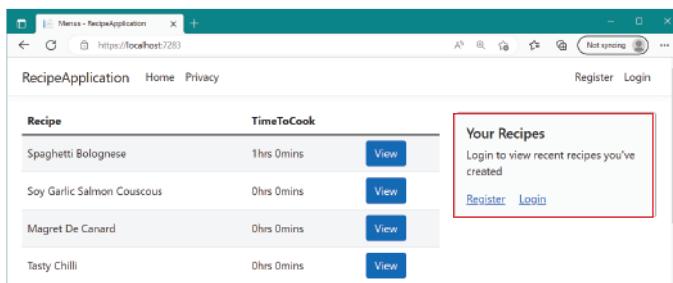
СОВЕТ Используйте частичные представления, если хотите инкапсулировать отрисовку определенной модели представления или ее части. Если у вас есть логика отрисовки, требующая бизнес-логики или доступа к базе данных, или если раздел логически отличается от содержимого главной страницы, рассмотрите возможность использования компонента представления.

Компоненты представления вызываются непосредственно из представлений и макетов Razor с использованием синтаксиса в стиле тег-хелпера с префиксом vc:

```
<vc:my-recipes number-of-recipes="3">
</vc:my-recipes>
```



Когда пользователь вошел в систему, компонент представления отображает список рецептов, созданных текущим пользователем и загруженных из базы данных



Когда пользователь не вошел в систему, компонент представления отображает ссылки на страницы регистрации и входа в систему

Рис. 32.4 Компонент представления отображает различное содержимое в зависимости от текущего пользователя, выполнившего вход. Он включает в себя как бизнес-логику (определяя, какие рецепты загружать из базы данных), так и логику отрисовки (указывающую, как отображать данные)

Специальные компоненты представления обычно наследуют от базового класса `ViewComponent` и реализуют метод `InvokeAsync()`, как показано в листинге 32.5. Наследуя от этого базового класса, вы обеспечиваете доступ к полезным вспомогательным методам почти так же, как если бы вы наследовали от класса `ControllerBase` в случае с контроллерами API.

В отличие от контроллеров API, параметры, передаваемые в `InvokeAsync`, не зависят от привязки модели. Вместо этого вы передаете параметры компоненту представления, используя свойства элемента тег-хелпера в представлении Razor.

Листинг 32.5 Специальный компонент представления для отображения рецептов текущего пользователя

Наследование от базового класса `ViewComponent` предоставляет полезные методы, такие как `View()`

```
public class MyRecipesViewComponent : ViewComponent
{
    private readonly RecipeService _recipeService;
    private readonly UserManager< ApplicationUser > _userManager;
    public MyRecipesViewComponent(RecipeService recipeService,
        UserManager< ApplicationUser > userManager)
    {
        _recipeService = recipeService;
        _userManager = userManager;
    }
}
```

Вы можете использовать внедрение зависимостей в компоненте представления

```

    public async Task<IViewComponentResult> InvokeAsync(
        int numberOfRecipes)
    {
        if (!User.Identity.IsAuthenticated)
        {
            return View("Unauthenticated");
        }

        var userId = _userManager.GetUserId(HttpContext.User);
        var recipes = await _recipeService.GetRecipesForUser(
            userId, numberOfRecipes);

        return View(recipes);
    }
}

Вы можете передавать параметры компоненту из представления
InvokeAsync отображает компонент представления.
Он должен вернуть Task<IViewComponentResult>
Вызов метода View() отобразит частичное представление с указанным именем
Вы можете использовать внешние асинхронные сервисы, что позволяет инкапсулировать логику в вашей предметной области
Вы можете передать модель представления частичному представлению. Файл Default.cshtml используется по умолчанию

```

Данный компонент представления обрабатывает всю логику, необходимую для отрисовки списка рецептов, когда пользователь выполнил вход, или другого представления, если пользователь не аутентифицирован. Имя компонента представления происходит от имени класса, как и в случае тег-хелперов. В качестве альтернативы можно применить к классу атрибут [ViewComponent] и задать совершенно другое имя.

Метод `InvokeAsync` должен вернуть `Task<IViewComponentResult>`. Это похоже на то, как вы возвращаете `IActionResult` из метода действия или обработчика страницы, но более ограничено; компоненты представления должны визуализировать какое-то содержимое, поэтому нельзя возвращать коды состояния или переадресацию. Обычно для отрисовки частичного шаблона представления используется вспомогательный метод `View()` (как в предыдущем листинге), хотя можно также возвращать строку напрямую с помощью вспомогательного метода `Content()`, который кодирует содержимое в HTML и напрямую отображает его на странице.

Методу `InvokeAsync` можно передавать любое количество параметров. Названия параметров (в данном случае `numberOfRecipes`) преобразуются в формат с дефисами и доступны в виде свойств в тег-хелпере компонента представления (`<number-of-recipes>`). Вы можете предоставить эти параметры, когда вызываете компонент представления из представления, и получите поддержку IntelliSense, как показано на рис. 32.5.



Рис. 32.5 Visual Studio обеспечивает поддержку IntelliSense для параметров метода компонента представления `InvokeAsync`. Имя параметра, в данном случае `numberOfRecipes`, теперь пишется с использованием дефисов, чтобы его можно было применить в качестве атрибута в тег-хелпере

Компоненты представления имеют доступ к текущему запросу и `HttpContext`. В листинге 32.5 видно, что мы проверяем, поступил ли текущий запрос от пользователя, прошедшего аутентификацию. Также видно, что мы использовали некую условную логику: если пользователь не прошел аутентификацию, то отображается шаблон Razor «`Unauthenticated`»; если он прошел аутентификацию, то отображается шаблон Razor по умолчанию, и мы передаем модели представления, загруженные из базы данных.

ПРИМЕЧАНИЕ Если вы не укажете конкретный шаблон представления Razor для использования в функции `View()`, компоненты представления будут использовать имя шаблона «`Default.cshtml`».

Частичные представления для компонентов представления работают аналогично другим частичным представлениям Razor, о которых вы узнали в главе 7, но хранятся отдельно. Нужно создавать частичные представления для компонентов представления в одном из следующих мест:

- `Views/Shared/Components/ComponentName/TemplateName`;
- `Pages/Shared/Components/ComponentName/TemplateName`.

Оба места подходят, поэтому для приложений Razor Pages я обычно использую папку `Pages/`. Для компонента представления из листинга 32.5, например, вы должны создать свои шаблоны представления:

- `Pages/Shared/Components/MyRecipes/Default.cshtml`;
- `Pages/Shared/Components/MyRecipes/Unauthenticated.cshtm`.

Это было краткое введение в компоненты представления, но оно должно вам помочь. Компоненты представления – простой способ встроить в систему изолированную сложную логику в свои макеты Razor. Однако при этом помните о следующих предостережениях:

- классы компонентов представления должны быть открытыми, невложенным и неабстрактными;
- хотя они и похожи на контроллеры MVC, нельзя использовать фильтры с компонентами представления;
- вы можете использовать макеты в представлениях своих компонентов представления для извлечения логики отрисовки, общей для определенного компонента. Этот макет может содержать `@sections`, как было показано в главе 7, но данные секции не зависят от макета основного представления Razor;
- компоненты представления изолированы от страницы Razor, на которой они отображаются, поэтому нельзя, например, определить `@section` в макете страницы Razor, а затем добавить это содержимое из компонента представления; контексты полностью разделены;
- при использовании синтаксиса тег-хелпера `<vc:my-recipes>` для вызова компонента представления нужно импортировать его в качестве специального тег-хелпера, как было показано в разделе 32.1;
- вместо использования синтаксиса тег-хелпера можно вызвать компонент представления из представления напрямую с помощью компонента `IViewComponentHelper Component`, хотя я не рекомендую использовать этот синтаксис. Например:

```
@await Component.InvokeAsync("MyRecipes", new { number0fRecipes = 3 })
```

Мы рассмотрели тег-хелперы и компоненты представления, которые являются функциями Razor в ASP.NET Core. В следующем разделе вы узнаете о другой, но близкой теме: как создать собственный атрибут `Data-Annotations`. Если вы использовали предыдущие версии ASP.NET, то это будет вам знакомо, но в ASP.NET Core есть несколько хитростей, которые могут быть вам полезны.

32.3 Создание собственного атрибута валидации

В этом разделе вы узнаете, как создать специальный атрибут валидации `DataAnnotations`, указывающий конкретные значения, которые может принимать строковое свойство. Затем вы узнаете, как расширить эту функциональность, чтобы сделать ее более универсальной, делегировав ее отдельному сервису, который настроен в вашем контроллере внедрения зависимостей. Это позволит вам создавать собственные проверки для конкретных предметных областей в своих приложениях.

Мы рассматривали привязку модели в главе 6, где вы узнали, как использовать встроенные атрибуты `DataAnnotations` в своих моделях привязки для проверки ввода данных пользователем. Они предоставляют несколько встроенных проверок:

- `[Required]` – свойство является обязательным и должно быть предоставлено;
- `[StringLength(min, max)]` – длина строкового значения должна находиться в пределах между `min` и `max`;
- `[EmailAddress]` – значение должно иметь допустимый формат адреса электронной почты.

Но что, если эти атрибуты не отвечают вашим требованиям? Рассмотрим следующий листинг, где показана модель привязки из конвертера валют. Модель содержит три свойства: валюта, из которой нужно конвертировать, валюта, в которую нужно конвертировать, и количество.

Листинг 32.6 Начальная модель привязки конвертера валют

```
public class CurrencyConverterModel
{
    > [Required]
    > [StringLength(3, MinimumLength = 3)]
    public string CurrencyFrom { get; set; }

    > [Required]
    > [StringLength(3, MinimumLength = 3)]
    public string CurrencyTo { get; set; }

    > [Required]
    > [Range(1, 1000)]
    public decimal Quantity { get; set; }
}
```

Все
свойства
обяза-
тельны

Строки должны
содержать ровно
3 символа

Количество может
составлять от 1 до 1000

Эта модель проходит базовую валидацию, но во время тестирования обнаруживается проблема: пользователи могут ввести любую трехбуквенную строку для свойств `CurrencyFrom` и `CurrencyTo`. Пользователи должны иметь возможность выбирать только действительный код валюты, например "USD" или "GBP", но какой-нибудь злоумышленник может легко отправить "XXX" или "£\$%".

Предполагая, что вы поддерживаете ограниченный набор валют, например фунт стерлингов, доллар США, евро и канадский доллар, можно выполнить проверку несколькими способами. Один из способов – проверить значения `CurrencyFrom` и `CurrencyTo` в методе обработчика страницы Razor, после того как валидации привязки модели и атрибутов уже произошли.

Еще один способ – использовать атрибут `[RegularExpression]` для поиска разрешенных строк. Подход, который я буду использовать здесь, заключается в создании специального атрибута `ValidationAttribute`. Цель состоит в том, чтобы иметь в своем распоряжении специальный атрибут проверки, который можно применить к свойствам `CurrencyFrom` и `CurrencyTo`, чтобы ограничить диапазон допустимых значений. Это выглядит примерно так, как показано в следующем примере.

Листинг 32.7 Применение специальных атрибутов валидации к модели привязки

```
public class CurrencyConverterModel
{
    [Required]
    [StringLength(3, MinimumLength = 3)]
    [CurrencyCode("GBP", "USD", "CAD", "EUR")]
    public string CurrencyFrom { get; set; } ←
                                                CurrencyCodeAttribute
                                                проверяет, что у свой-
                                                ства одно из указан-
                                                ных значений

    [Required]
    [StringLength(3, MinimumLength = 3)]
    [CurrencyCode("GBP", "USD", "CAD", "EUR")]
    public string CurrencyTo { get; set; } ←

    [Required]
    [Range(1, 1000)]
    public decimal Quantity { get; set; }
}
```

Создать специальный атрибут валидации просто; можно начать с базового класса `ValidationAttribute` и переопределить только один метод. В следующем листинге показано, как реализовать класс `CurrencyCodeAttribute`, чтобы гарантировать, что предоставленные коды валют соответствуют ожидаемым значениям.

Листинг 32.8 Специальный атрибут валидации для кодов валют

Наследуется от ValidationAttribute, чтобы гарантировать, что ваш атрибут используется во время валидации

```

public class CurrencyCodeAttribute : ValidationAttribute
{
    private readonly string[] _allowedCodes;
    public CurrencyCodeAttribute(params string[] allowedCodes)
    {
        _allowedCodes = allowedCodes;
    }
    protected override ValidationResult IsValid(
        object value, ValidationContext context)
    {
        if(value is not string code
            || !_allowedCodes.Contains(code))
        {
            return new ValidationResult("Not a valid currency code");
        }
        return ValidationResult.Success; <-->
    }
}

```

Атрибут принимает массив разрешенных кодов валют

В метод IsValid передается значение для валидации и объект контекста

Если предоставленное значение не является строкой, имеет значение null или недопустимый код, возвращается ошибка

В противном случае возвращаем успешный результат

Валидация выполняется в конвейере фильтров действий после привязки модели до выполнения действия или обработчика страницы Razor. Фреймворк валидации вызывает метод `IsValid()` для каждого экземпляра `ValidationAttribute` проверяемого свойства модели и передает `value` (значение проверяемого свойства) и `ValidationContext` к каждому атрибуту по очереди. Объект контекста содержит детали, которые можно использовать для валидации свойства.

Особо следует отметить свойство `ObjectInstance`. Его можно использовать для доступа к проверяемой модели верхнего уровня при проверке вложенного свойства. Например, если проверяется свойство `CurrencyFrom` класса `CurrencyConverterModel`, можно получить доступ к объекту верхнего уровня из `ValidationAttribute` следующим образом:

```
var model = validationContext.ObjectInstance as CurrencyConverterModel;
```

Это может быть полезно, если допустимость свойства зависит от значения другого свойства модели. Например, вам может потребоваться правило проверки, в котором указано, что GBP является допустимым значением для `CurrencyTo`, за исключением случаев, когда `CurrencyFrom` также является GBP. `ObjectInstance` упрощает подобные проверки-сравнения.

ПРИМЕЧАНИЕ Хотя использование `ObjectInstance` упрощает создание подобных сравнений на уровне модели, это снижает переносимость атрибута валидации. В этом случае вы сможете использовать только атрибут в приложении, которое определяет `CurrencyConverterModel`.

В методе `IsValid` можно привести `value` к требуемому типу данных (в данном случае это строка) и проверить список разрешенных кодов. Если данного кода нет в списке, атрибут возвращает `ValidationResult` с сообщением об ошибке, указывающим на то, что возникла проблема. Если код разрешен, возвращается `ValidationResult.Success`, и проверка успешно завершается.

Тестирование нашего атрибута на рис. 32.6 показывает, что если `CurrencyTo` имеет недопустимое значение (`£$%`), то валидация свойства заканчивается неудачей, и в `ModelState` добавляется ошибка. Вы могли бы немного подправить этот атрибут, чтобы можно было задать собственное сообщение, дабы разрешить пустые значения или отобразить имя недействительного свойства, но все важные функции здесь есть.

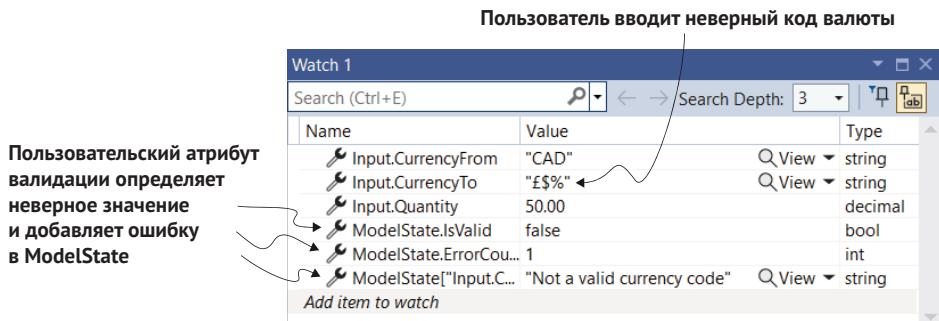


Рис. 32.6 Окно Watch в Visual Studio, где показан результат проверки с использованием `ValidationAttribute`. Пользователь указал недопустимое значение `currencyTo`, £\$%. Следовательно, `ModelState` недействителен и содержит единственную ошибку с сообщением «Недействительный код валюты»

Основная функция, отсутствующая в этом специальном атрибуте, – валидация на стороне клиента. Вы видели, что атрибут хорошо работает на стороне сервера, но если пользователь ввел недопустимое значение, он не получит соответствующую информацию до тех пор, пока недопустимое значение не будет отправлено на сервер. Это безопасно, и этого достаточно для обеспечения безопасности и согласованности данных, но проверка на стороне клиента может улучшить опыт взаимодействия с пользовательским интерфейсом, обеспечивая незамедлительную обратную связь.

Проверку на стороне клиента можно реализовать несколькими способами, но в значительной мере она зависит от библиотек JavaScript, которые вы используете для обеспечения функциональности. В настоящее время при выполнении валидации на стороне клиента шаблоны Razor ASP.NET Core полагаются на jQuery. См. раздел «Специальная валидация на стороне клиента» документа Microsoft «Валидация модели в ASP.NET Core MVC и Razor Pages», где приводится пример создания адаптера валидации jQuery для атрибутов: <http://mng.bz/Wd6g>.

СОВЕТ Вместо использования официальных проверочных библиотек на основе jQuery вы можете использовать библиотеку

aspnet-client-validation с открытым исходным кодом (<https://github.com/haacked/aspnet-client-validation>), как я описываю в своем блоге по адресу <http://mng.bz/AoXe>.

Еще одно усовершенствование для вашего атрибута валидации – загрузка списка валют из сервиса внедрения зависимостей, например ICurrencyProvider. К сожалению, вы не можете использовать конструктор для внедрения зависимостей в CurrencyCodeAttribute, поскольку в .NET в конструктор атрибута можно передавать только *постоянные* значения. В главе 22 мы обошли это ограничение для фильтров, используя атрибуты [TypeFilter] или [ServiceFilter], но для ValidationAttribute такого решения нет.

Для атрибутов валидации нужно использовать паттерн *Локатор сервисов*. Как уже обсуждалось в главе 9, по возможности лучше избегать применения этого антипаттерна, но, к сожалению, в данном случае это необходимо. Вместо объявления явной зависимости через конструктор нужно напрямую запросить у контейнера внедрения зависимостей экземпляр требуемого сервиса.

В листинге 32.9 показано, как переписать листинг 32.8, чтобы загрузить допустимые валюты из экземпляра ICurrencyProvider, вместо того чтобы жестко зашивать допустимые значения в код в конструкторе атрибута. Атрибут вызывает метод GetService<T>() в ValidationContext для получения экземпляра ICurrencyProvider из контейнера внедрения зависимостей. Обратите внимание, что ICurrencyProvider – это гипотетический сервис, который необходимо зарегистрировать в методе ConfigureServices() приложения в файле Startup.cs.

Листинг 32.9 Использование паттерна Локатор сервисов для доступа к сервисам

```
public class CurrencyCodeAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(
        object value, ValidationContext context)
    {
        var provider = context
            .GetRequiredService<ICurrencyProvider>();
        var allowedCodes = provider.GetCurrencies();

        if(value is not string code
           || !_allowedCodes.Contains(code))
        {
            return new ValidationResult("Not a valid currency code");
        }
        return ValidationResult.Success;
    }
}
```

СОВЕТ Обобщенный метод GetRequiredService<T> – это метод расширения, доступный в пространстве имён Microsoft.Extensions.DependencyInjection.

В качестве альтернативы можно использовать метод `GetService(Type type)`. Система валидации на основе `DataAnnotations`, используемая по умолчанию, может быть удобна благодаря своему декларативному характеру, но иногда с ней приходится идти на компромиссы, как показано выше с проблемой внедрения зависимостей. К счастью, можно полностью заменить систему валидации, используемую вашим приложением, как показано в следующем разделе.

32.4 Замена фреймворка валидации на `FluentValidation`

В этом разделе вы узнаете, как заменить фреймворк проверки на основе `DataAnnotations`, который используется по умолчанию в ASP.NET Core. Вы увидите аргументы в пользу этого и узнаете, как использовать стороннюю альтернативу: `FluentValidation`. Этот проект с открытым исходным кодом позволяет определять требования к валидации моделей отдельно от самих моделей. Такое разделение может упростить некоторые типы проверки и гарантировать, что каждый класс в вашем приложении имеет единственную ответственность.

Валидация – важная часть процесса привязки модели в ASP.NET Core. В главе 7 вы узнали, что минимальные API не имеют встроенной проверки, поэтому вы можете выбирать любую платформу, которая вам нравится. Я продемонстрировал использование `DataAnnotations`, но вы можете легко выбрать другой фреймворк проверки.

Однако в Razor Pages и MVC фреймворк валидации `DataAnnotations` встроен в ASP.NET Core. Вы можете применить атрибуты `DataAnnotations` к свойствам ваших моделей привязки, чтобы определить ваши требования, и ASP.NET Core автоматически проверит их. В разделе 32.3 мы даже создали специальный атрибут валидации.

Но ASP.NET Core очень гибкий и позволяет при желании заменять целые части фреймворка. Система валидации – одна из таких областей, которую многие предпочитают заменять.

`FluentValidation` (<https://fluentvalidation.net/>) – популярный альтернативный фреймворк валидации для ASP.NET Core. Это давно существующая библиотека, корни которой уходят далеко в прошлое, еще до появления ASP.NET Core. С `FluentValidation` код валидации пишется отдельно от кода модели привязки. Это дает ряд преимуществ:

- вы не ограничены недостатками атрибутов, такими как проблема внедрения зависимостей, которую нам пришлось решать в листинге 32.9;
- намного проще создать правила проверки, которые применяются к нескольким свойствам, например чтобы гарантировать, что свойство `EndDate` содержит более позднее значение, чем свойство `Start-Date`. Добиться этого с помощью атрибутов `DataAnnotations` можно, но сложно;
- как правило, тестировать валидаторы `FluentValidation` проще, чем атрибуты `DataAnnotations`;

- валидация строго типизирована, по сравнению с атрибутами Data-Annotations, где атрибуты можно применять бессмысленными способами, например применять атрибут [EmailAddress] к свойству типа int;
- отделение логики валидации от самой модели, возможно, лучше соответствует принципу единственной ответственности.

Последний момент часто приводится как причина не использовать FluentValidation: FluentValidation отделяет модель привязки от правил валидации. Некоторые рады принять ограничения DataAnnotations, чтобы не разделять модель и правила валидации.

Прежде чем я покажу, как добавить FluentValidation в приложение, посмотрим, как выглядят валидаторы FluentValidation.

32.4.1 Сравнение FluentValidation и атрибутов DataAnnotations

Чтобы лучше понять разницу между DataAnnotations и FluentValidation, мы преобразуем модели привязки из раздела 32.3 для использования FluentValidation. В следующем листинге показано, как будет выглядеть модель привязки из листинга 32.7 при использовании FluentValidation. С точки зрения структуры она идентична, но у нее нет атрибутов проверки.

Листинг 32.10 Начальная модель привязки конвертера валют для использования с FluentValidation

```
public class CurrencyConverterModel
{
    public string CurrencyFrom { get; set; }
    public string CurrencyTo { get; set; }
    public decimal Quantity { get; set; }
}
```

В FluentValidation правила валидации определяются в отдельном классе, на каждую модель по классу. Обычно они наследуются от базового класса `AbstractValidator<>`, который предоставляет набор методов расширения для определения правил валидации.

В следующем листинге показан валидатор для `CurrencyConverterModel`, который соответствует валидациям, добавленным с использованием атрибутов в листинге 32.7. Вы создаете набор правил проверки для свойства, вызывая метод `RuleFor()` и связывая вызовы методов, например `NotEmpty()`, из него. Данный стиль объединения методов называется «плавным» интерфейсом, отсюда и название.

Листинг 32.11 Валидатор FluentValidation для модели привязки конвертера валют

```
public class CurrencyConverterModelValidator : AbstractValidator<CurrencyConverterModel>
{
    private readonly string[] _allowedValues
        = new []{ "GBP", "USD", "CAD", "EUR" };

```

Валидатор наследуется от AbstractValidator

Определяет статический список поддерживаемых кодов валют

```

    public CurrencyConverterModelValidator()
    {
        RuleFor(x => x.CurrencyFrom)
            .NotEmpty()
            .Length(3)
            .Must(value => _allowedValues.Contains(value))
            .WithMessage("Not a valid currency code");

        RuleFor(x => x.CurrencyTo)
            .NotEmpty()
            .Length(3)
            .Must(value => _allowedValues.Contains(value))
            .WithMessage("Not a valid currency code");

        RuleFor(x => x.Quantity)
            .NotNull()
            .InclusiveBetween(1, 1000);
    }
}

```

Вы определяете правила валидации в конструкторе валидатора

Существуют эквивалентные правила для основных атрибутов валидации DataAnnotations

Метод RuleFor используется для добавления нового правила валидации. Синтаксис лямбда-функции позволяет использовать строгую типизацию

Вы можете легко добавлять собственные правила валидации, не создавая отдельных классов

Благодаря строгой типизации доступные правила зависят от проверяемого свойства

Впервые увидев этот код, вы можете подумать, что он не отличается компактностью по сравнению с листингом 32.7, но помните, что в листинге 32.7 использовался специальный атрибут валидации, [Currency-Code]. Валидации в листинге 32.11 больше ничего не требует – логика, реализованная атрибутом [CurrencyCode], находится прямо там, в валидаторе, благодаря чему проще рассуждать о поведении кода. Метод Must() может использоваться для выполнения произвольно сложных проверок без дополнительных уровней косвенности, необходимых для специальных атрибутов DataAnnotations.

Кроме того, вы заметите, что можете определять только те правила валидации, которые имеют смысл для проверяемого свойства. Раньше ничто не мешало нам применить атрибут [CurrencyCode] к свойству Quantity; с FluentValidation это просто невозможно.

Конечно, тот факт, что вы *можете* написать собственную логику [CurrencyCode] в режиме реального времени, не обязательно означает, что вы должны это делать. Если правило используется в нескольких частях приложения, возможно, имеет смысл извлечь его во вспомогательный класс. В следующем листинге показано, как извлечь логику кода валюты в метод расширения, который может использоваться в нескольких валидаторах.

Листинг 32.12 Метод расширения для проверки валюты

```

public static class ValidationExtensions
{
    public static IRuleBuilderOptions<T, string>
        MustBeCurrencyCode<T>(
            this IRuleBuilder<T, string> ruleBuilder)
    {
    }
}

```

Создает метод расширения, который можно связать с методом RuleFor() для свойств со строковым типом

```

    return ruleBuilder
        .Must(value => _allowedValues.Contains(value))
        .WithMessage("Not a valid currency code");
}

private static readonly string[] _allowedValues = 
    new []{ "GBP", "USD", "CAD", "EUR" };
}

```

Применяет ту же логику валидации, что и раньше

Допустимые значения кодов валют

Затем можно обновить `CurrencyConverterModelValidator`, чтобы использовать новый метод расширения, удалив дублирование кода в валидаторе и обеспечив согласованность между полями для кодов стран:

```

RuleFor(x => x.CurrencyTo)
    .NotEmpty()
    .Length(3)
    .MustBeCurrencyCode();

```

Еще одно преимущество `FluentValidation` при использовании автономных классов валидации заключается в том, что они создаются с использованием внедрения зависимостей, поэтому вы можете внедрять в них сервисы. В качестве примера рассмотрим атрибут валидации `[CurrencyCode]` из листинга 32.9, который использовал сервис `ICurrencyProvider` из контейнера внедрения зависимостей. Он требует применения паттерна *Локатор сервисов* для получения экземпляра `ICurrencyProvider` с использованием внедренного объекта контекста.

С помощью библиотеки `FluentValidation` можно просто вставить `ICurrencyProvider` прямо в свой валидатор, как показано в следующем листинге. Это требует меньшего количества усилий, чтобы получить желаемую функциональность, и делает зависимости валидатора явными.

Листинг 32.13 Валидатор конвертера валют, использующий внедрение зависимостей

```

public class CurrencyConverterModelValidator
    : AbstractValidator<CurrencyConverterModel>
{
    public CurrencyConverterModelValidator(ICurrencyProvider provider) <-->
    {
        RuleFor(x => x.CurrencyFrom)
            .NotEmpty()
            .Length(3)
            .Must(value => provider
                .GetCurrencies()
                .Contains(value))
            .WithMessage("Not a valid currency code");

        RuleFor(x => x.CurrencyTo)
            .NotEmpty()
            .Length(3)
            .MustBeCurrencyCode(provider.GetCurrencies());
    }
}

```

Внедрение сервиса с использованием стандартного внедрения зависимостей через конструктор

Использование внедренного сервиса

Использование внедренного сервиса с методом расширения

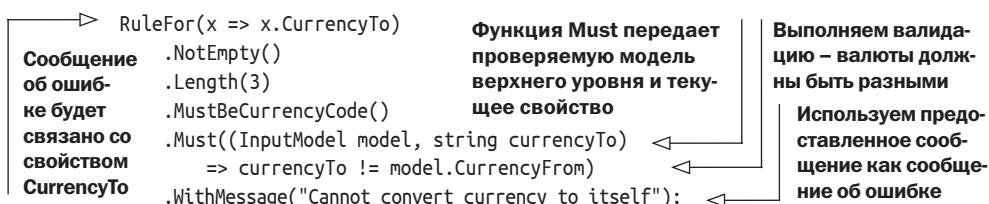
```

        RuleFor(x => x.Quantity)
            .NotNull()
            .InclusiveBetween(1, 1000);
    }
}

```

Последняя функция, которую я покажу, демонстрирует, насколько проще писать валидаторы, охватывающие несколько свойств, с помощью FluentValidation. Например, представьте, что мы хотим проверить, что значение `CurrencyTo` отличается от `CurrencyFrom`. Используя FluentValidation, это можно реализовать с помощью перегруженного варианта метода `Must()`, который предоставляет и модель, и проверяющее свойство, как показано в следующем листинге.

Листинг 32.14 Использование метода `Must()`, чтобы проверить, что два свойства различаются



Создание такого валидатора, безусловно, возможно и с атрибутами `DataAnnotations`, но для этого требуется гораздо больше церемоний, чем с FluentValidation, и, как правило, его труднее тестировать. В FluentValidation есть еще много функций, облегчающих написание и тестирование валидаторов:

- *валидация сложных свойств* – валидаторы могут применяться к сложным типам, а также к примитивным типам, таким как `string` и `int`, показанным здесь в этом разделе;
- *специальные валидаторы свойств* – помимо простых методов расширения можно создавать собственные валидаторы свойств для сложных сценариев проверки;
- *правила коллекций* – когда типы содержат коллекции, такие как `List<T>`, вы можете применить валидацию к каждому элементу списка, а также ко всей коллекции;
- *наборы правил* – можно создать несколько наборов правил, которые могут применяться к объекту в различных обстоятельствах. Это может быть особенно полезно, если вы используете FluentValidation в дополнительных областях своего приложения;
- *валидация на стороне клиента* – FluentValidation – это серверный фреймворк, но он генерирует те же атрибуты, что и атрибуты `DataAnnotations`, для активации проверки на стороне клиента, используя `jQuery`.

В дополнение к этим функциям есть еще и много других, поэтому обязательно просмотрите документацию на странице <https://docs>.

fluentvalidation.net/ для получения подробной информации. В следующем разделе вы увидите, как добавить FluentValidation в приложение ASP.NET Core.

32.4.2 Добавляем FluentValidation в приложение

Замена всей системы валидации ASP.NET Core кажется серьезным шагом, но библиотеку FluentValidation легко добавить в свое приложение. Просто выполните следующие действия.

- 1 Установите NuGet-пакет FluentValidation.AspNetCore с помощью диспетчера пакетов NuGet Visual Studio через интерфейс командной строки, выполнив команду `dotnet add package Fluent-Validation.AspNetCore` или добавив `<PackageReference>` в свой файл с расширением `.csproj`:

```
<PackageReference Include="FluentValidation.AspNetCore" Version="11.2.2" />
```

- 2 Настройте библиотеку FluentValidation в файле `Program.cs`, вызвав метод `builder.Services.AddFluentValidationAutoValidation()`. Вы можете дополнительно настроить библиотеку, как показано в листинге 32.15.
- 3 Зарегистрируйте валидаторы (например, `CurrencyConverterModelValidator` из листинга 32.13) в контейнере внедрения зависимостей. Их можно зарегистрировать вручную, используя любую область действия, которую вы выберете:

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages();
builder.Services.AddFluentValidationAutoValidation();
builder.Services.AddScoped<
    IValidator<CurrencyConverterModel>,
    CurrencyConverterModelValidator>();
```

В качестве альтернативы вы можете разрешить FluentValidation автоматически регистрировать все свои валидаторы, используя параметры, показанные в листинге 32.15.

Будучи зрелой библиотекой, FluentValidation имеет относительно мало параметров конфигурации. В следующем листинге показаны некоторые из доступных опций; в частности, показано, как автоматически зарегистрировать все специальные валидаторы в приложении и как полностью отключить валидацию на основе `DataAnnotations`.

Листинг 32.15 Настройка FluentValidation в приложении ASP.NET Core

```
var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddRazorPages();
```

Вместо того чтобы регистрировать валидаторы вручную, FluentValidation может делать это автоматически за вас

```
builder.Services.AddValidatorsFromAssemblyContaining<Program>();
```

```
Установка значения true полностью отключает
проверку DataAnnotations для привязки модели
builder.Services.AddFluentValidationAutoValidation(
    x => x.DisableDataAnnotationsValidation = true)
    .AddFluentValidationClientsideAdapters();

ValidatorOptions.Global.LanguageManager.Enabled = false;
```

Включает интеграцию с клиентской валидацией при помощи data*-атрибутов

FluentValidation
имеет полную поддержку локализации, но вы можете отключить ее, если она вам не нужна

Важно понять тот момент, что если вы не зададите для `DisableDataAnnotationsValidation` значение `true`, ASP.NET Core выполнит валидацию, используя `DataAnnotations`, и `Fluent-Validation`. Это может быть полезно, если вы находитесь в процессе перехода с одной системы на другую, но в противном случае я рекомендую отключить ее. Такое объединение валидации берет худшее из обоих вариантов!

И последнее, о чём следует подумать, – где разместить валидаторы. Здесь нет технических требований: если вы зарегистрировали свой валидатор в контейнере внедрения зависимостей, он будет использоваться правильно, поэтому выбор остается за вами. Лично я предпочитаю размещать валидаторы рядом с моделями, которые они проверяют.

Для валидаторов модели привязки Razor Pages я создаю валидатор в виде вложенного класса `PageModel` там же, где создаю `InputModel`, как было описано в главе 16. Это дает иерархию классов на странице Razor, подобную этой:

```
public class IndexPage : PageModel
{
    public class InputModel { }
    public class InputModelValidator: AbstractValidator<InputModel> { }
}
```

Конечно, это всего лишь мое предпочтение. При желании можно выбрать другой подход.

На этом мы подошли к концу главы, посвященной специальным компонентам Razor Pages. В сочетании с компонентами из предыдущей главы вы получаете отличную основу для расширения приложений ASP.NET Core в соответствии со своими потребностями. Это свидетельство дизайна ASP.NET Core: вы можете полностью менять местами целые разделы, например фреймворк валидации. Если вам не нравится, как работает какая-то часть фреймворка, посмотрите, не написал ли кто-нибудь альтернативный вариант!

Резюме

- С помощью тег-хелперов можно привязать свою модель данных к элементам HTML, облегчая генерацию динамического HTML. Тег-хелперы могут настраивать элементы, к которым они прикрепляются, добавлять дополнительные атрибуты и настраивать

способ их отрисовки в HTML. Это может значительно уменьшить количество разметки, которую вам нужно писать;

- имя класса тег-хелпера определяет имя элемента в шаблонах Razor, поэтому `SystemInfoTagHelper` соответствует элементу `<system-info>`. Вы можете выбрать другое имя элемента, добавив атрибут `[HtmlTargetElement]` к своему тег-хелперу;
- вы можете задать свойства объекта тег-хелпера из синтаксиса Razor, декорировав свойство атрибутом `[HtmlAttributeName("name")]` и предоставив имя. Можно задать эти свойства из Razor, используя атрибуты HTML, например `<system-info name = "value">`;
- параметр `TagHelperOutput`, передаваемый методам `Process` или `ProcessAsync`, управляет HTML, отображаемым на странице. Можно задать тип элемента с помощью свойства `TagName` и внутреннее содержимое с помощью `Content.SetContent()` или `Content.SetHtmlContent()`;
- вы можете предотвратить обработку внутреннего содержимого тег-хелпера, вызвав метод `SuppressOutput()`, и можете полностью удалить элемент, задав для свойства `TagName` значение `null`. Это полезно, если вы хотите условно отображать элементы;
- содержимое тег-хелпера можно получить, вызвав метод `GetChildContentAsync()` параметра `TagHelperOutput`. Затем можно визуализировать это содержимое в строку, вызвав метод `GetContent()`. Так вы визуализируете все выражения Razor и тег-хелперы в HTML, позволив себе управлять содержимым;
- компоненты представления похожи на частичные представления, но они позволяют использовать сложную бизнес-логику и логику отрисовки. Вы можете использовать их для разделов страницы, таких как корзина покупок, динамическое меню навигации или рекомендуемые статьи;
- создайте компонент представления, наследуя от базового класса `ViewComponent` и реализовав метод `InvokeAsync()`. Вы можете передать параметры этой функции из шаблона представления Razor, используя атрибуты HTML, аналогично тег-хелперам;
- компоненты представления могут использовать внедрение зависимостей, обращаться к `HttpContext` и отображать частичные представления. Частичные представления должны храниться в папке `Pages/Shared/Components/<Name>/`, где `Name` – это имя компонента представления. Если не указано иное, компоненты представления будут искать представление по умолчанию с именем `Default.cshtml`;
- вы можете создать собственный атрибут `DataAnnotations`, наследуя от класса `ValidationAttribute` и переопределив метод `IsValid`. Его можно использовать, чтобы декорировать свойства своей модели привязки и выполнить произвольную проверку;
- нельзя использовать внедрение зависимостей через конструктор со специальными атрибутами валидации. Если атрибуту валидации требуется доступ к службам из контейнера внедрения

зависимостей, необходимо использовать паттерн *Локатор сервисов*, чтобы получить их из контекста валидации, используя метод `GetService<T>`;

- `FluentValidation` – это альтернативная система валидации, которая может заменить систему валидации `DataAnnotations`, используемую по умолчанию. Она не основана на атрибутах, что упрощает написание собственных проверок для правил валидации и тестирование этих правил;
- чтобы создать валидатор модели, создайте класс, производный от `AbstractValidator<>`, и вызовите метод `RuleFor<>()` в конструкторе для добавления правил валидации. Вы можете связать несколько требований в `RuleFor<>()` так же, как можно добавить в модель несколько атрибутов `DataAnnotations`;
- если вам нужно создать собственное правило валидации, то можно использовать метод `Must()`, чтобы указать предикат. Если вы хотите повторно использовать правило валидации в нескольких моделях, инкапсулируйте правило, как метод расширения, чтобы уменьшить дублирование кода;
- чтобы добавить `FluentValidation` в свое приложение, установите NuGet пакет `FluentValidation.AspNetCore`, вызовите метод `AddFluentValidationAutoValidation()` в файле `Program.cs` и зарегистрируйте свои валидаторы в контейнере внедрения зависимостей. Так вы добавите проверки с помощью `FluentValidation` к встроенной системе `DataAnnotations`;
- чтобы удалить систему валидации на основе `DataAnnotations` и использовать только `FluentValidation`, задайте для параметра `DisableDataAnnotationsValidation` значение `true` в вызове метода `AddFluentValidationAutoValidation()`. По возможности придерживайтесь этого подхода, чтобы избежать выполнения методов валидации из двух разных систем;
- вы можете разрешить `FluentValidation` автоматически обнаруживать и регистрировать все валидаторы в своем приложении, вызвав `AddValidatorsFromAssemblyContaining<T>()`, где `T` – тип сканируемой сборки. Это значит, что вам не нужно отдельно регистрировать каждый валидатор в своем приложении в контейнере внедрения зависимостей.

33

Вызов удаленных API с помощью *IHttpClientFactory*

В этой главе:

- проблемы, вызванные неправильным использованием `HttpClient`, чтобы вызывать HTTP API;
- использование `IHttpClientFactory` для управления жизненными циклами `HttpClient`;
- инкапсуляция конфигурации и обработка временных ошибок с помощью `IHttpClientFactory`.

До сих пор в этой книге мы рассматривали создание веб-страниц и предоставление доступа к API. Будь то клиенты, просматривающие приложение Razor Pages или клиентские одностраничные приложения и приложения для мобильных устройств, использующие ваши API, мы писали API, чтобы их потребляли другие.

Однако очень часто ваше приложение взаимодействует со сторонними сервисами, потребляя *их* API. Например, сайту онлайн-торговли необходимо принимать платежи, отправлять сообщения по электронной почте и SMS-сообщения, а также получать курсы обмена валют от

стороннего сервиса. Наиболее распространенный подход к взаимодействию с сервисами – использование протокола HTTP. До сих пор в этой книге мы рассматривали, как *предоставлять* HTTP-сервисы, используя контроллеры API, но не рассматривали, как их потреблять.

В разделе 33.1 вы узнаете, как лучше всего взаимодействовать с HTTP-службами с помощью *HttpClient*. Если у вас есть опыт работы с C#, то, скорее всего, вы использовали этот класс для отправки HTTP-запросов, но здесь есть две ловушки, которые нужно учитывать. В противном случае ваше приложение может столкнуться с трудностями.

IHttpClientFactory был представлен в .NET Core версии 2.1; он упрощает создание экземпляров *HttpClient* и управление ими и позволяет избежать распространенных ошибок. В разделе 33.2 вы узнаете, как *IHttpClientFactory* это делает, управляя конвейером обработчика *HttpClient*.

Вы познакомитесь с созданием *именованных клиентов* для централизации конфигурации для вызова удаленных API и использованием *типовизированных клиентов* для инкапсуляции поведения удаленного сервиса.

Сбои в сети – жизненный факт, когда вы работаете с API по протоколу HTTP, поэтому важно, чтобы вы правильно с ними справлялись. В разделе 33.3 вы узнаете, как использовать Polly, библиотеку, представляющую возможности обеспечения отказоустойчивости и обработки временных сбоев с открытым исходным кодом для обработки распространенных временных ошибок с помощью простых повторных попыток с возможностью применения более сложных политик.

Наконец, в разделе 33.4 вы увидите, как создать собственный обработчик *HttpMessageHandler*, управляемый *IHttpClientFactory*. Вы можете использовать собственные обработчики для реализации сквозных задач, таких как журналирование, метрики или аутентификация, когда функция должна выполняться каждый раз, когда вы вызываете API по протоколу HTTP. Вы также увидите, как создать обработчик, который автоматически добавляет API-ключ ко всем исходящим запросам к API.

Перефразируя Джона Донна, можно сказать, что «приложение – это не остров» и наиболее распространенный способ взаимодействия с другими приложениями и сервисами – это протокол HTTP. В .NET это означает использование класса *HttpClient*.

33.1 Вызов API по протоколу HTTP: проблема с классом *HttpClient*

В этом разделе вы узнаете, как использовать класс *HttpClient* для вызова API по протоколу HTTP. Я остановлюсь на двух распространенных ошибках, возникающих при использовании *HttpClient*, – это исчерпание сокетов и проблемы ротации DNS – и покажу, почему они возникают. В разделе 33.2 вы увидите, как избежать этих проблем с помощью интерфейса *IHttpClientFactory*.

Приложение очень часто взаимодействует с другими сервисами для выполнения своих обязанностей. Возьмем, к примеру, типичный онлайн-

магазин. Даже в самой базовой версии приложения вам, вероятно, потребуется отправлять электронные письма и принимать платежи с помощью кредитных карт или других сервисов. Можно попробовать создать эту функциональность самостоятельно, но, вероятно, оно того не стоит.

Вместо этого имеет смысл делегировать эти обязанности сторонним сервисам, которые специализируются на данной функциональности. Какой бы сервис вы ни использовали, они почти наверняка предоставляют API по протоколу HTTP для взаимодействия. Для многих сервисов это будет *единственный* способ.

RESTful HTTP, gRPC и GraphQL

Существует множество способов взаимодействия со сторонними сервисами, но REST-совместимые HTTP-сервисы по-прежнему удерживают пальму первенства спустя десятилетия после того, как был впервые предложен протокол HTTP. Каждая платформа и язык программирования, которые только можно себе представить, включают в себя поддержку выполнения HTTP-запросов и обработки ответов. Такая повсеместность делает его наиболее популярным вариантом для большинства сервисов.

Несмотря на свою распространенность, REST-совместимые сервисы не идеальны. Они не отличаются компактностью, а это означает, что в конечном итоге отправляется и принимается больше данных, чем при использовании некоторых других протоколов. Также может быть сложно развернуть REST-совместимые API после их развертывания. Эти ограничения вызвали интерес, в частности, к двум альтернативным протоколам: gRPC и GraphQL.

gRPC призван стать эффективным механизмом обмена данными между серверами. Он построен на основе протокола HTTP/2 и обычно обеспечивает гораздо более высокую производительность по сравнению с традиционными REST-совместимыми API. Поддержка gRPC была добавлена в .NET Core 3.0, и он постоянно улучшается для повышения производительности и добавления новых функций. Полный обзор поддержки .NET можно найти в документации на странице <https://docs.microsoft.com/aspnet/core/grpc>.

В то время как gRPC в первую очередь предназначен для обмена данными между серверами, GraphQL лучше всего использовать для предоставления расширяемых API для мобильных и одностраничных приложений. Он стал очень популярным среди frontend-разработчиков, поскольку может снизить неудобства, связанные с развертыванием и использованием новых API. Для получения дополнительной информации рекомендую прочитать книгу Самера Буна «GraphQL в действии» (Manning, 2021).

Несмотря на преимущества и улучшения, которые могут принести gRPC и GraphQL, REST-совместимые HTTP-сервисы останутся в обозримом будущем, поэтому стоит убедиться, что вы понимаете, как использовать их с HttpClient.

В .NET мы применяем класс HttpClient, чтобы вызывать API по протоколу HTTP. Вы можете применять его для выполнения HTTP-запросов API, предоставления всех заголовков и тела для отправки в запросе и чтения заголовков ответов и данных, которые вы получаете обрат-

но. К сожалению, его сложно использовать правильно, и даже когда вы это делаете, у него есть ограничения. Источник этих трудностей частично связан с тем, что данный класс реализует интерфейс *IDisposable*. Когда вы используете класс, реализующий этот интерфейс, то должны заключать класс в конструкции *using* всякий раз, когда создаете новый экземпляр. Это гарантирует, что неуправляемые ресурсы, используемые типом, будут очищены при удалении класса.

```
using (var myInstance = new MyDisposableClass())
{
    // Используем myInstance;
}
```

СОВЕТ C# также включает упрощенную версию оператора *using*, называемую объявлением *using*, в котором отсутствуют фигурные скобки, как показано в листинге 33.1. Вы можете прочитать больше о синтаксисе по адресу <http://mng.bz/nW12>.

Это может привести вас к мысли, что правильный способ создания *HttpClient* показан в следующем листинге. Это простой пример, когда API-контроллер вызывает внешний API для получения последних курсов обмена валют и возвращает их в качестве ответа.

ВНИМАНИЕ! Не используйте класс *HttpClient*, как показано в листинге 33.1. Использование этого способа может привести к нестабильности вашего приложения, как вскоре будет показано.

Листинг 33.1 Неправильный способ использования класса *HttpClient*

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
app.MapGet("/", async () =>
{
    using HttpClient client = new HttpClient();           ← Заключение HttpClient в оператор using означает, что он удаляется в конце блока видимости
    client.BaseAddress = new Uri("https://example.com/rates/"); ← Настраиваем базовый URL-адрес, используемый для выполнения запросов с помощью HttpClient
    var response = await client.GetAsync("latest");          ← Считываем результат в виде строки и возвращаем его из метода действия
    response.EnsureSuccessStatusCode();
    return await response.Content.ReadAsStringAsync();         ←
});                                                       ←
app.Run();
```

Выполняем GET-запрос к API курсов валют

Возбуждаем исключение, если запрос не был успешным

HttpClient – особенный класс, и *не следует* использовать его таким образом! Проблема в первую очередь связана с тем, как работает реализация базового протокола. Когда вашему компьютеру нужно отправить запрос на HTTP-сервер, вы должны создать *соединение* между своим компьютером и сервером. Чтобы создать соединение, ваш компьютер от-

крывает порт, который имеет случайное число от 0 до 65 535, и подключается к IP-адресу и порту HTTP-сервера, как показано на рис. 33.1. После этого ваш компьютер может отправлять HTTP-запросы на сервер.

ОПРЕДЕЛЕНИЕ

Комбинация IP-адреса и порта называется *сокетом*.

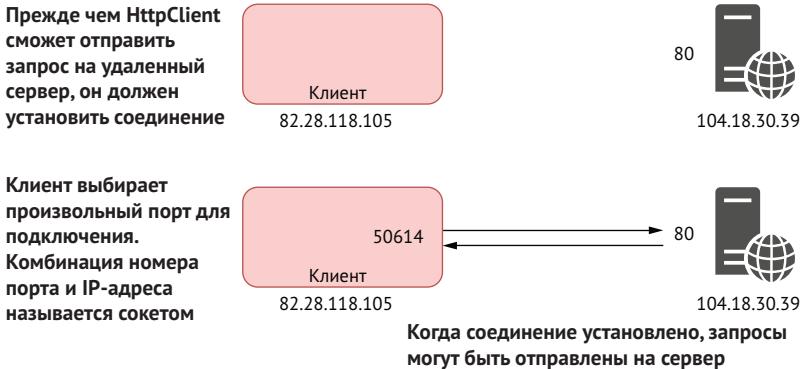


Рис. 33.1 Чтобы создать соединение, клиент выбирает случайный порт и подключается к порту и IP-адресу HTTP-сервера. После этого клиент может отправлять HTTP-запросы на сервер

Основная проблема с инструкцией `using` и `HttpClient` заключается в том, что это может привести к проблеме, которая носит название *исчерпание сокетов*, как показано на рис. 33.2. Это происходит, когда все порты на вашем компьютере заняты другими HTTP-соединениями, поэтому ваш компьютер больше не может отправлять запросы. В этот момент ваше приложение зависает, ожидая освобождения сокета. Очень плохой опыт!

Учитывая вышесказанное, нужно отметить, что существует 65 536 различных номеров портов, и вы можете подумать, что такая ситуация маловероятна. Это правда, вы, скорее всего, столкнетесь с подобной проблемой только на сервере, который устанавливает много подключений, но такое явление не настолько редкое, как вы думаете.

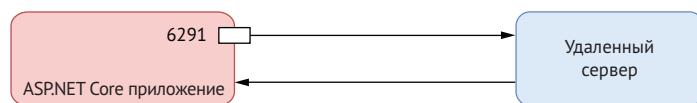
Проблема состоит в том, что когда вы удаляете `HttpClient`, *он не закрывает сокет немедленно*. Особенность протокола TCP/IP, используемого для HTTP-запросов, состоит в том, что соединение после попытки закрыть его переходит в состояние, называемое `TIME_WAIT`. Затем соединение ожидает в течение определенного периода (в Windows этот период составляет 240 с), прежде чем полностью закрыть сокет.

Пока период `TIME_WAIT` не истечет, вы не можете повторно использовать сокет в другом `HttpClient` для выполнения HTTP-запросов. Если вы делаете много запросов, это может быстро привести к исчерпанию сокетов, как показано на рис. 33.2.

СОВЕТ Вы можете просмотреть состояние активных портов или сокетов в Windows и Linux, запустив команду `netstat` из командной строки или окна терминала. Обязательно выполните команду `netstat -n` в Windows, чтобы пропустить разрешение DNS.

1. Приложение создает новый экземпляр HttpClient и инициирует запрос к удаленному серверу

2. Назначается случайный порт (6291), и устанавливается соединение с удаленным сервером

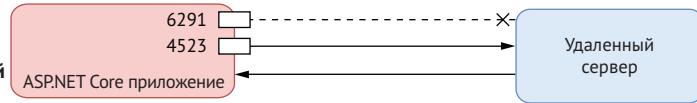


3. После получения ответа приложение освободит HttpClient и начнет закрывать соединение



4. Порт остается в состоянии TIME_WAIT в течение 240 секунд

5. Когда приложение захочет отправить другой запрос, оно создает новый экземпляр HttpClient



6. Порт 6291 все еще используется, поэтому необходимо использовать другой порт, например порт 4523

7. С достаточным количеством запросов на компьютере, на котором запущено ваше приложение, могут закончиться порты, так как все они застрянут в состоянии TIME_WAIT, и вы больше не сможете отправлять или получать новые запросы

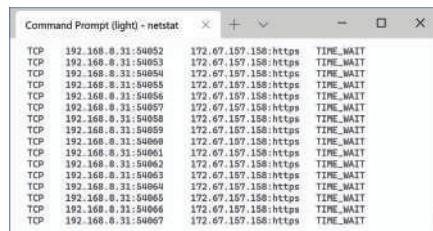


Рис. 33.2 Удаление HttpClient может привести к исчерпанию ресурсов сокета. Каждое новое подключение требует, чтобы операционная система назначила новый сокет, и закрытие сокета делает его недоступным, пока не истечет период TIME_WAIT, равный 240 с. В конце концов, у вас могут закончиться сокеты, и тогда вы не можете отправлять исходящие HTTP-запросы

Вместо того чтобы избавляться от HttpClient, рекомендовалось (до того как *IHttpClientFactory* был введен в .NET Core 2.1) использовать один экземпляр HttpClient, как показано в следующем листинге.

Листинг 33.2 Использование одного экземпляра HttpClient для предотвращения исчерпания сокетов

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
```

```
HttpClient client = new HttpClient
{
    BaseAddress = new Uri("https://example.com/rates/"),
};
app.MapGet("/", async () =>
```

Создается единственный экземпляр HttpClient и сохраняется как статическое поле

```

{
    var response = await client.GetAsync("latest"); ←
    response.EnsureSuccessStatusCode();
    return await response.Content.ReadAsStringAsync();
};

app.Run();

```

Несколько запросов используют один и тот же экземпляр HttpClient

Это решает проблему исчерпания сокетов. Поскольку вы не удаляете HttpClient, сокет не удаляется, поэтому вы можете повторно использовать один и тот же порт для нескольких запросов. Независимо от того, сколько раз вы вызываете метод GetRates() в предыдущем примере, вы будете использовать только один сокет. Задача решена!

К сожалению, это создает другую проблему, связанную с DNS. DNS отвечает за то, как удобные имена хостов, которые мы используем, например manning.com, преобразуются в IP-адреса, которые нужны компьютерам. Когда требуется новое соединение, HttpClient сначала проверяет DNS-запись для хоста, чтобы найти IP-адрес, а затем устанавливает соединение. Для последующих запросов соединение уже установлено, поэтому повторный вызов DNS не осуществляется.

Для единственного экземпляра HttpClient это может быть проблемой, потому что HttpClient не обнаружит изменения DNS. DNS часто используется в облачном окружении для балансировки нагрузки, чтобы выполнять постепенное развертывание¹. Если DNS-запись сервиса, который вы вызываете, изменяется в течение жизненного цикла вашего приложения, единственный экземпляр HttpClient продолжит вызывать старый сервис, как показано на рис. 33.3.

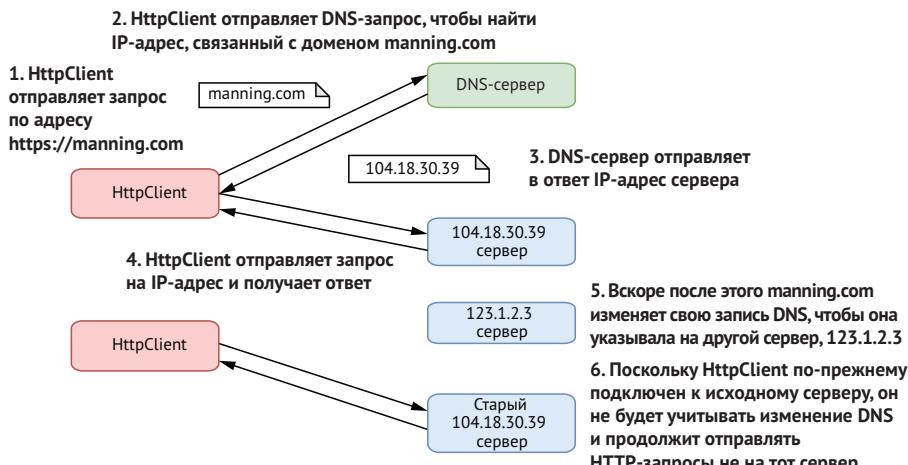


Рис. 33.3 HttpClient выполняет поиск в DNS перед установкой соединения, чтобы определить IP-адрес, связанный с именем хоста. Если DNS-запись имени хоста изменится, единственный экземпляр HttpClient не обнаружит его и продолжит отправку запросов на исходный сервер, к которому он подключен

¹ Например, Azure Traffic Manager использует DNS для маршрутизации запросов. Подробнее узнать о том, как это работает, можно на странице <https://azure.microsoft.com/en-gb/services/traffic-manager/>.

ПРИМЕЧАНИЕ HttpClient не будет учитывать изменение в DNS, пока существует исходное соединение. Если исходное соединение закрыто (например, если исходный сервер отключается от сети), то он будет учитывать изменение в DNS, поскольку ему нужно установить новое соединение.

Похоже, что в обоих случаях у вас будут неприятности! К счастью, *IHttpClientFactory* может позаботиться обо всем этом за вас.

33.2 Создание экземпляров класса *HttpClient* с помощью интерфейса *IHttpClientFactory*

В этом разделе вы узнаете, как использовать интерфейс *IHttpClientFactory*, чтобы избежать распространенных ошибок, связанных с *HttpClient*. Я покажу несколько шаблонов, которые можно использовать для создания экземпляров *HttpClient*:

- использование метода *CreateClient()* в качестве замены *HttpClient*;
- использование *именованных клиентов* для централизации конфигурации *HttpClient*, применяемой для вызова определенного стороннего API;
- использование *типовизированных клиентов* для инкапсуляции взаимодействия со сторонним API, чтобы упростить потребление со стороны своего кода.

Интерфейс *IHttpClientFactory* упрощает *правильное* создание экземпляров класса *HttpClient*, вместо того чтобы полагаться на один из ошибочных подходов, которые мы обсуждали в разделе 33.1. Он также упрощает настройку нескольких экземпляров *HttpClient* и позволяет создавать конвейер промежуточного ПО для исходящих запросов.

Прежде чем мы рассмотрим, как *IHttpClientFactory* делает все это, разберемся, как *HttpClient* работает «за кулисами».

33.2.1 Использование *IHttpClientFactory* для управления жизненным циклом *HttpClientHandler*

В этом разделе мы рассмотрим конвейер обработчиков, используемый *HttpClient*. Вы увидите, как *IHttpClientFactory* управляет жизненным циклом этого конвейера и как это позволяет ему избежать нехватки сокетов и проблем с DNS.

Класс *HttpClient*, который вы обычно используете для выполнения HTTP-запросов, отвечает за оркестрацию запросов, но не за создание самого соединения. Вместо этого *HttpClient* вызывает конвейер *HttpMessageHandler*, в конце которого находится *HttpClientHandler*, который устанавливает фактическое соединение и отправляет HTTP-запрос, как показано на рис. 33.4.

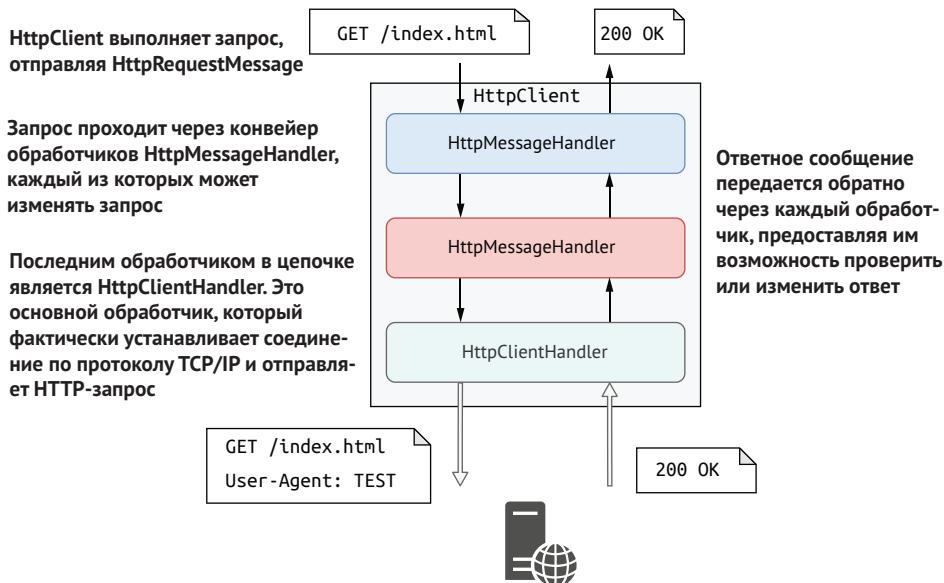


Рис. 33.4 Каждый экземпляр класса `HttpClient` содержит конвейер, состоящий из `HttpMessageHandlers`. Последний обработчик – `HttpClientHandler`, который устанавливает соединение с удаленным сервером и отправляет HTTP-запрос. Эта конфигурация аналогична конвейеру промежуточного ПО ASP.NET Core и позволяет вносить сквозные корректировки в исходящие запросы

Эта конфигурация очень напоминает конвейер промежуточного ПО, используемый приложениями ASP.NET Core, но это исходящий конвейер. Когда `HttpClient` делает запрос, каждый обработчик получает возможность изменить запрос до того, как последний `HttpClientHandler` выполнит настоящий HTTP-запрос. Каждый обработчик, в свою очередь, получает возможность просмотреть ответ после его получения.

СОВЕТ Вы увидите пример использования этого конвейера для сквозной задачи в разделе 33.3, когда мы добавим обработчик временных ошибок.

Проблемы исчерпания сокетов и DNS, описанные в разделе 33.1, связаны с удалением `HttpClientHandler` в конце конвейера. По умолчанию, когда вы удаляете `HttpClient`, вы также удаляете конвейер обработчиков. `IHttpClientFactory` отделяет жизненный цикл `HttpClient` от используемого `HttpClientHandler`.

Разделение жизненного цикла этих двух компонентов позволяет `IHttpClientFactory` решать проблемы исчерпания сокетов и ротации DNS. Это достигается двумя способами:

- путем создания пула доступных обработчиков – исчерпание сокетов происходит при удалении `HttpClientHandler` из-за проблемы с `TIME_WAIT`, описанной ранее. `IHttpClientFactory` решает эту проблему, создавая пул обработчиков. `IHttpClientFactory` поддерживает *активный*

обработчик, который используется для создания всех экземпляров класса *HttpClient* в течение двух минут. Когда *HttpClient* удаляется, базовый обработчик не удаляется, поэтому соединение не закрывается. В результате исчерпание сокетов – не проблема;

- *периодически удаляя (освобождая) обработчики* – совместное использование конвейеров обработчиков решает проблему исчерпания сокетов, но не решает проблему DNS. Чтобы обойти ее, *IHttpClientFactory* периодически (каждые две минуты) создает новый активный *HttpClientHandler*, который используется для каждого экземпляра *HttpClient*, созданного впоследствии. Поскольку эти экземпляры используют новый обработчик, они создают новое соединение TCP/IP, поэтому изменения DNS учитываются.

IHttpClientFactory периодически освобождает обработчики с «истекшим сроком действия» в фоновом режиме, если они больше не используются *HttpClient*. Это гарантирует, что экземпляры *HttpClient* в вашем приложении используют только ограниченное количество подключений.

СОВЕТ Я написал статью в блоге, в котором подробно рассматривается, как *IHttp-ClientFactory* выполняет эту ротацию. Это подробный пост, но он может быть интересен тем, кто хочет знать, как все реализуется за кулисами: <http://mng.bz/8NRK>.

Ротация обработчиков с помощью *IHttpClientFactory* решает обе проблемы, которые мы обсуждали. Еще один бонус заключается в том, что существующие способы использования *HttpClient* можно легко заменить на *IHttpClientFactory*.

IHttpClientFactory по умолчанию включен в ASP.NET Core; вам просто нужно добавить его в сервисы своего приложения в методе *Configure-Services()* файла *Startup.cs*:

```
builder.Services.AddHttpClient();
```

Так вы регистрируете *IHttpClientFactory* в качестве объекта-одиночки в своем приложении, поэтому можете внедрить его в любой другой сервис. Например, в следующем листинге показано, как заменить подход, где используется *HttpClient* из листинга 33.2, версией, использующей *IHttpClientFactory*.

Листинг 33.3 Использование *IHttpClientFactory* для создания экземпляра класса *HttpClient*

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddHttpClient();           ← Внедряем IHttpClientFactory с помощью внедрения зависимостей
WebApplication app = builder.Build();
app.MapGet("/", async (IHttpClientFactory factory) => {           ← Настраиваем
{   HttpClient для
    factory.CreateClient();   вызова API,
}   как и раньше
}
```

```
HttpClient client = factory.CreateClient(); ← Создаем экземпляр HttpClient с HttpClientHandler, управляемым фабрикой

client.BaseAddress =
    new Uri("https://example.com/rates/");

var response = await client.GetAsync("latest");
    Используем
    HttpClient
    точно так же,
    как и раньше

response.EnsureSuccessStatusCode();
return await response.Content.ReadAsStringAsync();

});

app.Run();
```

Непосредственным преимуществом использования `IHttpClientFactory`, таким образом, является эффективное решение проблем, связанных с сокетами и DNS. Чтобы воспользоваться преимуществами этого шаблона, необходимо внести минимальные изменения, поскольку основная часть вашего кода остается неизменной. Это хороший выбор, если вы проводите рефакторинг существующего приложения.

SocketsHttpHandler и `IHttpClientFactory`

Ограничения `HttpClient`, описанные в разделе 33.1, применяются конкретно к `HttpClientHandler` в конце конвейера обработчиков `HttpClient`. `IHttpClientFactory` предоставляет механизм для управления жизненным циклом и повторного использования экземпляров `HttpClientHandler`.

В .NET Core 5 была представлена замена `HttpClientHandler`: `SocketsHttpHandler`. Этот обработчик имеет несколько преимуществ, в первую очередь повышение производительности и согласованность между платформами. `SocketsHttpHandler` также можно настроить для использования пула соединений и повторного использования, как и `IHttpClientFactory`.

Итак, если `HttpClient` уже может использовать пул соединений, стоит ли применять `IHttpClientFactory`? В большинстве случаев я бы сказал: да. Вы должны вручную настроить пул соединений с помощью `SocketsHttpHandler`, а у `IHttpClientFactory` есть дополнительные функции, такие как именованные и типизированные клиенты.

Тем не менее если вы работаете со сценарием, где нет внедрения зависимостей и где нельзя использовать `IHttpClientFactory`, обязательно активируйте пул соединений `SocketsHttpHandler`, как описано в этом посте Стива Гордона: <http://mng.bz/E27q>.

Решение проблемы сокетов – одно из существенных преимуществ использования `IHttpClientFactory` вместо `HttpClient`, но не единственное преимущество. Вы также можете использовать `IHttpClientFactory` для очистки конфигурации клиента, как будет показано в следующем разделе.

33.2.2 Настройка именованных клиентов во время регистрации

В этом разделе вы узнаете, как использовать паттерн *Именованный клиент* с *IHttpClientFactory*. Этот паттерн инкапсулирует логику для вызова стороннего API в одном месте, что упрощает использование *HttpClient* в коде потребления.

ПРИМЕЧАНИЕ *IHttpClientFactory* использует тот же тип *HttpClient*, с которым вы знакомы, если вы применяете .NET Framework. Большая разница заключается в том, что *IHttpClientFactory* решает проблему исчерпания DNS и сокетов за счет управления базовыми обработчиками сообщений.

Использование *IHttpClientFactory* решает технические проблемы, которые я описал в разделе 33.1, но код в листинге 33.3 по-прежнему выглядит довольно запутанным, в первую очередь потому, что вы должны настроить *HttpClient* таким образом, чтобы он указывал на ваш сервис, прежде чем использовать его. Если вам нужно создать экземпляр *HttpClient* для вызова API в нескольких местах приложения, вы также должны *настроить* его в разных местах.

IHttpClientFactory предоставляет удобное решение этой проблемы, позволяя централизованно настраивать *именованные клиенты*. У этих клиентов есть строковое имя и функция конфигурации, которая запускается всякий раз, когда запрашивается экземпляр именованного клиента. Вы можете определить несколько функций конфигурации, которые запускаются последовательно для настройки нового экземпляра *HttpClient*.

Например, в следующем листинге показано, как зарегистрировать именованного клиента "rates". Этот клиент настроен с правильным *BaseAddress* и задает заголовки по умолчанию, которые должны отправляться с каждым исходящим запросом. После того как вы настроили именованного клиента, вы можете создать его из экземпляра *IHttpClientFactory*, используя имя клиента, "rates".

Листинг 33.4 Использование *IHttpClientFactory* для создания именованного клиента

```
Функция конфигурации выполняется каждый раз, когда запрашивается именованный HttpClient
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddHttpClient("rates", (HttpClient client) =>
{
    client.BaseAddress =
        new Uri("https://example.com/rates/");
    client.DefaultRequestHeaders.Add(
        HeaderNames.UserAgent, "ExchangeRateViewer");
})
    .ConfigureHttpClient((HttpClient client) => {})
    .ConfigureHttpClient(
        > (IServiceProvider provider, HttpClient client) => {});

Существуют дополнительные перегруженные варианты, которые разрешают доступ к контейнеру внедрения зависимостей при создании именованного клиента
Указываем имя клиента и функцию конфигурации
Вы можете добавить дополнительные функции конфигурации для именованного клиента, которые будут выполняться последовательно
```

```

WebApplication app = builder.Build();

app.MapGet("/", async (IHttpClientFactory factory) => {
    HttpClient client = factory.CreateClient("rates");
    var response = await client.GetAsync("latest");
    response.EnsureSuccessStatusCode();
    return await response.Content.ReadAsStringAsync();
});

app.Run();

```

Запрашивает именованного клиента с названием «rates»

Внедряет IHttpClientFactory с помощью инверсии зависимостей

Использует HttpClient, как раньше

ПРИМЕЧАНИЕ Вы по-прежнему можете создавать ненастроенных клиентов с помощью метода `CreateClient()` без имени. Имейте в виду, что если вы передаете ненастроенное имя, например `CreateClient("MyRates")`, то возвращаемый клиент не будет сконфигурирован. Будьте внимательны: имена клиентов чувствительны к регистру, поэтому "rates" и "Rates" – это разные клиенты.

Именованные клиенты позволяют централизовать конфигурацию `HttpClient` в одном месте, снимая ответственность за *настройку* клиента с кода потребления. Но на этом этапе вы все еще работаете с низкоуровневыми HTTP-вызовами – например, предоставляя относительный URL-адрес для вызова (`"/latest"`) и разбирая ответ. `IHttpClientFactory` включает в себя функцию, упрощающую поддержание чистоты этого кода.

33.2.3 Использование типизированных клиентов для инкапсуляции HTTP-вызовов

Существует распространенный шаблон, когда вам нужно взаимодействовать с API, – это инкапсуляция механизма такого взаимодействия в отдельный сервис. Это можно легко сделать с помощью уже знакомых вам функций `IHttpClientFactory`, извлекая тело функции `GetRates()` из листинга 33.4 в отдельный сервис. Но в `IHttpClientFactory` также есть более основательная поддержка данного шаблона.

`IHttpClientFactory` поддерживает *типовизированных клиентов*. Типизированный клиент – это класс, который принимает настроенный `HttpClient` в своем конструкторе. Он использует его для взаимодействия с удаленным API и предоставляет понятный интерфейс для вызова потребителей. Вся логика взаимодействия с удаленным API инкапсулирована в типизированном клиенте, например какие пути к URL-адресам нужно вызывать, какие HTTP-методы использовать и какие типы ответов возвращает API. Эта инкапсуляция упрощает вызов стороннего API из нескольких мест в приложении с помощью типизированного клиента.

Например, в следующем листинге показан пример типизированного клиента для API курсов обмена валют, показанного в предыдущих листингах. Он принимает `HttpClient` в своем конструкторе и предоставляет метод `GetLatestRates()`, который инкапсулирует логику взаимодействия со сторонним API.

Листинг 33.5 Создание типизированного клиента для API курсов обмена валют

```
public class ExchangeRatesClient
{
    private readonly HttpClient _client;
    public ExchangeRatesClient(HttpClient client)
    {
        _client = client;
    }
    public async Task<string> GetLatestRates() <|-->
    {
        var response = await _client.GetAsync("latest");
        response.EnsureSuccessStatusCode();

        return await response.Content.ReadAsStringAsync();
    }
}
```

Внедряем HttpClient с помощью внедрения зависимостей вместо IHttpClientFactory

Логика метода GetLatestRates() инкапсулирует логику взаимодействия с API

Используем HttpClient так же, как и раньше

Затем мы можем внедрить этот *ExchangeRatesClient* в потребляющие сервисы, и им не нужно ничего знать о том, как выполнять HTTP-запросы к удаленной службе; им просто нужно взаимодействовать с типизированным клиентом. Мы можем обновить листинг 33.3, чтобы использовать типизированного клиента, как показано в следующем листинге, после чего метод действия *GetRates()* становится пустяком.

Листинг 33.6 Использование типизированного клиента для инкапсуляции вызовов удаленного HTTP-сервера

```
app.MapGet("/", async (ExchangeRatesClient ratesClient) => <-->
    await ratesClient.GetLatestRates());
```

Вызываем API типизированного клиента. Типизированный клиент выполняет правильные HTTP-запросы

Внедряем типизированного клиента в конструктор

На этом этапе вы можете быть немного сбиты с толку: я еще не упомянул, как задействован *IHttpClientFactory*!

ExchangeRatesClient принимает *HttpClient* в своем конструкторе. *IHttpClientFactory* отвечает за создание *HttpClient*, его настройку для вызова удаленной службы и внедрение в новый экземпляр типизированного клиента.

Можно зарегистрировать *ExchangeRatesClient* как типизированного клиента и настроить *HttpClient*, который внедряется в *ConfigureServices*, как показано в следующем листинге. Это очень похоже на настройку именованного клиента, поэтому вы можете зарегистрировать дополнительную конфигурацию для *HttpClient*, которая будет внедрена в типизированного клиента.

Листинг 33.7 Регистрация типизированного клиента с помощью HttpClientFactory в файле Startup.cs

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddHttpClient<ExchangeRatesClient>

➤ Вы можете предоставить дополнительную функцию конфигурации для HttpClient, который будет внедрен
Регистрирует HttpClient client) =>
типовизированного клиента, client.BaseAddress = new Uri("https://example.com/rates/");
используя обобщенный метод client.DefaultRequestHeaders.Add(
AddHttpClient HeaderNames.UserAgent, "ExchangeRateViewer");
})
.ConfigureHttpClient((HttpClient client) => {});
}
◀ Как и для именованных клиентов, вы можете предоставить несколько функций конфигурации

WebApplication app = builder.Build();
app.MapGet("/", async (ExchangeRatesClient ratesClient) =>
    await ratesClient.GetLatestRates());
app.Run();
```

За кулисами вызов AddHttpClient<ExchangeRatesClient> выполняет несколько действий:

- регистрирует HttpClient как сервис с жизненным циклом Transient в контейнере. Это означает, что вы можете принять HttpClient в конструкторе любого сервиса в вашем приложении, и IHttpClientFactory внедрит экземпляр из пула по умолчанию, который не имеет дополнительной конфигурации;
- регистрирует ExchangeRatesClient как сервис с жизненным циклом Transient в контейнере;
- управляет созданием ExchangeRatesClient, поэтому всякий раз, когда требуется новый экземпляр, HttpClient в пуле настраивается, как определено в лямбда-методе AddHttpClient<T>.

СОВЕТ Типизированного клиента можно рассматривать как оболочку для именованного клиента. Я большой поклонник этого подхода, поскольку он объединяет всю логику для взаимодействия с удаленной службой в одном месте, а также избегает «волшебных строк», которые используются с именованными клиентами, устранив возможность опечаток.

Еще один вариант при регистрации типизированных клиентов – зарегистрировать интерфейс, помимо реализации. Часто это хорошая практика, поскольку значительно упрощает тестирование потребляющего кода. Например, если типизированный клиент из листинга 33.5 реализует интерфейс IExchangeRatesClient, вы можете зарегистрировать интерфейс и реализацию типизированного клиента, используя

```
builder.Services.AddHttpClient<IExchangeRatesClient, ExchangeRatesClient>()
```

Затем можно внедрить это в потребляющий код, используя тип интерфейса:

```
app.MapGet("/", async (IExchangeRatesClient ratesClient) =>
    await ratesClient.GetLatestRates());
```

Еще один часто используемый шаблон – не предоставлять никакой конфигурации для типизированного клиента в методе `ConfigureServices()`. Вместо этого вы можете поместить данную логику в конструкторе `ExchangeRatesClient` с использованием внедренного `HttpClient`:

```
public class ExchangeRatesClient
{
    private readonly HttpClient _client;
    public ExchangeRatesClient(HttpClient client)
    {
        _client = client;
        _client.BaseAddress = new Uri("https://example.com/rates/");
    }
}
```

Функционально это эквивалентно подходу, показанному в листинге 33.7. Куда поместить конфигурацию для своего `HttpClient` – это вопрос вкуса. Если вы воспользуетесь этим подходом, то вам не нужно предоставлять лямбда-функцию конфигурации в `ConfigureServices`:

```
builder.Services.AddHttpClient<ExchangeRatesClient>();
```

Именованные и типизированные клиенты удобны для управления и инкапсуляции конфигурации `HttpClient`, но `IHttpClientFactory` дает еще одно преимущество, на которое мы еще не обратили внимания: расширить конвейер обработчиков `HttpClient` становится проще.

33.3 Обработка временных ошибок HTTP с помощью библиотеки Polly

В этом разделе вы узнаете, как справиться с очень распространенной ситуацией: «временными» ошибками при вызове удаленной службы, вызванными ошибкой на удаленном сервере или временными проблемами сети. Вы увидите, как использовать `IHttpClientFactory` для решения подобных сквозных задач, добавляя обработчики в конвейер обработчиков `HttpClient`.

В разделе 33.2.1 я писал, что `HttpClient` состоит из «конвейера» обработчиков. Существенное преимущество этого конвейера, как и конвейера промежуточного ПО вашего приложения, заключается в том, что он позволяет решать сквозные задачи для всех запросов. Например, `IHttpClientFactory` автоматически добавляет обработчик к каждому экземпляру `HttpClient`, который фиксирует код состояния и продолжительность каждого исходящего запроса.

Помимо журналирования, еще одним очень распространенным требованием является обработка временных ошибок при вызове внешнего API. Временные ошибки могут возникать при отключении сети или временном отключении удаленного API. В случае временных ошибок простая

повторная попытка отправить запрос часто может быть успешной, но необходимость вручную писать код для этого довольно обременительна.

У ASP.NET Core есть библиотека Microsoft.Extensions.Http.Polly, которая упрощает обработку временных ошибок. Она использует библиотеку с открытым исходным кодом Polly (<https://github.com/App-vNext/Polly>) для автоматического повтора запросов, которые не выполняются из-за временных ошибок сети.

Polly – уже давно существующая библиотека для обработки временных ошибок, которая включает в себя множество различных стратегий обработки ошибок, таких как простые повторные отправки, экспоненциальная задержка отправки, паттерны *Предохранитель* и *Bulkhead Isolation* и многое другое. Каждая стратегия подробно описана на странице <https://github.com/App-vNext/Polly>, поэтому обязательно прочтите обо всех преимуществах и компромиссах при выборе стратегии.

Чтобы получить представление о том, какие варианты доступны, мы добавим простую стратегию повторной отправки в ExchangeRatesClient, как было показано в разделе 33.2. Если запрос не выполняется из-за проблемы в сети, такой как тайм-аут или ошибка сервера, мы настроим Polly для автоматической повторной отправки запроса как часть конвейера обработчиков, как показано на рис. 33.5.

Чтобы добавить обработку временных ошибок для именованного клиента или HttpClient, выполните следующие действия.

- 1 Установите пакет NuGet Microsoft.Extensions.Http.Polly в свой проект, выполнив команду dotnet add package Microsoft.Extensions.Http.Polly, используя проводник NuGet в Visual Studio или добавив элемент <PackageReference> в файл своего проекта:

```
<PackageReference Include="Microsoft.Extensions.Http.Polly"
    Version="7.0.0" />
```

- 2 Настройте именованного или типизированного клиента, как показано в листингах 33.4 и 33.7.
- 3 Настройте для своего клиента стратегию обработки временных ошибок, как показано в листинге 33.8.

Листинг 33.8 Настройка стратегии обработки временных ошибок для типизированного клиента

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddHttpClient<ExchangeRatesClient>()
```

```
.AddTransientHttpErrorPolicy(policy =>
{
    policy.WaitAndRetryAsync(new[] {
        TimeSpan.FromMilliseconds(200),
        TimeSpan.FromMilliseconds(500),
        TimeSpan.FromSeconds(1)
    });
});
```

Настраивает стратегию, которая выжидает и трижды повторяет запросы в случае возникновения ошибки

Вы можете добавить обработчики временных ошибок к именованным или типизированным клиентам

Используем методы расширения, предоставленные пакетом NuGet, для добавления обработчиков временных ошибок

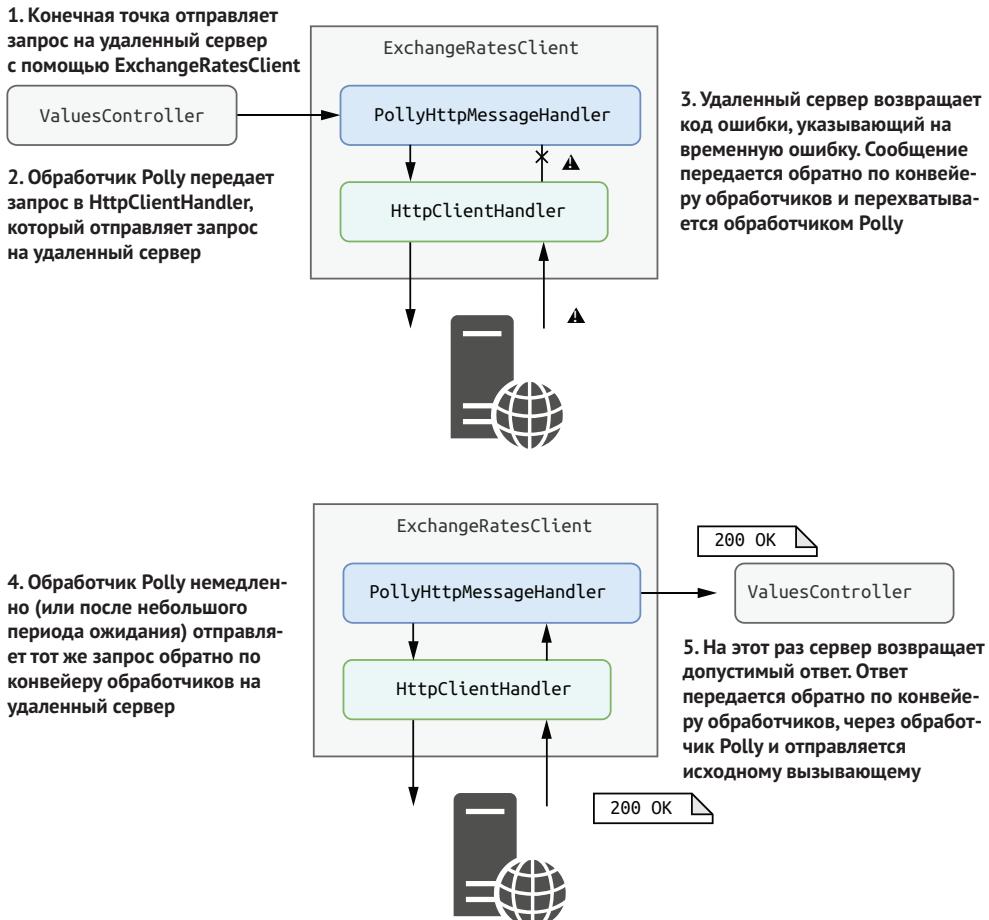


Рис. 33.5 Использование *PolicyHttpMessageHandler* для обработки временных ошибок. Если при вызове удаленного API возникает ошибка, обработчик Polly автоматически повторит запрос. Если запрос завершается успешно, результат возвращается вызывающей стороне. Вызывающей стороне не приходится самостоятельно обрабатывать ошибку, что упростило использование *HttpClient*, сохраняя устойчивость к временным ошибкам

В приведенном выше листинге мы настраиваем обработчик для перехвата временных ошибок, который выполняет три повторные попытки с увеличением промежутка времени между запросами. Если три попытки запроса завершатся неудачно, обработчик проигнорирует ошибку и вернет ее клиенту, как если бы обработчика ошибок не было вообще. По умолчанию обработчик будет повторять любой запрос, который либо:

- возбуждает исключение *HttpRequestException*, указывая на ошибку на уровне протокола, например закрытое соединение;
- возвращает код состояния HTTP 5xx, указывая на ошибку сервера в API;
- возвращает код состояния HTTP 408, указывая на тайм-аут.

СОВЕТ Если вы хотите обрабатывать больше случаев автоматически или ограничивать ответы, которые будут автоматически повторяться, можно настроить логику выбора, как описано в документации «Polly и HttpClientFactory» на сайте GitHub: <http://mng.bz/NY7E>.

Использование стандартных обработчиков, таких как обработчик временных ошибок, позволяет применять одну и ту же логику ко всем запросам, которые выполняются определенным экземпляром `HttpClient`. Конкретная стратегия, которую вы выберете, будет зависеть от характеристик сервиса и запроса, но хорошая стратегия повторных отправок необходима всякий раз, когда вы взаимодействуете с потенциально ненадежными API для протокола HTTP.

ВНИМАНИЕ! При разработке политики обязательно учитывайте ее последствия. В некоторых обстоятельствах может быть лучше быстро потерпеть неудачу, чем повторять запрос, который никогда не будет успешным. Polly включает дополнительные политики, такие как автоматические выключатели, для создания более продвинутых подходов.

Обработчик ошибок Polly – это пример необязательного обработчика `HttpMessageHandler`, который вы можете подключить к своему `HttpClient`, но вы также можете создать собственный обработчик. В следующем разделе вы увидите, как создать обработчик, который добавляет заголовок ко всем исходящим запросам.

33.4 Создание специального обработчика `HttpMessageHandler`

Для большинства сторонних API при их вызове требуется некая форма аутентификации. Например, многие службы требуют, чтобы вы прикрепили API-ключ к исходящему запросу, чтобы запрос можно было привязать к вашей учетной записи. Вместо того чтобы не забывать об этом и вручную добавлять этот заголовок для каждого запроса к API, можно настроить собственный обработчик `HttpMessageHandler` для автоматического прикрепления заголовка.

ПРИМЕЧАНИЕ Более сложные API-интерфейсы могут использовать веб-токены JSON (JWT), полученные от поставщика идентификационной информации. В таком случае рассмотрите возможность использования библиотеки IdentityModel с открытым исходным кодом (<https://identitymodel.readthedocs.io>), обеспечивающей точки интеграции для ASP.NET Core Identity и `HttpClientFactory`.

Вы можете настроить именованного или типизированного клиента с помощью `IHttpClientFactory` для использования обработчика API-

ключа как часть конвейера обработчиков *HttpClient*, как показано на рис. 33.6. Когда вы используете *HttpClient* для отправки сообщения, *HttpRequestMessage* передается через каждый обработчик по очереди. Обработчик API-ключа добавляет дополнительный заголовок и передает запрос следующему обработчику в конвейере. В конце концов, *HttpClientHandler* делает сетевой запрос для отправки HTTP-запроса. После получения ответа каждый обработчик получает возможность проверить (и потенциально изменить) ответ.

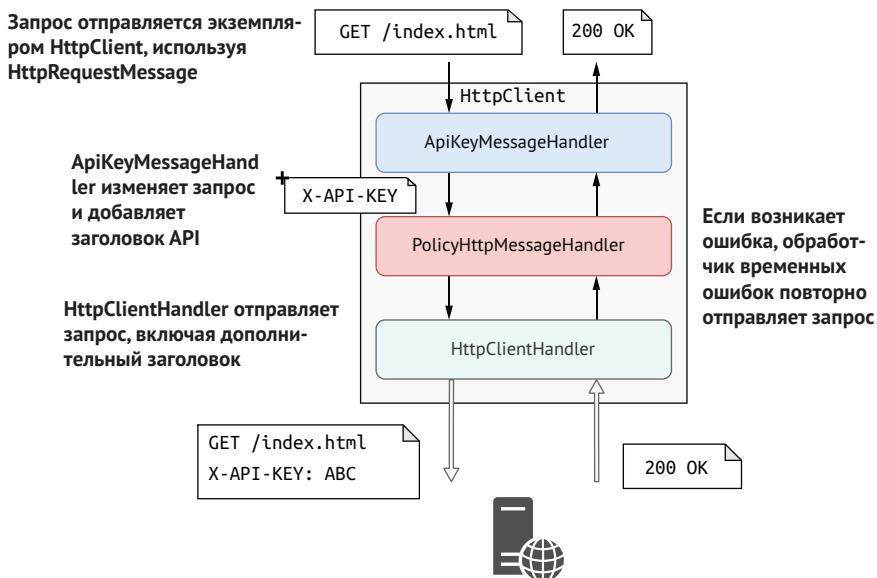


Рис. 33.6 Можно использовать собственный обработчик *HttpMessageHandler* для изменения запросов до того, как они будут отправлены сторонним API. Каждый запрос проходит через специальный обработчик до того, как финальный обработчик (*HttpClientHandler*) отправит запрос в API по протоколу HTTP. После получения ответа каждый обработчик получает возможность проверить и изменить ответ

Чтобы создать собственный *HttpMessageHandler* и добавить его в конвейер типизированного или именованного клиента, можно выполнить следующие действия.

- 1 Создайте специальный обработчик, наследуя от базового класса *DelegatingHandler*.
- 2 Переопределите метод *SendAsync()*, чтобы обеспечить специальное поведение. Вызовите *base.SendAsync()*, чтобы выполнить оставшуюся часть конвейера обработчиков.
- 3 Зарегистрируйте обработчик в контейнере внедрения зависимостей. Если вашему обработчику не требуется состояние, то можно зарегистрировать его как сервис с жизненным циклом *Singleton*; в противном случае следует зарегистрировать его как сервис с жизненным циклом *Transient*.

- 4 Добавьте обработчик к одному или нескольким именованным или типизированным клиентам, вызвав метод `AddHttpMessageHandler<T>()` в `IHttpClientBuilder`, где `T` – ваш тип обработчика. Порядок, в котором вы регистрируете обработчиков, определяет порядок, в котором они будут добавлены в конвейер обработчиков `HttpClient`. При желании можно добавить один и тот же тип обработчика несколько раз в конвейер, а также для нескольких типизированных или именованных клиентов.

В следующем листинге показан пример специального обработчика `HttpMessageHandler`, который добавляет заголовок к каждому исходящему запросу. В этом примере мы будем использовать собственный заголовок "X-API-KEY", но нужный заголовок будет зависеть от стороннего API, который вы вызываете. В этом примере используется строго типизированная конфигурация для внедрения секретного API-ключа, как было показано в главе 10.

Листинг 33.9 Создание специального обработчика `HttpMessageHandler`

Специальные обработчики `HttpMessageHandler` должны наследовать от `DelegatingHandler`

```
public class ApiKeyMessageHandler : DelegatingHandler
{
    private readonly ExchangeRateApiSettings _settings;
    public ApiKeyMessageHandler(
        IOptions<ExchangeRateApiSettings> settings)
    {
        _settings = settings.Value;
    }

    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        request.Headers.Add("API-KEY", _settings.ApiKey);

        HttpResponseMessage response =
            await base.SendAsync(request, cancellationToken);

        return response;
    }
}
```

Внедряем строго типизированные значения конфигурации с помощью внедрения зависимостей

Добавляем дополнительный заголовок ко всем исходящим запросам

Специальные обработчики `HttpMessageHandler` должны наследовать от `DelegatingHandler`

Переопределяем метод `SendAsync`, чтобы реализовать специальное поведение

Вызываем оставшуюся часть конвейера и получаем ответ

Вы можете проверить или изменить ответ, прежде чем возвращать его

Чтобы использовать обработчик, вы должны зарегистрировать его в контейнере внедрения зависимостей и добавить его к именованному или типизированному клиенту. В следующем листинге мы добавляем его в `ExchangeRatesClient` наряду с обработчиком временных ошибок, который мы зарегистрировали в листинге 33.8. Так мы создаем конвейер, аналогичный тому, что показан на рис. 33.6.

Листинг 33.10 Регистрация специального обработчика в *Startup.ConfigureServices*

Настраиваем типизированного клиента для использования настраиваемого обработчика

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddTransient<ApiKeyMessageHandler>();

builder.Services.AddHttpClient<ExchangeRatesClient>()
    .AddHttpMessageHandler<ApiKeyMessageHandler>()
    .AddTransientHttpErrorPolicy(policy => {
        policy.WaitAndRetryAsync(new[] {
            TimeSpan.FromMilliseconds(200),
            TimeSpan.FromMilliseconds(500),
            TimeSpan.FromSeconds(1)
        });
    });

Регистрируем специальный обработчик в контейнере внедрения зависимостей
Добавляем обработчик временных ошибок. Порядок их регистрации определяет их порядок в конвейере
```

Каждый раз, когда вы делаете запрос, используя типизированного клиента *ExchangeRatesClient*, вы можете быть уверены, что API-ключ будет добавлен и временные ошибки будут обрабатываться автоматически.

На этом мы подошли к концу главы, посвященной *IHttpClientFactory*. Учитывая трудности, связанные с правильным использованием класса *HttpClient*, которые были показаны в разделе 33.1, по возможности всегда следует отдавать предпочтение *IHttpClientFactory*. В качестве бонуса *IHttpClientFactory* позволяет легко централизовать конфигурацию API с использованием именованных клиентов и инкапсулировать взаимодействие с API, используя типизированных клиентов.

Резюме

- Используйте класс *HttpClient* для вызова API по протоколу HTTP. Вы можете применять его для выполнения вызовов API по протоколу HTTP, предоставляя все заголовки и тело для отправки в запросе и читая заголовки ответов и данные, которые вы получаете обратно;
- *HttpClient* использует конвейер обработчиков, состоящий из нескольких обработчиков *HttpMessageHandler*, подключенных аналогично конвейеру промежуточного ПО, используемому в ASP.NET Core. Последний обработчик – *HttpClientHandler*, который отвечает за установку подключения к сети и отправку запроса;
- *HttpClient* реализует интерфейс *IDisposable*, но обычно не следует уничтожать экземпляра этого класса. Когда *HttpClientHandler*, который устанавливает соединение TCP/IP, удаляется, он сохраняет соединение открытым в течение периода *TIME_WAIT*. Удаление большого числа экземпляров *HttpClient* за короткий период времени может привести к исчерпанию сокетов, что не позволит машине обрабатывать никакие запросы;
- до появления .NET Core версии 2.1 рекомендовалось использовать один экземпляр *HttpClient* на протяжении всего жизненного цикла приложения. К сожалению, при таком варианте не будут учитываться изменения в DNS, которые обычно используются для управления трафиком в облачном окружении;

- `IHttpClientFactory` решает обе эти проблемы, управляя жизненным циклом конвейера `HttpMessageHandler`. Вы можете создать новый экземпляр `HttpClient`, вызвав метод `CreateClient()`, а `IHttpClientFactory` позаботится об удалении конвейера обработчиков, если он больше не используется;
- вы можете централизовать конфигурацию `HttpClient` в методе `ConfigureServices()` с использованием *именованных* клиентов, вызвав `AddHttpClient("test", c => {})`. Затем можно получить настроенный экземпляр клиента в своих сервисах, вызвав `IHttpClientFactory.CreateClient("test")`;
- вы можете создать *типовизированный* клиента, внедрив `HttpClient` в сервис `T` и настроив клиента с помощью `AddHttpClient<T>(c => {})`. Типизированные клиенты отлично подходят для абстрагирования механизмов HTTP от потребителей клиента;
- вы можете использовать библиотеку `Microsoft.Extensions.Http.Polly`, чтобы добавить обработку временных ошибок HTTP в свои экземпляры `HttpClient`. Вызовите метод `AddTransientHttpErrorPolicy()` при настройке `IHttpClientFactory` в `ConfigureServices` и предоставьте стратегию `Polly`, чтобы контролировать, когда ошибки должны автоматически обрабатываться, а запросы – повторяться;
- обычно используется простая стратегия повторной отправки, чтобы попытаться сделать запрос несколько раз, прежде чем сдаться и вернуть ошибку. При проектировании стратегии обязательно учитывайте ее влияние; в некоторых случаях может быть лучше быстро потерпеть неудачу, нежели повторять запрос, который никогда не будет успешным. `Polly` включает дополнительные стратегии, такие как предохранители, для создания более продвинутых подходов;
- по умолчанию промежуточное ПО для обработки временных ошибок обрабатывает ошибки подключения, ошибки сервера, которые возвращают код 5xx, и ошибки 408 (тайм-аут). Вы можете выполнить настройку, если хотите обрабатывать дополнительные типы ошибок, но убедитесь, что повторяете только безопасные запросы;
- можно создать собственный обработчик `HttpMessageHandler`, чтобы изменять каждый запрос, сделанный через именованного или типизированного клиента. Собственные обработчики хорошо подходят для реализации сквозных задач, таких как журналирование, метрики и аутентификация;
- чтобы создать собственный обработчик `HttpMessageHandler`, наследуйте от `DelegatingHandler` и переопределите метод `SendAsync()`. Вызовите метод `base.SendAsync()`, чтобы отправить запрос следующему обработчику в конвейере, и, наконец, `HttpClientHandler`, который выполняет HTTP-запрос;
- зарегистрируйте свой обработчик в контейнере внедрения зависимостей с жизненным циклом `Transient` или `Singleton`. Добавьте его к именованному или типизированному клиенту с помощью `AddHttpMessageHandler<T>()`. Порядок, в котором вы регистрируете обработчик в `IHttpClientBuilder`, – это порядок, в котором обработчик будет выполняться в конвейере обработчиков `HttpClient`.

34

Создание фоновых задач и сервисов

В этой главе:

- создание задач, которые выполняются в фоновом режиме для вашего приложения;
- использование обобщенного интерфейса `IHost` для создания служб Windows и демонов Linux;
- использование Quartz.NET для выполнения задач по расписанию в кластерном окружении.

На данный момент мы рассмотрели в этой книге очень много вопросов. Вы узнали, как создавать приложения с помощью Razor Pages и API для мобильных клиентов и служб, как добавить аутентификацию и авторизацию в свое приложение, как использовать EF Core для хранения состояния в базе данных и создавать специальные компоненты в соответствии со своими требованиями.

Помимо этих приложений, ориентированных на пользовательский интерфейс, вам может потребоваться создать фоновые или пакетные сервисы. Эти сервисы не предназначены для прямого взаимодействия с пользователями. Они продолжают работать в фоновом режиме, обрабатывая элементы из очереди или периодически выполняя процесс с длительным временем запуска.

Например, вам может потребоваться фоновый сервис, который отправляет подтверждения по электронной почте для заказов из онлайн-магазина, или пакетное задание, которое рассчитывает продажи и убытки для розничных магазинов после окончания рабочего дня. ASP.NET Core включает поддержку этих фоновых задач, предоставляя абстракции для выполнения задачи в фоновом режиме при запуске приложения.

В разделе 34.1 вы узнаете о поддержке фоновых задач, предоставляемой в ASP.NET Core интерфейсом *IHostedService*. Вы узнаете, как использовать вспомогательный класс *BackgroundService* для создания задач, запускаемых по таймеру, и как правильно управлять жизненными циклами внедрения зависимостей в долгосрочной задаче.

В разделе 34.2 мы пойдем дальше и создадим «автономные» сервисы рабочей роли (*worker service*) с использованием обобщенного интерфейса *IHost*. Сервисы рабочей роли не используют Razor Pages или контроллеры API; они состоят только из экземпляров *IHostedService*, выполняющих задачи в фоновом режиме. Вы также узнаете, как настроить и установить сервис рабочей роли в качестве службы Windows или демона Linux.

В разделе 34.3 я представлю библиотеку с открытым исходным кодом Quartz.NET, которая предоставляет широкие возможности планирования для создания фоновых сервисов. Вы узнаете, как установить Quartz.NET в свои приложения, как создавать сложные расписания для своих задач и как добавить избыточность в сервисы рабочей роли с помощью кластеризации.

Прежде чем перейти к более сложным сценариям, мы рассмотрим встроенную поддержку для запуска фоновых задач в ваших приложениях.

34.1 Запуск фоновых задач с помощью *IHostedService*

В большинстве приложений есть задачи, которые выполняются в фоновом режиме, а не в ответ на запрос. Это может быть задача по обработке очереди электронных писем, обработка событий, опубликованных в какой-тошине сообщений, или запуск пакетного процесса для расчета ежедневной прибыли. Перенося эту работу в фоновую задачу, ваш пользовательский интерфейс может оставаться отзывчивым. Например, вместо того чтобы пытаться сразу же отправить электронное письмо, вы можете добавить запрос в очередь и немедленно вернуть ответ пользователю. Фоновая задача может обслуживать эту очередь в фоновом режиме.

В ASP.NET Core можно использовать интерфейс *IHostedService* для выполнения задач в фоновом режиме. Классы, реализующие этот интерфейс, запускаются при запуске вашего приложения, вскоре после того, как приложение начинает обрабатывать запросы, и останавливаются незадолго до остановки приложения, что дает вам точки подключения, необходимые для выполнения большинства задач.

ПРИМЕЧАНИЕ Даже сервер ASP.NET Core, Kestrel, работает как *IHostedService*. В некотором смысле почти все в приложении ASP.NET Core является «фоновой» задачей.

В этом разделе вы увидите, как использовать `IHostedService` для создания фоновой задачи, которая непрерывно выполняется на протяжении всего жизненного цикла вашего приложения. Его можно использовать для разных целей, но в следующем разделе вы увидите, как с его помощью заполнить простой кеш. Вы также узнаете, как использовать сервисы с жизненным циклом `Scoped` в фоновых задачах `Singleton`, самостоятельно управляя областями контейнера.

34.1.1 Запуск фоновых задач по таймеру

В этом разделе вы узнаете, как создать фоновую задачу, которая периодически запускается по таймеру на протяжении всего жизненного цикла вашего приложения. Запуск фоновых задач может быть полезен по многим причинам, например для планирования работы, которая будет выполняться позже, или выполнения работы заранее.

Например, в главе 33 мы использовали `IHttpClientFactory` и типизированного клиента для вызова стороннего сервиса, чтобы получить текущий обменный курс между различными валютами и вернуть их в контроллер API, как показано в следующем листинге.

Листинг 34.1 Использование типизированного клиента для возврата обменных курсов валют из стороннего сервиса

```
app.MapGet("/", async (ExchangeRatesClient ratesClient) =>
    await ratesClient.GetLatestRatesAsync());
```

Типизированный клиент используется для получения обменных курсов из удаленного API и их возврата

Типизированный клиент, созданный с помощью `IHttpClientFactory`, внедряется в конструктор

Простая оптимизация этого кода может заключаться в кешировании значений обменного курса за какой-то период. Есть несколько способов реализовать это, но в данном разделе мы будем использовать простой кеш, который предварительно извлекает обменные курсы в фоновом режиме, как показано на рис. 34.1. Контроллер API просто читает из кеша; ему не нужно делать HTTP-вызывы, поэтому он остается быстрым.

ПРИМЕЧАНИЕ В качестве альтернативного подхода можно добавить кеширование внутри строго типизированного клиента `ExchangeRateClient`. Обратной стороной такого подхода является тот факт, что когда вам нужно обновить курсы, придется выполнить запрос немедленно, а это замедлит общий ответ. Использование фоновой службы обеспечивает стабильную работу вашего контроллера API.

Вы можете реализовать фоновую задачу с помощью интерфейса `IHostedService`. Он состоит из двух методов:

```
public interface IHostedService
{
    Task StartAsync(CancellationToken cancellationToken);
    Task StopAsync(CancellationToken cancellationToken);
}
```

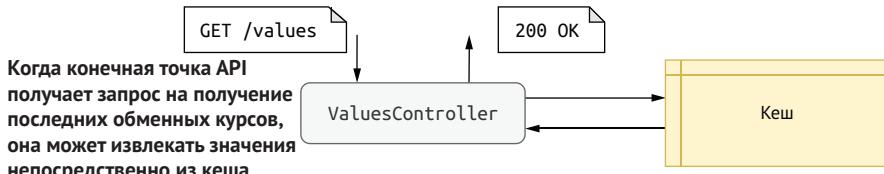
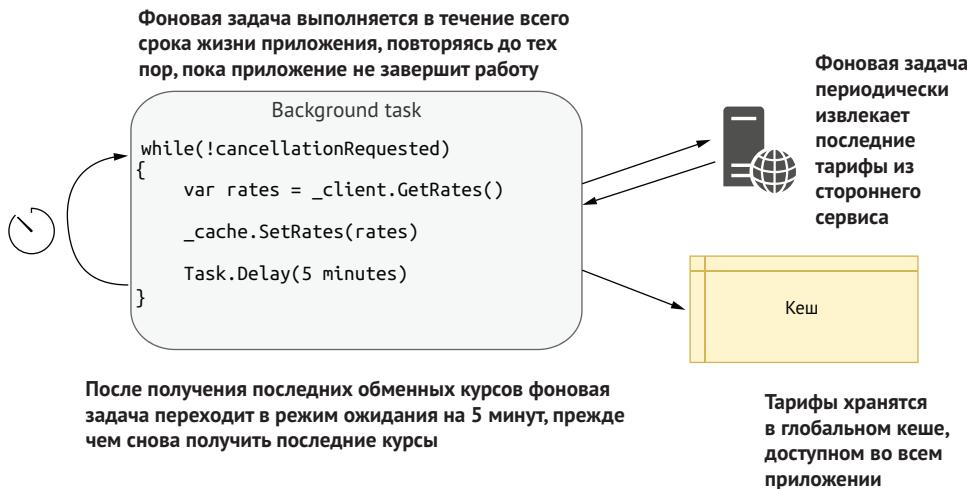


Рис. 34.1 Вы можете использовать фоновую задачу для кеширования результатов стороннего API по расписанию. Затем контроллер API может читать непосредственно из кеша, вместо того чтобы вызывать сам сторонний API. Это уменьшает задержку запросов к вашему контроллеру API, обеспечивая при этом актуальность данных

Существуют тонкости для правильной реализации интерфейса. В частности, метод `StartAsync()`, хотя и является асинхронным, выполняется последовательно как часть запуска вашего приложения. Фоновые задачи, которые, как ожидается, будут выполняться в течение всего жизненного цикла вашего приложения, должны незамедлительно вернуть `Task` и запланировать фоновую работу в другом потоке.

ВНИМАНИЕ! Вызов `await` в методе `IHostedService.StartAsync()` блокирует запуск вашего приложения до завершения метода. В некоторых случаях это может быть полезно, но часто нежелательно для фоновых задач. Чтобы упростить создание фоновых сервисов с использованием передовых паттернов, ASP.NET Core предоставляет абстрактный базовый класс `BackgroundService`, который реализует интерфейс `IHostedService` и используется для длительных задач.

Чтобы создать фоновую задачу, нужно переопределить единственный метод этого класса, `ExecuteAsync()`. Вы можете использовать `async-await` внутри этого метода и продолжать выполнение метода в течение всего жизненного цикла приложения. Например, в следующем листин-

также показан фоновый сервис, который извлекает последние процентные ставки с помощью типизированного клиента и сохраняет их в кеше, как было показано на рис. 34.1. Метод `ExecuteAsync()` продолжает цикл и обновляет кеш до тех пор, пока `CancellationToken`, переданный в качестве аргумента, не укажет, что приложение завершает работу.

Листинг 34.2 Реализация класса `BackgroundService`, который вызывает удаленный API по протоколу HTTP

```

    Наследуем от BackgroundService, чтобы создать
    задачу, которая будет выполняться в течение
    всего жизненного цикла вашего приложения

public class ExchangeRatesHostedService : BackgroundService
{
    private readonly IServiceProvider _provider; ←
    private readonly ExchangeRatesCache _cache; ←

    public ExchangeRatesHostedService(
        IServiceProvider provider, ExchangeRatesCache cache)
    {
        _provider = provider;
        _cache = cache;
    } ←
        Вы должны переопределить
        ExecuteAsync, чтобы настро-
        ить поведение сервиса

    protected override async Task ExecuteAsync( ←
        CancellationToken stoppingToken) ←

    { ←
        Продолжаем
        цикл, пока при-
        ложение не за-
        вершится

        Получаем
        последние
        курсы из уда-
        ленного API
    } ←
        while (!stoppingToken.IsCancellationRequested)
    {
        var client = _provider
            .GetRequiredService<ExchangeRatesClient>(); ←

        string rates = await client.GetLatestRatesAsync();
        _cache.SetRates(rates); ←
            Сохраняет курсы
            в кеш
        await Task.Delay(TimeSpan.FromMinutes(5), stoppingToken); ←
            Ждет 5 минут (или завершения приложения)
            перед обновлением кеша
    }
}

```

Простой кеш для курсов валют

Внедряем `IServiceProvider`, чтобы вы могли создавать экземпляры типизированного клиента

CancellationToken, переданный в качестве аргумента, срабатывает при завершении работы приложения.

Создаем новый экземпляр типизированного клиента, чтобы `HttpClient` был недолговечным

`ExchangeRateCache` из листинга 34.2 – это простой объект-одиночка, в котором хранятся последние курсы обмена валют. Он должен быть потокобезопасным, так как ваши контроллеры API будут обращаться к нему одновременно. Простую реализацию можно увидеть в исходном коде этой главы.

Чтобы зарегистрировать фоновый сервис в контейнере внедрения зависимостей, используйте метод расширения `AddHostedService()` в методе `ConfigureServices()` файла `Startup.cs`, как показано в следующем листинге.

Листинг 34.3 Регистрация *IHostedService* в контейнере внедрения зависимостей

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddHttpClient<ExchangeRatesClient>();
builder.Services.AddSingleton<ExchangeRatesCache>();
builder.Services.AddHostedService<ExchangeRatesHostedService>();
```

Регистрируем типизированного клиента, как и раньше

Регистрируем сервис *ExchangeRatesHostedService* как *IHostedService*

Добавляем объект кеша в качестве объекта-одиночки, потому что вы должны использовать один и тот же экземпляр во всем приложении

Используя фоновый сервис для получения курсов валют, ваш контроллер API становится очень простым. Вместо того чтобы извлекать самые последние курсы, он возвращает значение из кеша, которое обновляется фоновым сервисом:

```
app.MapGet("/", (ExchangeRatesCache cache) =>
    cache.GetLatestRatesAsync());
```

Этот подход к кешированию упрощает API, но вы, возможно, заметили потенциальный риск: если API получит запрос до того, как фоновая служба успешно обновит тарифы, API не сможет вернуть какие-либо данные.

Это может быть и нормально, но вы можете пойти другим путем. Помимо периодического обновления ставок, вы можете использовать метод *StartAsync*, чтобы заблокировать запуск приложения до тех пор, пока ставки не будут успешно обновлены. Таким образом, вы гарантируете, что тарифы будут доступны до того, как приложение начнет обрабатывать запросы, поэтому API всегда будет успешно возвращать данные. В листинге 34.4 показано, как можно обновить листинг 34.2, чтобы заблокировать запуск до тех пор, пока тарифы не обновятся. При этом они продолжают периодически обновляться в фоновом режиме.

Листинг 34.4 Реализация *StartAsync* для блокировки запуска в *IHostedService*

```
public class ExchangeRatesHostedService : BackgroundService
{
    private readonly IServiceProvider _provider;
    private readonly ExchangeRatesCache _cache;
    public ExchangeRatesHostedService(
        IServiceProvider provider, ExchangeRatesCache cache)
    {
        _provider = provider;
        _cache = cache;
    }
    public override async Task StartAsync(
        CancellationToken cancellationToken)
```

Метод *StartAsync* запускается при запуске, прежде чем приложение начнет обрабатывать запросы

```

{
    var success = false;
    while(!success && !cancellationToken.IsCancellationRequested)
    {
        success = await TryUpdateRatesAsync();
    }

    await base.StartAsync(cancellationToken); ←
}

protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
        await Task.Delay(TimeSpan.FromMinutes(5), stoppingToken);
        await TryUpdateRatesAsync();
    }
}

private async Task<bool> TryUpdateRatesAsync()
{
    try
    {
        var client = _provider
            .GetRequiredService<ExchangeRatesClient>();
        string rates = await client.GetLatestRatesAsync();
        _cache.SetRates(rates);
        return true;
    }
    catch(Exception ex)
    {
        return false;
    }
}
}

```

Продолжает попытки обновить ставки, пока это не удастся

После успешного обновления запускается фоновый процесс

ВНИМАНИЕ! Недостатком листинга 34.4 является то, что если возникнет проблема с получением тарифов, приложение никогда не запустится и не начнет прослушивать запросы. Будете вы считать это ошибкой или функцией, зависит от вашего процесса развертывания! Многие оркестраторы, например, будут использовать чередующиеся обновления, которые гарантируют, что новое развертывание прослушивает запросы перед завершением работы старых экземпляров развертывания.

В листингах 34.2 и 34.4 есть один своеобразный аспект. Он заключается в том, что я использовал паттерн *Локатор сервисов* для получения типизированного клиента. Это не идеальный вариант, но вам не следует напрямую внедрять типизированных клиентов в фоновые сервисы. Эти клиенты предназначены для кратковременного существования, чтобы вы могли воспользоваться преимуществами ротации обработчиков HttpClient, как описано в главе 21. Напротив, фоновые сервисы

сы – это синглтоны, которые существуют в течение всего жизненного цикла вашего приложения.

СОВЕТ При желании можно избежать использования паттерна *Локатор сервисов* из листинга 34.2, используя паттерн *Фабрика*, описанный в посте Стива Гордона: <http://mng.bz/opDZ>.

Потребность в краткосрочных сервисах порождает еще один распространенный вопрос: как использовать сервисы с жизненным циклом `Scoped` в фоновой задаче?

34.1.2 Использование сервисов с жизненным циклом `Scoped` в фоновых задачах

Фоновые сервисы, реализующие интерфейс `IHostedService`, создаются один раз при запуске вашего приложения. Это означает, что они являются синглтонами, поскольку экземпляр класса всегда будет только один.

Это приводит к проблеме, если вам нужно использовать сервисы, зарегистрированные с жизненным циклом `Scoped`. Любые сервисы, которые вы внедряете в конструктор своего синглтона `IHostedService`, сами должны быть зарегистрированы как синглтоны. Означает ли это, что фоновому сервису нельзя использовать зависимости с жизненным циклом `Scoped`?

ПРИМЕЧАНИЕ Как я уже говорил в главе 9, зависимости сервиса всегда должны иметь такой же или более продолжительный жизненный цикл, чем сам сервис, чтобы избежать захваченных зависимостей.

Например, давайте представим небольшую вариацию примера с кэшированием из раздела 34.1.1. Вместо того чтобы хранить курсы валют в объекте-одиночке, вы хотите хранить курсы в базе данных, чтобы иметь возможность искать старые курсы.

Большинство поставщиков баз данных, включая `DbContext` от EF Core, регистрируют свои сервисы с жизненным циклом `Scoped`. Это означает, что вам нужно получить доступ к `DbContext` с жизненным циклом `Scoped` изнутри синглтон-сервиса `ExchangeRatesHostedService`, что исключает внедрение `DbContext` с помощью внедрения в конструктор. Решение состоит в том, чтобы создавать новую область контейнера каждый раз, когда вы обновляете обменные курсы.

В типичных приложениях ASP.NET Core фреймворк создает новую область контейнера каждый раз при получении нового запроса, непосредственно перед выполнением конвейера промежуточного ПО. Все сервисы, которые используются в этом запросе, извлекаются из `scoped`-контейнера. Однако в фоновом сервисе запросы *отсутствуют*, поэтому области контейнера не создаются. Решение – создать собственную область.

Вы можете создать новую область контейнера везде, где есть доступ к `IServiceProvider`, вызвав метод `IServiceProvider.CreateScope()`. Так вы создаете контейнер, который можно использовать для получения сервисов с жизненным циклом `Scoped`.

ВНИМАНИЕ! Всегда избавляйтесь от `IServiceScope`, который возвращается методом `CreateScope()`, когда закончите с ним, обычно с помощью инструкции `using`. Так вы удалите все сервисы, которые были созданы контейнером с жизненным циклом `Scoped`, и предотвратите утечку памяти.

В следующем листинге показана версия сервиса `ExchangeRatesHostedService`, которая хранит последние курсы обмена валюты в базе данных с помощью EF Core. Здесь мы создаем новую область для каждой итерации цикла `while` и получаем `AppDbContext` с жизненным циклом `Scoped` из контейнера.

Листинг 34.5 Использование сервисов с жизненным циклом `Scoped` из `IHostedService`

```

→ public class ExchangeRatesHostedService : BackgroundService
{
    private readonly IServiceProvider _provider;
    public ExchangeRatesHostedService(IServiceProvider provider)
    {
        _provider = provider;
    }
}

Background-Service регистрируется в качестве объекта-одиночки
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
        Создаем новую область с помощью корневого IServiceProvider
        Извлекаем из контейнера сервисы с жизненным циклом Scoped
        Получаем последние котировки и сохраняем их с помощью EF Core
        using(IServiceScope scope = _provider.CreateScope())
        {
            var scopedProvider = scope.ServiceProvider;
            var client = scope.ServiceProvider
                .GetRequiredService<ExchangeRatesClient>();

            var context = scope.ServiceProvider
                .GetRequiredService<AppDbContext>();

            var rates = await client.GetLatestRatesAsync();

            context.Add(rates);
            await context.SaveChangesAsync();
        }
        Удаляем область действия с помощью оператора using.
        await Task.Delay(TimeSpan.FromMinutes(5), stoppingToken);
    }
}

```

Боковые комментарии:

- Внедренный `IServiceProvider` можно использовать для получения сервисов с жизненным циклом `Singleton` или для создания областей действия
- Область действия предоставляет `IServiceProvider`, который можно использовать для получения компонентов с заданной областью действия
- Удаляем область действия с помощью оператора `using`.
- Ждем следующей итерации. На следующей итерации создается новая область действия

Создание таких областей – общее решение, когда вы обнаруживаете, что вам нужен доступ к сервисам с жизненным циклом `Scoped`, и вы не работаете в контексте запроса. Яркий пример – реализация интерфейса `IConfigureOptions`, как было показано в главе 31. Вы можете использовать тот же подход – создав новую область, – как показано в посте моего блога под названием «Доступ к сервисам внутри `ConfigureServices` с помощью `IConfigureOptions` в ASP.NET Core»: <http://mng.bz/nMD5>.

СОВЕТ Использование локации сервисов таким образом всегда кажется немного запутанным. Обычно я пытаюсь извлечь тело задачи в отдельный класс и использовать локацию сервисов только для получения этого класса. Пример данного подхода можно увидеть в разделе «Использование службы с жизненным циклом `Scoped` в фоновой задаче» документа Microsoft «Фоновые задачи с размещенными службами в ASP.NET Core»: <http://mng.bz/4ZER>.

`IHostedService` доступен в ASP.NET Core, поэтому вы можете запускать фоновые задачи в Razor Pages или в приложениях с API-контроллерами. Однако иногда все, что вам необходимо, – это фоновая задача, и вам не нужен пользовательский интерфейс. В таких случаях можно использовать низкоуровневую абстракцию `IHost`, абсолютно не беспокоясь об обработке HTTP.

34.2 Создание сервисов рабочей роли без пользовательского интерфейса с использованием `IHost`

В этом разделе вы узнаете о сервисах рабочей роли, которые представляют собой приложения ASP.NET Core, не обрабатывающие HTTP-трафик. Вы узнаете, как создать новый сервис рабочей роли из шаблона, и сравните генерированный код с традиционным приложением ASP.NET Core, а также узнаете, как установить сервис рабочей роли в качестве службы Windows или как демона `systemd` в Linux.

В разделе 34.1 мы кешировали курсы обмена валют, исходя из предположения, что они потребляются напрямую пользовательским интерфейсом вашего приложения: Razor Pages или контроллерами API, например. Однако в примере из раздела 34.1.2 мы сохранили курсы в базе данных, вместо того чтобы хранить их в памяти. Это увеличивает возможность того, что *другие* приложения будут иметь доступ к базе данных, также используя эти курсы. Пойдем еще дальше: можно создать приложение, которое отвечает *только* за кеширование этих курсов и вообще не имеет пользовательского интерфейса?

Начиная с .NET Core версии 3.0 ASP.NET Core основывается на «универсальной» (в отличие от «сетевой») реализации интерфейса `IHost`. Это реализация, которая предоставляет такие функции, как конфигурация, журналирование и внедрение зависимостей. ASP.NET Core добавляет конвейер промежуточного ПО для обработки HTTP-запросов, а также такие парадигмы, как Razor Pages или MVC, как показано на рис. 34.2.

Универсальная абстракция `IHost` предоставляет абстракции ведения журнала, настройки и внедрения зависимостей

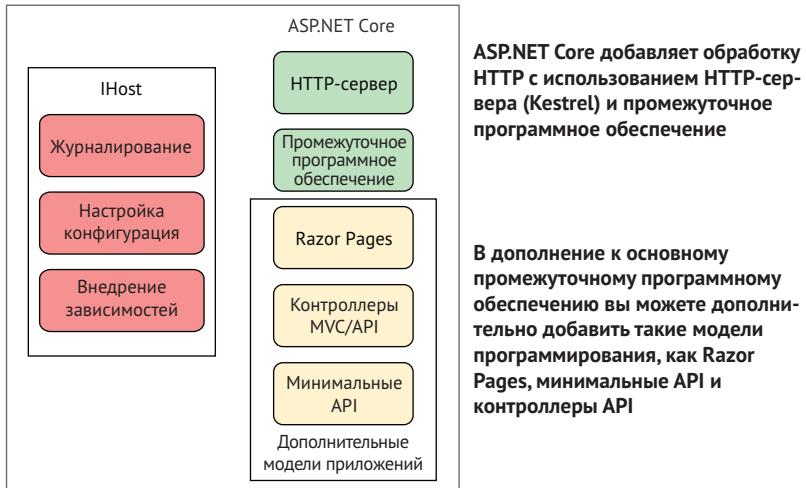


Рис. 34.2 ASP.NET Core основан на реализации обобщенного интерфейса `IHost`. `IHost` предоставляет такие функции, как конфигурация, внедрение зависимостей и журналирование. В ASP.NET Core к этому добавляются обработка HTTP-запросов с помощью конвейера промежуточного ПО, Razor Pages и контроллеры API. Если вам не нужна обработка HTTP-запросов, вы можете использовать `IHost` без дополнительных библиотек ASP.NET Core для создания небольшого приложения

Если вашему приложению не требуется обрабатывать HTTP-запросы, нет реальной причины использовать ASP.NET Core. Вы можете применять только реализацию `IHost` для создания приложения, которое будет занимать меньший объем памяти, более быстрый запуск и меньшее количество внешних интерфейсов, о которых следует беспокоиться с точки зрения безопасности, вместо полноценного приложения ASP.NET Core. Приложения .NET Core, использующие данный подход, обычно называются *сервисами рабочей роли*.

ОПРЕДЕЛЕНИЕ Сервис рабочей роли – это приложение .NET Core, которое использует обобщенный интерфейс `IHost`, но не включает в себя библиотеки ASP.NET Core для обработки HTTP-запросов. Иногда их называют «безголовыми» (headless), поскольку они не предоставляют пользовательский интерфейс, с которым можно взаимодействовать.

Сервисы рабочей роли обычно используются для выполнения фоновых задач (реализации `IHostedService`), для которых не требуется пользовательский интерфейс. Это могут быть задачи для запуска пакетных заданий, многократного выполнения задач по расписанию или для обработки событий с использованием шины сообщений. В следующем разделе мы создадим сервис рабочей роли для получения последних курсов валют из удаленного API, вместо того чтобы добавлять фоновую задачу в приложение ASP.NET Core.

34.2.1 Создание сервиса рабочей роли из шаблона

В этом разделе вы увидите, как создать базовый сервис рабочей роли из шаблона. Visual Studio включает в себя шаблон для создания сервисов рабочей роли: выберите **File > New > Project > Worker Service**. Можно создать аналогичный шаблон с помощью интерфейса командной строки .NET, выполнив команду `dotnet new worker`. Полученный в результате шаблон состоит из двух файлов C#:

- *Worker.cs* – это простая реализация класса `BackgroundService`, которая делает запись в журнал каждую секунду. Вы можете заменить этот класс собственной реализацией фонового сервиса, например воспользовавшись кодом из листинга 34.5;
- *Program.cs* – как и в типичном приложении ASP.NET Core, он содержит точку входа для вашего приложения, и именно там создается и запускается `IHost`. В отличие от типичного приложения ASP.NET Core, здесь вы также настраиваете контейнер внедрения зависимостей для своего приложения.

Листинг 34.6 Реализация `BackgroundService` по умолчанию для шаблона сервиса рабочей роли

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;
    public Worker(ILogger<Worker> logger)
    {
        _logger = logger;
    }

    protected override async Task ExecuteAsync(
        CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation(
                "Worker running at: {time}",
                DateTimeOffset.Now);
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

Служба Worker наследуется от `BackgroundService`

`ExecuteAsync` запускает основной цикл выполнения службы

Когда приложение закрывается, `CancellationToken` отменяется

Служба записывает сообщение в журнал каждую секунду, пока приложение не закроется

Наиболее заметное различие между шаблоном сервиса рабочей роли и шаблоном ASP.NET Core заключается в том, что `Program.cs` не использует API классов `WebApplicationBuilder` и `WebApplication` для минимального хостинга. Вместо этого он использует вспомогательный метод `Host.CreateDefaultBuilder()`, о котором вы узнали в главе 30, для создания `IHostBuilder`.

ПРИМЕЧАНИЕ NET 8 изменит шаблон сервиса рабочей роли, чтобы использовать новый тип, `HostApplicationBuilder`, аналог `WebHostBuilder`. `HostApplicationBuilder` обеспечивает знакомый

сценарий установки минимального хостинга для сервисов рабочей роли вместо использования подхода `IHostBuilder` на основе обратного вызова.

Вы настраиваете свои сервисы в `Program.cs` с помощью метода `ConfigureServices()` в `IHostBuilder`, как показано в листинге 34.7. Этот метод принимает лямбда-метод, который принимает два аргумента:

- объект `HostBuilderContext`. Этот объект контекста предоставляет `IConfiguration` вашего приложения через свойство `Configuration`, а `IHostEnvironment` – через свойство `HostingEnvironment`;
- объект `ISeviceCollection`. Вы добавляете свои службы в эту коллекцию так же, как добавляете их в `WebApplicationBuilder.Services` в типичных приложениях ASP.NET Core.

В следующем листинге показано, как настроить EF Core, типизированный клиент обменных курсов из главы 33 и фоновую службу, которая сохраняет обменные курсы в базу данных, как вы видели в разделе 34.1.2. Он использует операторы верхнего уровня C#, поэтому точка входа `static void Main` не отображается.

Листинг 34.7 `Program.cs` для рабочей службы, которая сохраняет обменные курсы с помощью EF Core

```
using Microsoft.EntityFrameworkCore;

IHost host = Host.CreateDefaultBuilder(args) <-- Создает IHostBuilder с использованием вспомогательного класса Host
    .ConfigureServices((hostContext, services) => <-- Конфигурирует ваш контейнер внедрения зависимостей
    {
        services.AddHttpClient<ExchangeRatesClient>(); <-- Доступ к IConfiguration можно получить через параметр HostBuilder-Context
        services.AddHostedService<ExchangeRatesHostedService>();

        var connectionString = hostContext.Configuration
            .GetConnectionString("SqlLiteConnection") <-- Собирает экземпляр IHost

        services.AddDbContext<AppDbContext>(options =>
            options.UseSqlite(connectionString));
    })
    .Build(); <-- Собирает экземпляр IHost

host.Run(); <-- Запускает приложение и ожидает завершения работы
```

Изменения в `Program.cs`, позволяющие использовать универсальный узел вместо минимального хостинга, являются наиболее очевидными различиями между сервисом рабочей роли и приложением ASP.NET Core, однако в файле проекта `.csproj` также есть некоторые важные различия. В следующем листинге показан файл проекта для сервиса рабочей роли, использующий `IHttpClientFactory` и EF Core, а также выделены некоторые различия с аналогичным приложением ASP.NET Core.

Листинг 34.8 Файл проекта для сервиса рабочей роли

```
<Project Sdk="Microsoft.NET.Sdk.Worker"> <!-- Сервисы рабочей роли используют другой тип Project SDK, в отличие от приложений ASP.NET Core -->

<PropertyGroup>
    <TargetFramework>net7.0</TargetFramework> <!-- Целевой фреймворк такой же, как и для приложений ASP.NET Core -->
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <UserSecretsId>5088-4277-B226-DC0A790AB790</UserSecretsId> <!-- Сервисы рабочей роли применяют конфигурацию, поэтому они могут использовать UserSecrets, как и приложения ASP.NET Core -->
</PropertyGroup>
<ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting"
        Version="7.0.0" /> <!-- Все сервисы рабочей роли должны явно добавить этот пакет. Приложения ASP.NET Core добавляют его неявно -->
    Если вы используете IHttpClientFactory, вам нужно будет добавить этот пакет в сервисы рабочей роли
<PackageReference Include="Microsoft.Extensions.Http"
        Version="7.0.0" />
<PackageReference Include="Microsoft.EntityFrameworkCore.Design"
        Version="7.0.0" PrivateAssets="All" />
<PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite"
        Version="7.0.0" /> <!-- Пакеты EF Core должны быть добавлены явно, как и для приложений ASP.NET Core -->
</ItemGroup>
</Project>
```

Некоторые части файла проекта одинаковы как для сервисов рабочей роли, так и для приложений ASP.NET Core:

- оба типа приложений должны указывать <TargetFramework>, например net7.0 для .NET 7;
- оба типа приложений используют систему конфигурации, поэтому вы можете использовать <UserSecretsId> для управления секретами в окружении разработки, как описано в главе 10;
- оба типа приложений должны явно добавлять ссылки на пакеты NuGet EF Core, чтобы использовать EF Core в приложении.

В шаблоне проекта также есть несколько отличий:

- атрибут Sdk элемента <Project> для сервиса рабочей роли должен выглядеть так: Microsoft.NET.Sdk.Worker, а для приложения ASP.NET Core – Microsoft.NET.Sdk.Web. Web SDK включает в себя неявные ссылки на дополнительные пакеты, которые обычно не требуются в сервисах рабочей роли;
- сервис рабочей роли *должен* включать явную ссылку на NuGet-пакет Microsoft.Extensions.Hosting при помощи элемента PackageReference. Этот пакет включает в себя универсальную реализацию IHost, используемую сервисами рабочей роли;
- возможно, вам потребуется включить дополнительные пакеты для использования той же функциональности по сравнению с приложением ASP.NET Core. Пример – пакет Microsoft.Extensions.

Http (который предоставляет `IHttpClientFactory`). На него неявно ссылаются в приложениях ASP.NET Core, но в сервисах рабочей роли это следует делать явно.

Запуск сервиса рабочей роли аналогичен запуску приложения ASP.NET Core: используйте команду `dotnet run` из командной строки или нажмите клавишу **F5** в Visual Studio. По сути, сервис рабочей роли – это просто консольное приложение (как и приложения ASP.NET Core), поэтому запускается так же.

Сервисы рабочей роли можно запускать практически в тех же местах, что и приложение ASP.NET Core, хотя, поскольку сервис рабочей роли не обрабатывает HTTP-трафик, некоторые параметры имеют больше смысла. В следующем разделе мы рассмотрим два поддерживаемых способа запуска вашего приложения: в качестве службы Windows или в качестве демона `systemd` Linux.

34.2.2 Запуск сервисов рабочей роли в промышленном окружении

В этом разделе вы узнаете, как запускать сервисы рабочей роли в промышленном окружении. Вы узнаете, как установить сервис рабочей роли в качестве службы Windows, чтобы операционная система отслеживала и запускала его автоматически, а также как подготовить свое приложение для установки в качестве демона `systemd` в Linux.

Сервисы рабочей роли, как и приложения ASP.NET Core, в основном представляют собой консольные приложения .NET Core. Разница состоит в том, что обычно они предназначены для приложений с длительным временем работы. Обычным подходом к запуску этих типов приложений в Windows является использование службы Windows или использование демона `systemd` в Linux.

ПРИМЕЧАНИЕ Также очень часто приложения запускают в облаке с применением контейнеров Docker или специализированных платформ, таких как Azure App Service. Процесс развертывания сервиса рабочей роли в этих управляемых службах обычно идентичен развертыванию приложения ASP.NET Core.

Добавить поддержку служб Windows или `systemd` очень просто благодаря двум дополнительным пакетам NuGet:

- *Microsoft.Extensions.Hosting.Systemd* – добавляет поддержку для запуска приложения в виде приложения `systemd`. Чтобы активировать интеграцию с `systemd`, вызовите метод `UseSystemd()` в своем `IHostBuilder` в файле `Program.cs`;
- *Microsoft.Extensions.Hosting.WindowsServices* – добавляет поддержку для запуска приложения в качестве службы Windows. Чтобы активировать интеграцию, вызовите метод `UseWindowsService()` для `IHostBuilder` в файле `Program.cs`.

Каждый из них добавляет один метод расширения к `IHostBuilder`, который обеспечивает соответствующую интеграцию при запуске в ка-

честве демона systemd или службы Windows. Например, в следующем листинге показано, как активировать поддержку службы Windows.

Листинг 34.9 Добавление поддержки службы Windows в сервис рабочей роли

```
IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) =>
{
    Services.AddHostedService<Worker>();
})
.UseWindowsService()   ←
.Build();

host.Run();
```

Настраиваем
сервис рабочей
роли, как обычно

Добавляем поддержку
запуска в качестве
службы Windows

Во время разработки или если вы запускаете свое приложение как консольное, использование метода `UseWindowsService()` ничего не дает; оно работает точно так же, как и без вызова метода. Однако теперь приложение можно установить в качестве службы Windows, поскольку у него есть необходимые средства интеграции для работы с системой служб Windows. Следующие шаги показывают, как установить сервис рабочей роли в качестве службы Windows.

- 1 Добавьте пакет `Microsoft.Extensions.Hosting.WindowsServices` в свое приложение с помощью Visual Studio, выполнив команду `dotnet add package Microsoft.Extensions.Hosting.WindowsServices` в папке проекта или добавив элемент `<PackageReference>` в файл с расширением `.csproj`:

```
<PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices"
Version="5.0.0" />
```

- 2 Добавьте вызов метода `UseWindowsService()` интерфейса `IHostBuilder`, как показано в листинге 34.9.
- 3 Опубликуйте приложение, как описано в главе 27. Из командной строки вы можете выполнить команду `dotnet publish -c Release` из папки проекта.
- 4 Откройте командную строку от имени администратора и установите приложение с помощью утилиты Windows `sc`. Вам необходимо указать путь к файлу опубликованного проекта с расширением `.exe` и имя, которое будет использоваться для службы, например `My Test Service`:
`sc create "My Test Service" BinPath="C:\path\to\MyService.exe".`
- 5 Вы можете управлять службой из панели управления службами в Windows, как показано на рис. 34.3. В качестве альтернативы, чтобы запустить службу из командной строки, выполните команду `sc start "My Test Service"` или, чтобы удалить службу, воспользуйтесь командой `sc delete "My Test Service"`.

После выполнения этих шагов сервис рабочей роли будет работать как служба Windows.

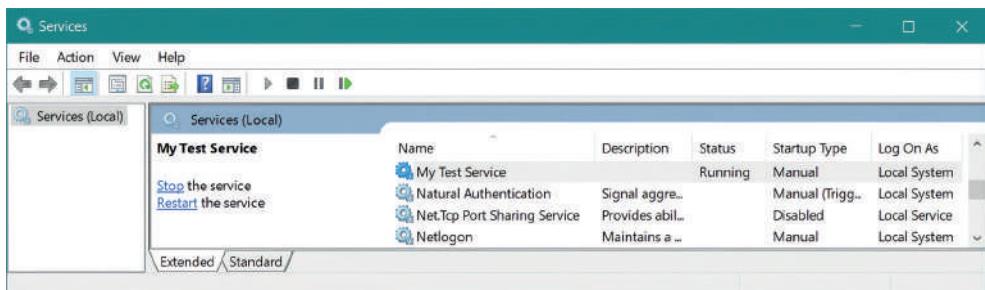


Рис. 34.3 Панель управления службами в Windows. После установки сервиса рабочей роли в качестве службы Windows с помощью утилиты `sc` вы можете управлять им отсюда. Это позволяет контролировать, когда служба Windows запускается и останавливается, с какой учетной записью пользователя запускается приложение и способы обработки ошибок

ПРИМЕЧАНИЕ Эти шаги – лишь минимум, необходимый для установки службы Windows. При работе в промышленном окружении нужно учитывать множество аспектов безопасности, которые здесь не рассматриваются. Дополнительные сведения см. в документе Microsoft: <http://mng.bz/Xdy9>.

Интересно отметить, что установка в качестве службы Windows или демона `systemd` не ограничивается только сервисами рабочей роли – точно так же можно установить приложение ASP.NET Core. Просто следуйте предыдущим инструкциям, добавьте вызов метода `UseWindowsService()` и установите приложение ASP.NET Core. Все это осуществляется благодаря тому, что функциональность ASP.NET Core строится непосредственно на универсальной функциональности `Host`.

Чтобы установить сервис рабочей роли в качестве демона `systemd`, можно выполнить аналогичный процесс, установив пакет Microsoft.Extensions.Hosting.Systemd и вызвав метод `UseSystemd()` для `IApplicationBuilder`. Для получения дополнительных сведений о настройке `systemd` см. раздел «Мониторинг приложения» в документации Microsoft: <http://mng.bz/yYDp>.

До сих пор в этой главе мы использовали интерфейс `IHostedService` и класс `BackgroundService` для выполнения задач, которые повторяются с интервалом, и вы видели, как установить сервисы рабочей роли в качестве приложений с длительным временем запуска, установив их как службу Windows.

В последнем разделе этой главы мы рассмотрим, как создавать более сложные расписания для фоновых задач, а также как повысить отказоустойчивость своего приложения, запустив несколько экземпляров сервисов рабочей роли. Для этого мы будем использовать стороннюю библиотеку Quartz.NET.

34.3 Координация фоновых задач с помощью Quartz.NET

В этом разделе вы узнаете, как использовать библиотеку планировщика с открытым исходным кодом Quartz.NET, как установить и настроить ее, а также как добавить фоновое задание для выполнения по расписанию. Вы также узнаете, как активировать кластеризацию для своих приложений, чтобы можно было запускать несколько экземпляров сервиса рабочей роли и распределять задания между ними.

Все фоновые задачи, которые вы видели до сих пор в этой главе, повторяют задачу с интервалом до бесконечности с момента запуска приложения. Однако иногда нужно больше контроля. Может быть, вы всегда хотите запускать приложение на 15-й минуте каждого часа. Или, может быть, вы хотите запустить задачу только во второй вторник месяца в 3 часа ночи. Кроме того, возможно, вам нужно запустить несколько экземпляров своего приложения для обеспечения избыточности, но быть уверенным, что только одна из служб выполняет задачу одновременно.

Конечно, можно было бы самостоятельно встроить всю эту дополнительную функциональность в приложение, но есть отличные библиотеки, у которых уже все это есть. Две наиболее известные из них в .NET-пространстве – это Hangfire (www.hangfire.io) и Quartz.NET (www.quartz-scheduler.net).

Hangfire – это библиотека с открытым исходным кодом, у которой также имеется вариант подписки «Pro». Одной из ее самых популярных функций является пользовательский интерфейс панели инструментов, который показывает состояние всех ваших запущенных задач, историю каждой задачи и любые возникающие ошибки.

Quartz.NET – это библиотека с полностью открытым исходным кодом. По сути, она предлагает расширенную версию функциональности класса `BackgroundService`. У нее имеются обширные возможности для планирования, а также она поддерживает работу в кластерном окружении, где несколько экземпляров вашего приложения координируют распределение заданий.

ПРИМЕЧАНИЕ Quartz.NET основана на аналогичной библиотеке Java под названием Quartz Scheduler. При поиске информации о Quartz.NET убедитесь, что это именно та библиотека, которая вам нужна.

Quartz.NET основана на четырех основных концепциях:

- **задания** – это фоновые задачи, реализующие вашу логику;
- **триггеры** – определяют, когда задание будет запускаться по расписанию, например «каждые пять минут» или «каждый второй вторник». У задания может быть несколько триггеров;
- **фабрика заданий** – фабрика заданий отвечает за создание экземпляров заданий. Quartz.NET интегрируется с контейнером внедрения зависимостей ASP.NET Core, поэтому вы можете использовать внедрение зависимостей в классах заданий;

- **планировщик** – планировщик Quartz.NET отслеживает триггеры в вашем приложении, создает задания, используя фабрику заданий, и запускает их. Обычно он работает как `IHostedService` на протяжении всего жизненного цикла вашего приложения.

Фоновые сервисы и задания cron

Обычно задания cron используются для выполнения задач по расписанию в Linux, а в Windows используется планировщик заданий. Они применяются для периодического запуска приложения или файла сценария, что обычно является кратковременной задачей.

Напротив, приложения .NET Core, использующие фоновые сервисы, спроектированы как приложения с длительным жизненным циклом, даже если они используются только для выполнения задач по расписанию. Это позволяет приложению корректировать свое расписание по мере необходимости или выполнять оптимизацию. Кроме того, долговечность означает, что вашему приложению не нужно просто выполнять задачи по расписанию. Оно может реагировать на специальные события, такие как события в очереди сообщений.

Конечно, если вам не нужны эти возможности и не нужно приложение с длительным временем запуска, можно использовать .NET Core в сочетании с заданиями cron. Вы можете создать простое консольное приложение .NET, которое запускает задачу, а затем закрывается, и запланировать его периодическое выполнение в качестве задания cron. Выбор за вами!

В этом разделе я покажу, как установить Quartz.NET и настроить фоновый сервис для запуска по расписанию. Затем я объясню, как активировать кластеризацию, чтобы можно было запускать несколько экземпляров приложения и распределять задания между ними.

34.3.1 Установка Quartz.NET в приложение ASP.NET Core

В этом разделе я покажу, как установить планировщик Quartz.NET в приложение ASP.NET Core. Quartz.NET будет работать в фоновом режиме так же, как и реализации `IHostedService`. Фактически Quartz.NET использует абстракции `IHostedService` для планирования и выполнения заданий.

ОПРЕДЕЛЕНИЕ Задание в Quartz.NET – это выполняемая задача, реализующая интерфейс `IJob`. Здесь вы определяете логику, которую будут выполнять ваши задачи.

Quartz.NET можно установить в любое приложение .NET 7, поэтому вы также увидите, как установить Quartz.NET в сервис рабочей роли. Мы установим необходимые зависимости и настроим планировщик Quartz.NET для работы в качестве фонового сервиса в сервисе рабочей роли. В разделе 34.3.2 мы преобразуем задачу загрузчика курсов обмена валют из раздела 34.1 в интерфейс Quartz.NET `IJob` и настроим триггеры для запуска по расписанию.

ПРИМЕЧАНИЕ Инструкции в этом разделе можно использовать для установки Quartz.NET либо в сервис рабочей роли, либо в полноценное приложение ASP.NET Core. Единственная разница состоит в том, используете вы универсальный хост в файле Program.cs или WebApplicationBuilder.

Чтобы установить Quartz.NET, выполните следующие действия.

- 1 Установите пакет Quartz.AspNetCore в свой проект, выполнив команду `dotnet add package Quartz.Extensions.Hosting`, используя обозреватель NuGet в Visual Studio или добавив элемент `<PackageReference>` в файл проекта:

```
<PackageReference Include="Quartz.Extensions.Hosting" Version="3.5.0" />.
```

- 2 Добавьте планировщик Quartz.NET `IHostedService`, вызвав метод `AddQuartzHostedService()` для `IServiceCollection` в методе `Configure-Services` (или для `WebApplicationBuilder.Services`), как показано ниже. Задайте для `WaitForJobsToComplete` значение `true`, чтобы ваше приложение ожидало завершения всех текущих заданий при завершении работы.

```
services.AddQuartzHostedService(q => q.WaitForJobsToComplete = true); .
```

- 3 Настройте необходимые сервисы Quartz.NET. В следующем листинге мы настраиваем фабрику заданий Quartz.NET для получения реализаций заданий из контейнера внедрения зависимостей с жизненным циклом `Scoped` и добавляем требуемый сервис.

Листинг 34.10 Настройка Quartz.NET

```
using Quartz;

IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) => {
        Настраиваем
        Quartz.NET
        для получения
        заданий из
        контейнера
        внедрения за-
        висимостей
        {
            services.AddQuartz(q => {
                Регистрируем сервисы Quartz.
                NET в контейнере внедрения
                зависимостей
                {
                    q.UseMicrosoftDependencyInjectionJobFactory();
                });
                Добавляем
                IHostedService от
                Quartz.NET, который
                запускает планиров-
                щик Quartz.NET
                services.AddQuartzHostedService(
                    q => q.WaitForJobsToComplete = true);
            })
            .Build();
        host.Run();
    });

```

Эта конфигурация регистрирует все необходимые компоненты Quartz.NET, поэтому теперь вы можете запускать свое приложение, используя команду `dotnet run` или нажав клавишу **F5** в Visual Studio. Когда ваше приложение запускается, `IHostedService` запускает свой планировщик, как показано на рис. 34.4. У нас еще нет заданий для запуска, поэтому планировать пока нечего.

Quartz.NET использует хранилище в памяти для отслеживания фоновых задач и расписаний

Quartz.NET по умолчанию работает в некластеризованном режиме, поэтому каждый запущенный экземпляр вашего приложения независим

Для этого приложения не было настроено никаких фоновых задач или триггеров

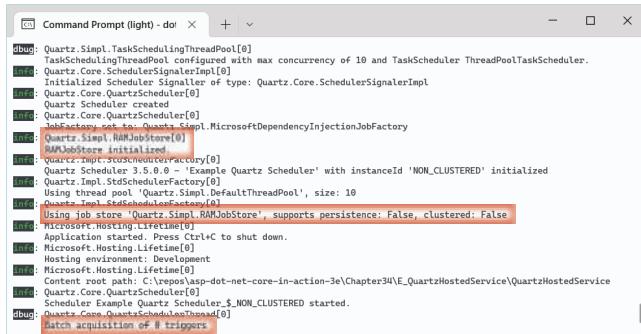


Рис. 34.4 Планировщик Quartz.NET запускается при запуске приложения и выводит свою конфигурацию. В конфигурации по умолчанию список заданий и их расписаний хранится в памяти и запускается в некластеризованном режиме. В этом примере видно, что не было зарегистрировано ни одного задания или триггера, поэтому планировщику еще нечего планировать

СОВЕТ Рекомендуется запускать приложение *до* добавления каких-либо заданий. Это позволяет проверить, правильно ли вы установили и настроили Quartz.NET, прежде чем перейти к более сложной конфигурации.

Планировщик без заданий для планирования не очень полезен, поэтому в следующем разделе мы создадим задание и добавим триггер, чтобы запускать его по таймеру.

34.3.2 Настройка задания для запуска по расписанию с помощью Quartz.NET

В разделе 34.1 мы создали экземпляр интерфейса `IHostedService`, который скачивает курсы обмена валют из удаленного сервиса и сохраняет результаты в базе данных с помощью EF Core. В этом разделе вы увидите, как создать аналогичный экземпляр интерфейса Quartz.NET, `IJob`, и настроить его для запуска по расписанию.

В следующем листинге показана реализация `IJob`, загружающая последние курсы обмена валют из удаленного API с помощью типизированного клиента `ExchangeRatesClient`. Затем результаты сохраняются с использованием `AppDbContext`.

Листинг 34.11 Экземпляр `IJob` для загрузки и сохранения курсов обмена валют

```
→ public class UpdateExchangeRatesJob : IJob
{
    Задания     private readonly ILogger<UpdateExchangeRatesJob> _logger;
    Quartz.NET   private readonly ExchangeRatesClient _typedClient;
    должны       private readonly AppDbContext _dbContext;
    реализовы-  public UpdateExchangeRatesJob(
    вать интер-      ILogger<UpdateExchangeRatesJob> logger,
    фейс IJob        ExchangeRatesClient typedClient,
```

Вы можете использовать стандартное внедрение зависимостей для внедрения любых зависимостей

```

IJob требует,          AppDbContext dbContext)
чтобы вы          {
реализо-          _logger = logger;
вали един-          _typedClient = typedClient;
ственный          _dbContext = dbContext;
асинхрон-
ный метод
Execute          }

    public async Task Execute(IJobExecutionContext context)
    {
        _logger.LogInformation("Fetching latest rates");
        var latestRates = await _typedClient.GetLatestRatesAsync(); <|-->

        _dbContext.Add(latestRates);
        await _dbContext.SaveChangesAsync();

        _logger.LogInformation("Latest rates updated");
    }
}

```

Сохраняем курсы в базу **данных**

Вы можете исполь- зовать стандарт-
ное внедрение зависимостей для
внедрения любых зависимостей

Скачиваем кур-сы из удален-
ного API

Функционально IJob из листинга 34.11 выполняет задачу, аналогичную реализации класса BackgroundService из листинга 34.5, с несколькими заметными исключениями:

- IJob только определяет задачу, которую нужно выполнить; он не определяет информацию о времени. В реализации BackgroundService мы также должны были контролировать частоту выполнения задачи;
- новый экземпляр IJob создается каждый раз при выполнении задания. Напротив, реализация BackgroundService создается только один раз, и его метод Execute вызывается только один раз;
- мы можем внедрить зависимости с жизненным циклом Scoped непосредственно в реализацию IJob. Чтобы использовать зависимости с жизненным циклом Scoped в реализации IHostedService, нам пришлось вручную создать собственную область и использовать локацию сервисов для загрузки зависимостей. Quartz.NET делает это за нас, позволяя использовать чистое внедрение конструктора. Каждый раз, когда задание выполняется, создается новая область, которая используется для создания нового экземпляра IJob.

IJob определяет, что выполнять, но не определяет, когда это делать. Для этого Quartz.NET использует *триггеры*. Триггеры можно использовать для определения произвольно сложных интервалов времени, в течение которого должно быть выполнено задание. Например, можно указать время начала и окончания, сколько раз повторять и временные интервалы, когда задание должно или не должно запускаться (например, только с 9:00 до 17:00 с понедельника по пятницу).

В следующем листинге мы зарегистрируем UpdateExchangeRatesJob в контейнере внедрения зависимостей, используя метод AddJob<T>(), и предоставляем уникальное имя для идентификации задания. Мы также настраиваем триггер, который срабатывает немедленно, а затем каждые пять минут, пока приложение не завершит работу.

Листинг 34.12 Настройка IJob и триггера

```

using Quartz;

IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) =>
{
    services.AddQuartz(q =>
    {
        q. UseMicrosoftDependencyInjectionJobFactory();

        var jobKey = new JobKey("Update exchange rates");
        q.AddJob<UpdateExchangeRatesJob>(opts =>
            opts.WithIdentity(jobKey));

        q.AddTrigger(opts => opts
            .ForJob(jobKey)
            .WithIdentity(jobKey.Name + " trigger")
            .StartNow()
            .WithSimpleSchedule(x => x
                .WithInterval(TimeSpan.FromMinutes(5))
                .RepeatForever()));
    });
    services.AddQuartzHostedService(
        q => q.WaitForJobsToComplete = true);
});
.Build();
host.Run();

```

Создаем уникальный ключ задания, чтобы связать его с триггером

Указываем уникальное имя триггера, которое будет использоваться при журнализации и в классерных сценариях

Запускаем триггер, как только запускается планировщик Quartz.NET при запуске приложения

Добавляем IJob в контейнер внедрения зависимостей и связываем его с ключом задания

Регистрируем триггер для IJob с помощью ключа задания

Запускаем триггер каждые пять минут, пока приложение не завершит работу

Простые триггеры, такие как определенное здесь расписание, довольно распространены, но вы также можете получить более сложные конфигурации, используя другое расписание. Например, следующая конфигурация делает так, чтобы триггер запускался каждую неделю, в пятницу в 17:30:

```

q.AddTrigger(opts => opts
    .ForJob(jobKey)
    .WithIdentity("Update exchange rates trigger")
    .WithSchedule(CronScheduleBuilder
        .WeeklyOnDayAndHourAndMinute(DayOfWeek.Friday, 17, 30)));

```

С помощью Quartz.NET можно настроить широкий спектр триггеров на основе времени и календаря. Вы также можете контролировать, как Quartz.NET обрабатывает пропущенные триггеры, то есть триггеры, которые должны были сработать, но ваше приложение в то время не было запущено. Чтобы увидеть подробное описание параметров конфигурации триггера и другие примеры, см. документацию по Quartz.NET: www.quartz-scheduler.net/documentation/.

СОВЕТ Распространенная проблема, с которой приходится сталкиваться при выполнении заданий с длительным временем выполнения, заключается в том, что Quartz.NET будет продолжать запускать новые экземпляры задания при срабатывании триггера, даже если задание уже запущено. Чтобы этого избежать, нужно указать Quartz.NET не запускать другой экземпляр, декорировав свою реализацию `IJob` с помощью атрибута `[DisallowConcurrentExecution]`.

Возможность настраивать расширенные расписания, простое использование внедрения зависимостей в фоновых задачах и отделение заданий от триггеров – для меня этого достаточно, чтобы рекомендовать Quartz.NET, если у вас есть что-то большее, чем самые простые потребности фоновых сервисов. Однако настоящий переломный момент наступает, когда вам нужно масштабировать приложение для увеличения избыточности или производительности – именно тогда возможности кластеризации Quartz.NET проявляются во всей своей красе.

34.3.3 Использование кластеризации для повышения избыточности фоновых задач

В этом разделе вы узнаете, как настроить Quartz.NET для сохранения его конфигурации в базе данных. Это необходимый шаг для активации кластеризации, чтобы несколько экземпляров вашего приложения могли координировать запуск заданий.

По мере того как ваши приложения становятся более популярными, вы можете обнаружить, что вам нужно запускать больше экземпляров приложения для обработки получаемого ими трафика. Если приложение ASP.NET Core не хранит состояние, процесс масштабирования будет относительно простым: чем больше у вас есть приложений, тем больше трафика вы сможете обработать при прочих равных условиях.

Однако масштабирование приложений, использующих `IHostedService` для запуска фоновых задач, может оказаться не такой простой задачей. Например, представьте, что ваше приложение включает в себя фоновый сервис, который мы создали в разделе 34.1.2, сохраняющий курсы валют в базе данных каждые пять минут. Когда вы запускаете один экземпляр приложения, задача запускается каждые пять минут, как и ожидалось.

Но что произойдет, если масштабировать приложение и запустить десять его экземпляров? Каждое из этих приложений будет запускать `BackgroundService`, и все они будут обновляться каждые пять минут с момента запуска каждого экземпляра!

Один из вариантов – переместить `BackgroundService` в отдельный сервис рабочей роли. Затем вы можете продолжить масштабирование приложения ASP.NET Core для обработки трафика по мере необходимости, но развернуть один экземпляр сервиса рабочей роли. Поскольку будет запущен только один экземпляр `BackgroundService`, курсы обмена валют снова будут обновляться по правильному расписанию.

СОВЕТ Разные требования к масштабированию, как в этом примере, являются одной из самых подходящих причин для разделения больших приложений на более мелкие микросервисы. Однако такое разделение приложения связано с накладными расходами на сопровождение, поэтому подумайте о компромиссах, если решите пойти по данному пути. Чтобы узнать больше об этом компромиссе, я рекомендую книгу «Микросервисы в .NET Core», 2-е изд., Кристиана Хорсдала Гаммельгаарда (Manning, 2021).

Однако если вы выберете этот путь, то добавьте жесткое ограничение, согласно которому у вас может быть только один экземпляр сервиса рабочей роли. Если вам нужно будет запустить больше экземпляров для обработки дополнительной нагрузки, вы застрянете.

Альтернативным вариантом принудительного использования единого сервиса является использование *кластеризации*. Кластеризация позволяет запускать несколько экземпляров приложения, а задачи распределяются между всеми его экземплярами. Quartz.NET обеспечивает кластеризацию, используя базу данных в качестве хранилища. Когда триггер указывает, что задание необходимо выполнить, планировщики Quartz.NET в каждом приложении пытаются получить блокировку для выполнения задания, как показано на рис. 34.5. Только одно приложение может успешно получить блокировку, и это гарантирует, что только одно приложение обрабатывает триггер для IJob.

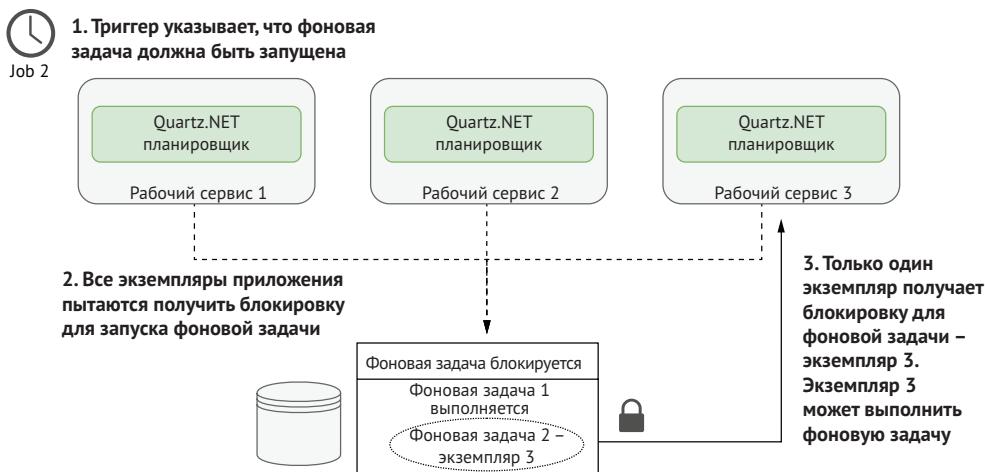


Рис. 34.5 Использование кластеризации позволяет Quartz.NET масштабироваться горизонтально. Для этого используется база данных, которая гарантирует, что только один экземпляр приложения обрабатывает триггер в момент времени. Это позволяет запускать несколько экземпляров вашего приложения для выполнения требований масштабируемости

Quartz.NET полагается на базу данных для долговременного хранения данных. Он хранит описания заданий и триггеров в базе данных, включая время последнего срабатывания триггера. Это функции бло-

кировки базы данных, которые гарантируют, что только одно приложение может выполнять задачу одновременно.

СОВЕТ Можно активировать долговременное хранение без использования кластеризации. Это позволяет планировщику Quartz.NET корректно обработать пропущенные триггеры.

В следующем листинге показано, как активировать долговременное хранение для Quartz.NET и, кроме того, кластеризацию. В этом примере данные хранятся на сервере MS SQL Server (или LocalDB), но Quartz.NET поддерживает множество других баз данных. В данном примере используются рекомендуемые значения для активации кластеризации и долговременного хранения, как указано в документации.

СОВЕТ В документации Quartz.NET обсуждается множество элементов управления параметрами конфигурации для долговременного хранения: <http://mng.bz/PP0R>. Чтобы использовать рекомендуемый сериализатор JSON для сохранения, необходимо также установить NuGet-пакет Quartz.Serialization.Json.

Листинг 34.13 Активация долговременного хранения и кластеризации для Quartz.NET

```
using Quartz;
IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) => {
        var connectionString = Configuration
            .GetConnectionString("DefaultConnection");
        services.AddQuartz(q => {
            q.SchedulerId = "AUTO";
            q.UseMicrosoftDependencyInjectionJobFactory();
        });
        q.UsePersistentStore(s => {
            s.UseSqlServer(connectionString);
            s.UseClustering();
            s.UseProperties = true;
            s.UseJsonSerializer();
        });
        var jobKey = new JobKey("Update_exchange_rates");
        q.AddJob<UpdateExchangeRatesJob>(opts =>
            opts.WithIdentity(jobKey));
        q.AddTrigger(opts => opts
            .ForJob(jobKey))
    });
}

Получаем строку подключения для своей базы данных из конфигурации

Активируем долговременное хранение в базе данных для данных планировщика Quartz.NET

Активирует кластеризацию между несколькими экземплярами приложения
```

Конфигурация идентична как для приложений ASP.NET Core, так и для сервисов рабочей роли

Каждый экземпляр приложения должен иметь уникальный SchedulerId. Значение AUTO позаботится об этом за вас

Сохраняем данные планировщика в базе данных SQL Server (или LocalDb)

Добавляет рекомендуемую конфигурацию для сохранения задания

```

        .WithIdentity(jobKey.Name + " trigger")
        .StartNow()
        .WithSimpleSchedule(x => x
            .WithInterval(TimeSpan.FromMinutes(5))
            .RepeatForever())
    );
});

services.AddQuartzHostedService(
    q => q.WaitForJobsToComplete = true);
})
    .Build();
host.Run();

```

В этой конфигурации Quartz.NET хранит список заданий и триггеров в базе данных и использует блокировку базы данных, чтобы гарантировать, что только один экземпляр вашего приложения обрабатывает триггер и запускает соответствующее задание.

ВНИМАНИЕ! SQLite не поддерживает примитивы блокировки базы данных, необходимые для кластеризации. Вы можете использовать SQLite в качестве долговременного хранилища, но не сможете использовать кластеризацию.

Quartz.NET хранит данные в вашей базе данных, но не пытается создавать таблицы, которые использует сам. Нужно вручную добавлять необходимые таблицы. Quartz.NET предоставляет SQL-скрипты на сайте GitHub для всех поддерживающих типов серверов баз данных, включая MS SQL Server, SQLite, PostgreSQL, MySQL и многие другие: <http://mng.bz/JDeZ>.

СОВЕТ Если вы используете миграции EF Core для управления своей базой данных, то предлагаю использовать их даже для таких специальных сценариев. В примере кода для этой главы можно увидеть миграцию, которая создает необходимые таблицы с помощью скриптов Quartz.NET.

Кластеризация – одна из тех расширенных функций, которая необходима только в начале масштабирования приложения, но это важный инструмент, который нужно иметь наготове. Она дает вам возможность безопасно масштабировать свои сервисы по мере добавления новых заданий. Однако следует помнить о некоторых важных вещах, поэтому я предлагаю прочитать предупреждения, приведенные в документации Quartz.NET: <http://mng.bz/a0zj>.

На этом мы подошли к концу главы, посвященной фоновым сервисам. В последних главах этой книги я опишу один важный аспект веб-разработки, который иногда, несмотря на самые лучшие намерения, часто оставляют напоследок: тестирование. Вы узнаете, как писать простые модульные тесты для своих классов, как проектировать приложения с возможностью тестирования и как создавать интеграционные тесты, чтобы протестировать все приложение.

Резюме

- Вы можете использовать интерфейс `IHostedService` для запуска задач в фоновом режиме приложений ASP.NET Core. Вызовите метод `AddHostedService<T>()`, чтобы добавить реализацию `T` в контейнер внедрения зависимостей. `IHostedService` полезен для реализации задач с длительным временем выполнения;
- как правило, для создания экземпляра `IHostedService` нужно наследовать от класса `BackgroundService`, поскольку он реализует передовые практики, необходимые для задач с длительным временем выполнения. Вы должны переопределить единственный метод `ExecuteAsync`, который вызывается при запуске приложения. Вы должны выполнять свои задачи в этом методе, пока предоставленный `CancellationToken` не укажет на то, что приложение завершает работу;
- вы можете вручную создавать области внедрения зависимостей с помощью `IServiceProvider.CreateScope()`. Это полезно для доступа к сервисам с жизненным циклом `Scoped` в рамках компонента с жизненным циклом `Singleton`, например из реализации `IHostedService`;
- *сервис рабочей роли* – это приложение .NET Core, использующее обобщенный интерфейс `IHost`, но оно не включает в себя библиотеки ASP.NET Core для обработки HTTP-запросов. Обычно оно использует меньший объем памяти и дискового пространства, по сравнению с аналогами ASP.NET Core;
- сервисы рабочей роли используют те же системы журналирования, конфигурации и внедрения зависимостей, что и приложения ASP.NET Core. Однако они не используют файл `Startup.cs`, поэтому нужно настраивать свои сервисы внедрения зависимостей в `IHostBuilder.ConfigureServices()`;
- чтобы запустить сервис рабочей роли или приложение ASP.NET Core как службу Windows, добавьте пакет `Microsoft.Extensions.Hosting.WindowsServices` и вызовите метод `UseWindowsService()` интерфейса `IHostBuilder`. Вы можете установить свое приложение и управлять им с помощью утилиты Windows, `sc`;
- чтобы установить демон `systemd` в Linux, добавьте пакет `Microsoft.Extensions.Hosting.Systemd` и вызовите метод `AddSystemd()` для `IHostBuilder`. Оба пакета ничего не делают при запуске приложения в качестве консольного приложения, что отлично подходит для тестирования приложения;
- `Quartz.NET` запускает задания на основе триггеров с использованием расширенных расписаний. Он основан на реализации `IHostedService`, чтобы добавлять дополнительные функции и масштабируемость. Вы можете установить `Quartz`, добавив NuGet-пакет `Quartz.AspNetCore` и вызывав методы `AddQuartz()` и `AddQuartzHostedService()` в `ConfigureServices()`;

- вы можете создать задание Quartz.NET, реализовав интерфейс `IJob`. Он требует реализации единственного метода, `Execute`. Можно активировать внедрение зависимостей для задания, вызвав `UseMicrosoftDependencyInjectionScopedJobFactory` в методе `AddQuartz()`. Это позволяет напрямую внедрять сервисы с жизненными циклами `Scoped` или `Transient` в свое задание, не создавая собственных областей;
- вы должны зарегистрировать свое задание `T` с помощью внедрения зависимостей, вызвав метод `AddJob<T>()` и указав `JobKey`. Можно добавить ассоциированный триггер, вызвав метод `AddTrigger()` и предоставив `JobKey`. Для триггеров доступно большое количество расписаний, чтобы контролировать время выполнения задания;
- по умолчанию триггеры продолжат порождать новые экземпляры задания настолько часто, насколько это необходимо. В случае с заданиями с длительным временем выполнения и коротким интервалом запуска это приведет к тому, что многие экземпляры вашего задания будут работать одновременно. Если вам нужен триггер, чтобы выполнять задание, только когда экземпляр еще не запущен, декорируйте задание атрибутом `[DisallowConcurrentExecution]`;
- Quartz.NET поддерживает долговременное хранение в базе данных при выполнении триггеров. Чтобы активировать его, вызовите метод `UsePersistentStore()` в методе конфигурации `AddQuartz()` и настройте базу данных, применяя, например, метод `UseSqlServer()`. Используя долговременное хранение, Quartz.NET может сохранять сведения о заданиях и триггерах между перезапусками приложения;
- активация долговременного хранения также допускает использование кластеризации. Кластеризация позволяет нескольким приложениям, использующим Quartz.NET, координировать свои действия, так чтобы задания распределялись по нескольким планировщикам. Чтобы активировать ее, сначала активируйте долговременное хранение, а затем вызовите метод `UseClustering()`. SQLite не поддерживает кластеризацию из-за ограничений самой базы данных.

35

Тестирование приложений с помощью xUnit

В этой главе:

- тестирование в ASP.NET Core;
- создание проектов модульного тестирования с помощью xUnit;
- создание тестов с атрибутами [Fact] и [Theory].

Когда я только начинал программировать, то не понимал преимуществ автоматизированного тестирования. Для этого требовалось писать так много кода – не лучше было бы работать над новыми функциями? Я оценил эти преимущества, только когда мои проекты стали расти. Вместо того чтобы вручную запускать приложение и тестировать каждый сценарий, я мог бы нажать кнопку «Воспроизвести» в наборе тестов и автоматически протестировать свой код.

Тестирование повсеместно признано хорошей практикой, но то, как оно вписывается в ваш процесс разработки, часто может превратиться в религиозные дебаты. Сколько тестов вам нужно? Если покрытие вашей кодовой базы составляет менее 100 %, то будет ли этого достаточно? Следует ли писать тесты до, во время или после основного кода?

В этой главе не рассматривается ни один из этих вопросов. Вместо этого я сосредоточусь на *механике тестирования* приложения ASP.NET Core. Я покажу, как использовать изолированные *модульные тесты* для проверки изолированного поведения ваших сервисов, как тестировать контроллеры API и специальное промежуточное ПО и как создавать *интеграционные тесты*, которые одновременно проверяют несколько компонентов вашего приложения.

СОВЕТ Для более широкого обсуждения тестирования или если вы новичок в модульном тестировании, см. книгу «Искусство модульного тестирования», 3-е изд., Роя Ошерова (Manning, 2024).

Если вы хотите изучить передовые практики модульного тестирования на примерах C#, см. книгу «Модульное тестирование: принципы, практики и паттерны» Владимира Хорикова (Manning, 2020). «Эффективное тестирование программного обеспечения: Руководство для разработчиков» Маурисио Аниче (Manning, 2020) использует примеры на Java, но охватывает широкий спектр тем и методов. Кроме того, для более подробного изучения тестирования с помощью xUnit в .NET Core см. книгу «.NET Core в действии» Дастина Мецгара (Manning, 2023).

В разделе 35.1 я расскажу о фреймворке тестирования в .NET SDK и о том, как использовать его для создания приложений для модульного тестирования. Я опишу задействованные компоненты, включая SDK и сами фреймворки для тестирования, такие как xUnit и MSTest. Наконец, я рассмотрю некоторые термины, которые буду использовать в этой главе и главе 36.

В данной главе основное внимание уделяется механизму начала работы с xUnit. Вы узнаете, как создавать проекты модульных тестов, ссылаясь на классы в других проектах и запускать тесты с помощью Visual Studio или интерфейса командной строки (CLI) .NET. Вы создадите тестовый проект и будете использовать его для проверки поведения базовой службы конвертации валют. Наконец, вы напишете несколько простых модульных тестов, которые проверяют, возвращает ли служба ожидаемые результаты и выдает ли исключения, когда вы этого ожидаете.

Начнем с рассмотрения общего ландшафта тестирования ASP.NET Core, доступных вариантов и задействованных компонентов.

35.1 Тестирование в ASP.NET Core

В этом разделе вы узнаете об основах тестирования в ASP.NET Core, о различных типах тестов, которые вы можете написать, таких как модульные и интеграционные тесты, а также о том, почему следует писать оба типа. Наконец, вы увидите, как тестирование вписывается в ASP.NET Core.

Если у вас есть опыт создания приложений с версией .NET Framework или приложений для мобильных устройств с Xamarin, то, возможно, у вас уже имеется опыт работы с фреймворками для модульного тестирования. Если вы создавали приложения в Visual Studio, шаги по созда-

нию тестового проекта в разных фреймворках (xUnit, NUnit, MSTest) различались бы, и для запуска тестов в Visual Studio часто требовалось установить подключаемый модуль. Кроме того, запуск тестов из командной строки также отличался в зависимости от фреймворка.

С помощью .NET SDK тестирование в ASP.NET Core и .NET Core теперь стало одной из основных возможностей наравне со сборкой, восстановлением пакетов и запуском вашего приложения. Так же, как вы можете выполнить команду `dotnet build` для сборки проекта или `dotnet run` для его выполнения, вы можете использовать команду `dotnet test` для выполнения тестов в тестовом проекте, независимо от используемого фреймворка.

Команда `dotnet test` использует базовый набор средств разработки .NET для выполнения тестов для определенного проекта. Это то же самое, когда вы запускаете свои тесты с помощью выполняющего тесты механизма Visual Studio, поэтому какой бы подход вы ни предпочли, результаты будут одинаковыми.

Тестовые проекты – это консольные приложения, содержащие несколько *тестов*. Обычно тест представляет собой метод, который оценивает, ведет ли себя определенный класс в вашем приложении так, как ожидалось. Тестовый проект обычно зависит как минимум от трех компонентов:

- набор средств разработки для тестирования на платформе .NET;
- фреймворк для модульного тестирования, такой как xUnit, NUnit, Fixie или MSTest;
- адаптер тестирования для выбранного вами фреймворка тестирования, чтобы вы могли выполнять тесты, вызвав команду `dotnet test`.

Эти зависимости представляют собой обычные пакеты NuGet, которые можно добавить в проект, но они позволяют подключиться к команде `dotnet test` и выполняющему тесты механизму Visual Studio. В следующем разделе вы увидите пример файла с расширением `.csproj` из тестового приложения.

Обычно тест состоит из метода, который запускает небольшую часть вашего приложения изолированно и проверяет, что оно имеет нужное поведение. Если бы вы тестировали класс `Calculator`, то могли бы использовать тест, который проверяет, что при передаче значений 1 и 2 методу `Add()` возвращается ожидаемый результат, 3.

Вы можете написать множество небольших изолированных тестов, похожих этому, для классов своего приложения, чтобы убедиться, что каждый компонент работает правильно, независимо от других компонентов. Подобные небольшие изолированные тесты называются *модульными*.

Используя фреймворк ASP.NET Core, можно создавать приложения, которые легко тестировать с помощью модульных тестов; вы можете протестировать некоторые аспекты своих контроллеров отдельно от фильтров действий и привязки модели, потому что фреймворк:

- избегает статических типов;
- использует интерфейсы вместо конкретных реализаций;

- имеет модульную архитектуру, например вы можете протестировать свои контроллеры отдельно от фильтров действий и привязки модели.

Но то, что все ваши компоненты работают независимо друг от друга, не означает, что они будут работать, когда вы соберете их вместе. Для этого нужны *интеграционные тесты*, которые проверяют, как несколько компонентов взаимодействуют между собой.

Определение интеграционного теста – еще один довольно спорный вопрос, но я рассматриваю интеграционные тесты как ситуации, когда вы тестируете несколько компонентов вместе или большие вертикальные срезы своего приложения: тестируете класс диспетчера пользователей, который может сохранять значения в базе данных, например, или проверяете, что запрос, сделанный к конечной точке проверки работоспособности, возвращает ожидаемый ответ. Интеграционные тесты не обязательно включают *все* приложение, но они определенно используют больше компонентов, по сравнению с модульными тестами.

ПРИМЕЧАНИЕ Я не рассматриваю тесты пользовательского интерфейса, которые, например, взаимодействуют с браузером, чтобы обеспечить настояще сквозное автоматизированное тестирование. Selenium (www.seleniumhq.org) и Cypress (www.cypress.io) – два самых известных инструмента для тестирования пользовательского интерфейса.

Когда дело доходит до интеграционного тестирования, у ASP.NET Core есть в рукаве пара трюков. Вы можете использовать пакет Test Host для запуска внутрипроцессного сервера ASP.NET Core, на который можно отправлять запросы и проверять ответы. Это избавит вас от головной боли при попытке запустить веб-сервер в другом процессе, убедитесь, что порты доступны и т. д., но тем не менее позволяет использовать все приложение.

С другой стороны, поставщик баз данных EF Core SQLite в памяти позволяет изолировать тесты от базы данных. Взаимодействие с базой данных и ее настройка часто являются одним из самых сложных аспектов автоматизации тестов, поэтому этот поставщик позволяет полностью обойти данную проблему. Вы увидите, как его использовать, в главе 36.

Самый простой способ разобраться, что такое тестирование, – это опробовать, как оно работает, поэтому в следующем разделе вы создадите свой первый тестовый проект и воспользуетесь им для написания модульных тестов для простого специального сервиса.

35.2 Создание первого тестового проекта с xUnit

Как я уже писал в разделе 35.1, для создания тестового проекта необходимо использовать фреймворк для тестирования. У вас есть много вариантов, таких как NUnit или MSTest, но чаще всего с .NET Core используется фреймворк xUnit (<https://xunit.net>). Сам ASP.NET Core применяет xUnit в качестве фреймворка для тестирования, поэтому он

стал чем-то вроде соглашения. Если вы знакомы с другим фреймворком, то можете смело использовать его.

Visual Studio включает в себя шаблон для создания тестового проекта .NET Core с xUnit, как показано на рис. 35.1. Выберите **File > New Project** (Файл > Новый проект) и **xUnit Test Project (.NET Core)** (Тестовый проект для xUnit (.NET Core)) в диалоговом окне «Новый проект». В качестве альтернативы можно выбрать **Unit Test Project (.NET Core)** (Проект модульного тестирования (.NET Core)) или **NUnit Test Project (.NET Core)**, если вам удобнее работать с этими фреймворками.

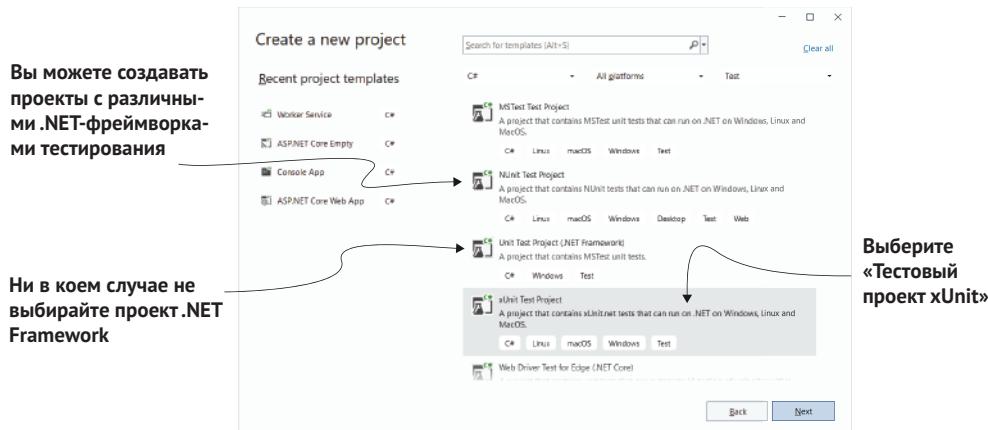


Рис. 35.1 Диалоговое окно «Новый проект» в Visual Studio. Выберите «Тестовый проект для xUnit», чтобы создать проект xUnit, или выберите «Проект модульного тестирования», чтобы создать проект для MSTest

Кроме того, если вы не используете Visual Studio, то можете создать аналогичный шаблон с помощью интерфейса командной строки .NET:

```
dotnet new xunit
```

Независимо от того, используете вы Visual Studio или интерфейс командной строки .NET, шаблон создает консольный проект и добавляет необходимые тестовые пакеты NuGet в файл с расширением .csproj, как показано в следующем листинге. Если вы решили создать тестовый проект для MSTest (или другого фреймворка), пакеты xUnit и адаптер xUnit будут заменены пакетами, соответствующими выбранному вами фреймворку.

Листинг 35.1 Файл с расширением .csproj для тестового проекта с xUnit

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <IsPackable>false</IsPackable>
  </PropertyGroup>
  <ItemGroup>
```

Тестовый проект представляет собой стандартный проект .NET Core, ориентированный на .NET 7.0

Тестовый
адаптер
xUnit для
.NET Test
SDK

```
<PackageReference
    Include="Microsoft.NET.Test.Sdk" Version="17.3.2" />
<PackageReference Include="xunit" Version="2.4.2" />
<PackageReference
    Include="xunit.runner.visualstudio" Version="2.4.5" />
<PackageReference Include="coverlet.collector" Version="3.1.2" />
/>/ItemGroup>
</Project>
```

.NET Test SDK, необходимый
для всех тестовых проектов

Фреймворк тести-
рования xUnit

Необязательный пакет, который собирает метрики
о том, какая часть вашего кода покрыта тестами

СОВЕТ При добавлении пакета Microsoft.NET.Test.Sdk проект помечается как тестовый путем установки MsBuild свойства IsTestProject.

Помимо пакетов NuGet, шаблон включает в себя один пример модульного теста. Он ничего не делает, но тем не менее это действительный тест xUnit, как показано в следующем листинге. В xUnit тест – это метод открытого класса, декорированный атрибутом [Fact].

Листинг 35.2 Пример модульного теста с xUnit, созданного с помощью шаблона по умолчанию

▶ public class UnitTest1
{
 [Fact]
 public void Test1()
 {
 }

Тесты
xUnit
должны
находить-
ся в от-
крытих
классах

Атрибут [Fact] указывает
на то, что метод является
тестовым

Метод, отмеченный атрибутом [Fact], должен
быть открытым и не иметь параметров

Несмотря на то что этот тест ничего не проверяет, он выделяет некоторые характеристики тестов для xUnit:

- тесты обозначаются атрибутом [Fact];
- метод должен быть открытым, без аргументов метода;
- метод ничего не возвращает. Это также может быть асинхронный метод, который возвращает Task;
- метод находится внутри открытого нестатического класса.

ПРИМЕЧАНИЕ Атрибут [Fact] и эти ограничения относятся к фреймворку тестирования xUnit. Другие фреймворки будут использовать иные способы обозначения тестовых классов и будут иметь иные ограничения на сами классы и методы.

Также стоит отметить, что хотя я и сказал, что тестовые проекты являются консольными приложениями, здесь нет класса Program или метода static void Main. Приложение здесь больше похоже на библиотеку классов. Это связано с тем, что SDK для тестирования автоматически внедряет класс Program во время сборки. В целом вам не о чем беспокоиться, но у вас могут возникнуть проблемы, если вы попытаетесь добавить собственный файл Program.cs в свой тестовый проект.

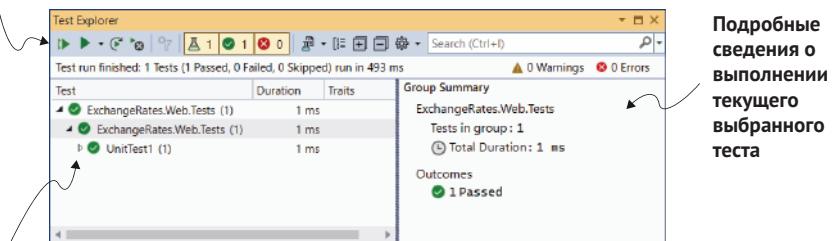
ПРИМЕЧАНИЕ Такое бывает нечасто, но мне иногда приходилось встречаться с подобными случаями. Я подробно описываю эту проблему и способы ее устранения в посте своего блога «Устранение ошибки “У программы имеется несколько точек входа” для консольных приложений, содержащих тесты xUnit»: <http://mng.bz/w9q5>.

Прежде чем мы продолжим и создадим несколько полезных тестов, мы запустим тестовый проект как есть, используя Visual Studio и .NET SDK, чтобы увидеть ожидаемый результат.

35.3 Запуск тестов командой `dotnet test`

Когда вы создаете тестовое приложение, использующее .NET Test SDK, то можете запускать тесты либо с помощью Visual Studio, либо с помощью интерфейса командной строки .NET. В Visual Studio вы запускаете тесты, выбрав **Tests > Run All Tests** (Тесты > Выполнить все тесты) в главном меню или щелкнув по кнопке **Run All** (Выполнить все) в окне обозревателя тестов, как показано на рис. 35.2.

Нажмите кнопку «Запустить все», чтобы запустить все тесты в решении



Все тесты в решении и их последний статус

Подробные сведения о выполнении текущего выбранного теста

Рис. 35.2 В окне обозревателя тестов в Visual Studio перечислены все тесты, которые есть в решении, и их последний статус: пройден / не пройден. Щелкните по тесту на левой панели, чтобы просмотреть сведения о самом последнем запуске на правой панели

В окне обозревателя тестов перечислены все тесты, которые есть в вашем решении, и результаты каждого теста. В xUnit тест считается пройденным, если он не возбуждает исключение, поэтому `Test1` прошел успешно.

ПРИМЕЧАНИЕ Обозреватель тестов в Visual Studio использует протокол VSTest (с открытым исходным кодом, <https://github.com/microsoft/vstest>) для перечисления и отладки тестов. Он также используется в Visual Studio for Mac и в Visual Studio Code.

Кроме того, можно запускать тесты из командной строки с помощью интерфейса командной строки .NET

`dotnet test`

из папки проекта модульных тестов, как показано на рис. 35.3.

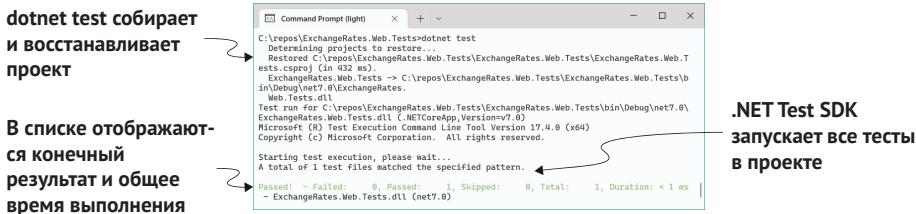


Рис. 35.3 Вы можете запускать тесты из командной строки, используя команду `dotnet test`. Она восстанавливает и собирает тестовый проект перед выполнением всех тестов в проекте

ПРИМЕЧАНИЕ Вы также можете выполнить эту команду из папки «Решения». Она запустит все тестовые проекты, указанные в файле с расширением .sln.

Вызов команды `dotnet test` запускает восстановление и сборку вашего тестового проекта, а затем запускает тесты, как видно из вывода консоли на рис. 35.3. За кулисами интерфейс командной строки .NET вызывает ту же базовую инфраструктуру, что и Visual Studio (.NET SDK), поэтому вы можете использовать способ, который лучше подходит вашему стилю разработки.

Вы видели успешный запуск теста, так что пришло время заменить его чем-нибудь полезным. Но все по порядку: вам нужно что-то протестировать.

35.4 Ссылка на ваше приложение из тестового проекта

При разработке через тестирование (TDD) модульные тесты обычно пишутся до того, как будет написан фактический класс, который вы тестируете, но мы пойдем по более традиционному пути и сначала создадим класс для тестирования, а потом напишем для него тесты.

Предположим, вы создали приложение под названием `ExchangeRates.Web`, которое используется для конвертации разных валют, и хотите добавить для него тесты. Вы добавили тестовый проект для своего решения, как описано в разделе 35.2.1, поэтому ваше решение выглядит так, как показано на рис. 35.4.

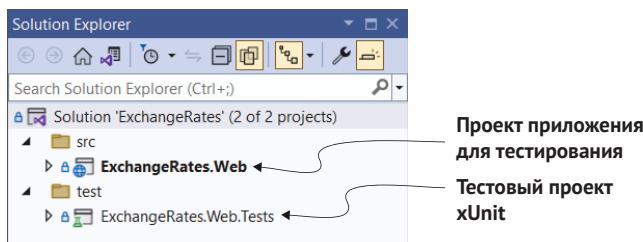


Рис. 35.4 Базовое решение, содержащее приложение ASP.NET Core под названием `ExchangeRates.Web` и тестовый проект `ExchangeRates.Web.Tests`

Чтобы проект ExchangeRates.Web.Tests мог тестировать классы в проекте ExchangeRates.Web, необходимо добавить ссылку на этот веб-проект в свой тестовый проект. В Visual Studio это можно сделать, щелкнув правой кнопкой мыши по узлу тестового проекта «Зависимости» и выбрав «Добавить ссылку», как показано на рис. 35.5. После этого можно выбрать веб-проект в диалоговом окне «Добавить ссылку». После добавления в проект он отображается внутри узла «Зависимости» в разделе «Проекты».

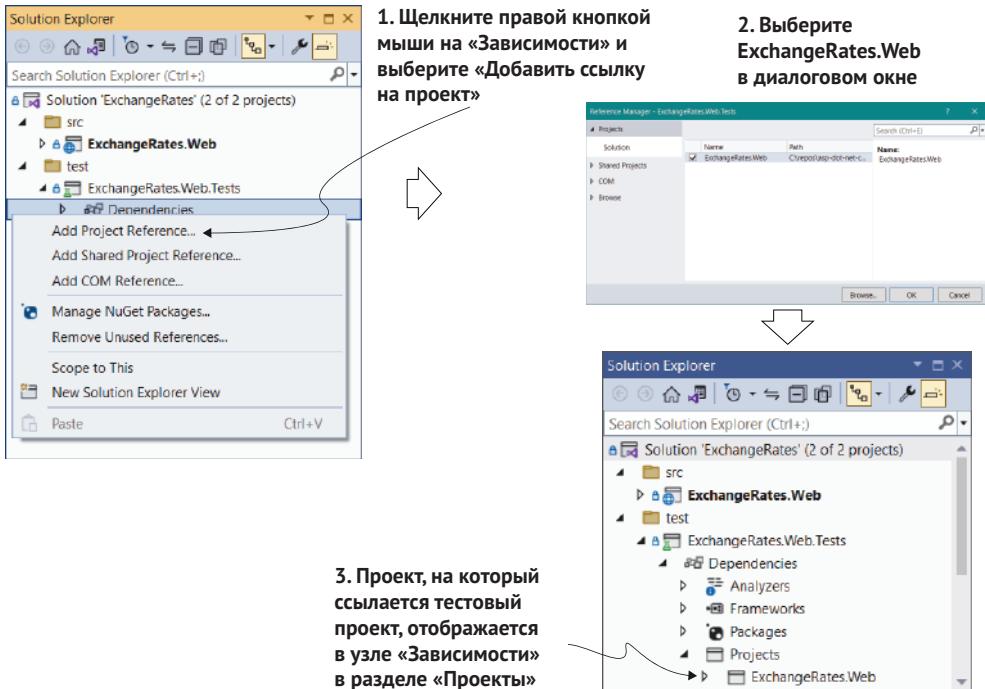


Рис. 35.5 Чтобы протестировать проект приложения, нужно добавить ссылку на него из тестового проекта. Щелкните правой кнопкой мыши узел «Зависимости» и выберите «Добавить ссылку на проект». Ссылка на проект приложения отображается внутри узла «Зависимости» в разделе «Проекты»

Кроме того, вы можете напрямую отредактировать файл с расширением .csproj и добавить элемент <ProjectReference> внутрь элемента <ItemGroup> с относительным путем к файлу проекта .csproj:

```
<ItemGroup>
  <ProjectReference
    Include=".\\src\\ExchangeRates.Web\\ExchangeRates.Web.csproj" />
</ItemGroup>
```

Обратите внимание, что это *относительный* путь. Знак «..» в пути означает родительскую папку, поэтому показанный относительный путь правильно проходит через структуру каталогов решения, включая папки src и test, показанные в обозревателе решений на рис. 35.5.

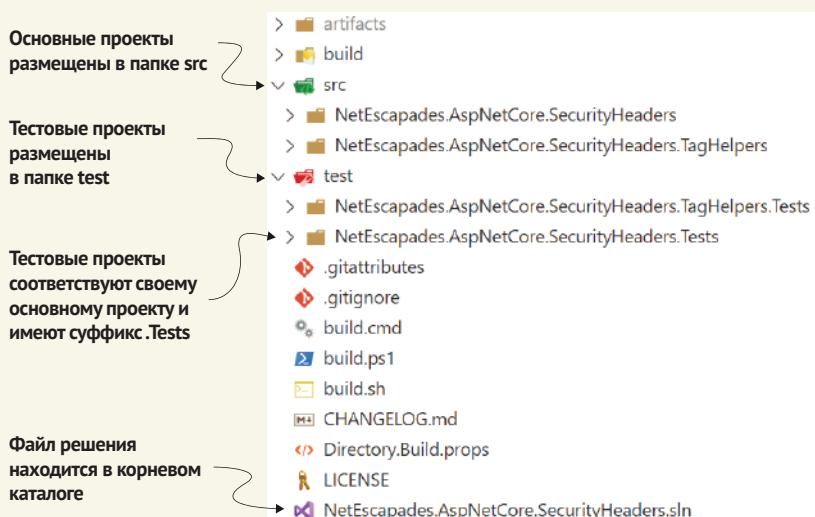
COBET Помните, что вы можете редактировать файл с расширением .csproj прямо в Visual Studio, дважды щелкнув мышью по проекту в обозревателе решений.

Общепринятые соглашения для расположения проекта

Расположение и именование проектов в рамках решения полностью зависят от вас, но в проектах ASP.NET Core обычно используется несколько соглашений, которые немного отличаются от принятых по умолчанию Visual Studio File > New. Эти соглашения применяются командой ASP.NET в GitHub, а также во многих других проектах C# с открытым исходным кодом.

На следующем рисунке показан пример этих соглашений. Вкратце они выглядят так:

- файл решения с расширением .sln находится в корневом каталоге;
- основные проекты помещаются во вложенный каталог src;
- тестовые проекты помещаются во вложенный каталог test или tests;
- у каждого основного проекта есть эквивалент тестового проекта, названный так же, как и связанный с ним основной проект, с суффиксом «.Test» или «.Tests»;
- другие папки, такие как samples, tools или docs, содержат образцы проектов, инструменты для создания проекта или документацию.



Соглашения о структурах проектов появились в библиотеках фреймворка ASP.NET Core и в проектах с открытым исходным кодом на GitHub. Вам не обязательно следовать им в собственном проекте, но о них стоит знать

Будете вы следовать этим соглашениям или нет, полностью зависит от вас, но по крайней мере о них полезно будет знать, чтобы вы могли легко перемещаться по другим проектам на GitHub.

Теперь ваш тестовый проект ссылается на ваш веб-проект, поэтому вы можете писать тесты для классов в веб-проекте. Мы будем тестиировать простой класс, используемый для конвертации валют, как показано в следующем листинге.

Листинг 35.3 Пример класса *CurrencyConverter* для конвертации валют в фунты стерлингов

```
public class CurrencyConverter
{
    public decimal ConvertToGbp(
        decimal value, decimal exchangeRate, int decimalPlaces)
    {
        if (exchangeRate <= 0)
        {
            throw new ArgumentException(
                "Exchange rate must be greater than zero",
                nameof(exchangeRate));
        }
        var valueInGbp = value / exchangeRate; ←
        return decimal.Round(valueInGbp, decimalPlaces); ←
    }
}
```

Используется проверка, поскольку действительны только положительные курсы обмена

Метод *ConvertToGbp* преобразует значение, используя указанный обменный курс, и округляет его

Преобразует значение

Округляет результат и возвращает его

У этого класса есть только один метод, *ConvertToGbp()*, который преобразует *value* из одной валюты в фунты стерлингов с учетом предоставленного параметра *exchangeRate*. Затем он округляет значение до необходимого количества десятичных знаков и возвращает его.

ВНИМАНИЕ! Данный класс – лишь базовая реализация. На практике вам нужно будет обрабатывать арифметические переполнения / потери значимости для больших или отрицательных значений, а также учитывать другие крайние случаи. Здесь этот код приводится в сугубо демонстрационных целях!

Представьте, что вы хотите конвертировать 5,27 доллара США в фунт стерлингов, а обменный курс составляет 1,31 доллара за 1 фунт. Если вы хотите округлить до четырех знаков после запятой, то должны выполнить следующий вызов:

```
converter.ConvertToGbp(value: 5.27, exchangeRate: 1.31, decimalPlaces: 4);
```

У вас есть образец приложения, класс для тестируирования и тестовый проект, так что пора писать тесты.

35.5 Добавление модульных тестов с атрибутами *Fact* и *Theory*

Когда я пишу модульные тесты, то обычно выбираю один из трех путей:

- **успешный путь** – когда приводятся типичные аргументы с ожидаемыми значениями;

- *ошибочный путь* – когда переданные аргументы недействительны и тестируются;
- *граничные случаи* – когда предоставленные аргументы находятся на грани ожидаемых значений.

Я понимаю, что это обширная классификация, но она помогает мне представить различные сценарии, которые мне нужно рассмотреть.

СОВЕТ Совершенно иной подход к тестированию – это тестирование на основе свойств. Этот увлекательный подход распространён в сообществах функционального программирования, таких как F#. Вы можете найти отличное введение Скотта Влашина в его статье «Введение в тестирование на основе свойств»: <http://mng.bz/e5j9>. В этой статье используется язык F#, но тем не менее здесь все очень доходчиво изложено, даже если вы новичок в этом языке.

Начнем с успешного пути, написав модульный тест, который проверяет, что метод `ConvertToGbp()` работает должным образом с типичными входными значениями.

Листинг 35.4 Модульный тест для метода ConvertToGbp с использованием ожидаемых аргументов

```
Вы можете называть тест как хотите
Ожидаемый результат
[Fact]
public void ConvertToGbp_ConvertsCorrectly()
{
    var converter = new CurrencyConverter();
    decimal value = 3;
    decimal rate = 1.5m;
    int dp = 4;
    decimal expected = 2;

    var actual = converter.ConvertToGbp(value, rate, dp);

    Assert.Equal(expected, actual);
}
```

Атрибут [Fact] помечает метод как тестовый

Параметры теста, которые будут переданы в `ConvertToGbp`

Тестируемый класс, обычно называемый «тестируемой системой»

Выполняет метод и фиксирует результат

Проверяет соответствие ожидаемых и фактических значений. Если они не совпадают, возбуждается исключение

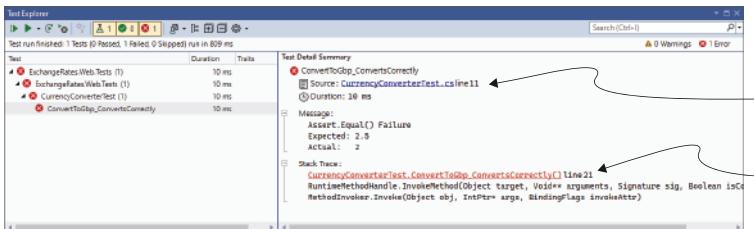
Это ваш первый правильный модульный тест, который был настроен с помощью паттерна Arrange-Act-Assert (AAA):

- *Arrange* – определить все параметры и создать экземпляр тестируемой системы (класса);
- *Act* – выполнить тестируемый метод и зафиксировать результат;
- *Assert* – убедиться, что результат этапа Act имеет ожидаемое значение.

Большая часть кода в этом teste – стандартный C#, но если вы новичок в тестировании, то вызов `Assert` будет вам незнаком. Это вспомогательный класс, предоставляемый xUnit для установки утверждений в отношении вашего кода. Если параметры, предоставленные `Assert.Equal()`, не равны, вызов `Equal()` возбудит исключение, и тест завершится неудачно. Если вы измените переменную `expected` в листинге 35.4 на

2,5 вместо 2, например, и запустите тест, то увидите, что обозреватель тестов показывает сбой, как видно на рис. 35.6.

СОВЕТ Альтернативные библиотеки утверждений, такие как Fluent Assertions (<https://fluentassertions.com/>) и Shouldly (<https://github.com/shouldly/shouldly>), позволяют писать утверждения в более естественном стиле, например `actual.Should().Be(expected)`. Эти библиотеки не являются обязательными, но я считаю, что они делают тесты более читабельными, а сообщения об ошибках проще понять.



Неудачные тесты
помечены красным
крестиком

На панели сведений
указано, почему тест
не удался

Рис. 35.6 Если тест завершается неудачно, он помечается красным крестиком в обозревателе тестов. Если щелкнуть по тесту на левой панели, на правой панели отобразится причина сбоя. В этом случае ожидаемое значение равнялось 2,5, но фактическое значение было равно 2

В листинге 35.4 вы выбрали определенные значения для `value`, `exchangeRate` и `decimalPlaces`, чтобы протестировать успешный путь. Но это только один набор значений из бесконечного числа возможностей, поэтому вам, вероятно, следует протестировать хотя бы *несколько* различных комбинаций. Один из способов сделать это – скопировать и вставить тест несколько раз, настроить параметры и изменить имя метода тестирования, чтобы сделать его уникальным. xUnit предоставляет альтернативный способ добиться того же, без большого количества дублирования кода.

ПРИМЕЧАНИЕ Имена вашего тестового класса и метода используются во фреймворке тестирования для описания теста. Вы можете настроить их отображение в Visual Studio и в интерфейсе командной строки, настроив файл `xunit.runner.json`, как описано здесь: <https://xunit.net/docs/configuration-files>.

Вместо создания метода тестирования с атрибутом `[Fact]` можно создать метод тестирования с атрибутом `[Theory]`. Он предоставляет способ параметризации методов тестирования, позволяя брать метод тестирования и запускать его несколько раз с разными аргументами. Каждый набор аргументов считается отдельным тестом.

Можно переписать тест с атрибутом `[Fact]` из листинга 35.4, чтобы он стал тестом с атрибутом `[Theory]`, как показано ниже. Вместо того чтобы указывать переменные в теле метода, передайте их методу в качестве параметров, а затем декорируйте метод тремя атрибутами

[InlineData]. Каждый экземпляр атрибута предоставляет параметры для одного запуска теста.

Листинг 35.5 Тест для метода ConvertToGbp с атрибутом [Theory], тестирующий несколько наборов значений

```
[Theory] ← Помечает метод как параметризованный тест
[InlineData(0, 3, 0)] ← Каждый атрибут [InlineData] пред-
[InlineData(3, 1.5, 2)] → ставляет все параметры для одного
[InlineData(3.75, 2.5, 1.5)] → запуска метода тестирования

public void ConvertToGbp_ConvertsCorrectly (
    decimal value, decimal rate, decimal expected) ← Метод принимает
    {                                         → параметры, пред-
        var converter = new CurrencyConverter(); → ставляемые атри-
        int dps = 4;                            ← бутами [InlineData]
    }                                         Сравнение результатов → Переменная dps
                                            → не изменяется, по-
                                            → этому нет необ-
                                            → додимости добавлять
                                            → её в [InlineData]
```

Запуск тес-тируемой системы

Если вы запустите этот тест с помощью команды `dotnet test` или Visual Studio, он будет отображаться как три отдельных теста, по одному для каждого набора [InlineData], как показано на рис. 35.7.

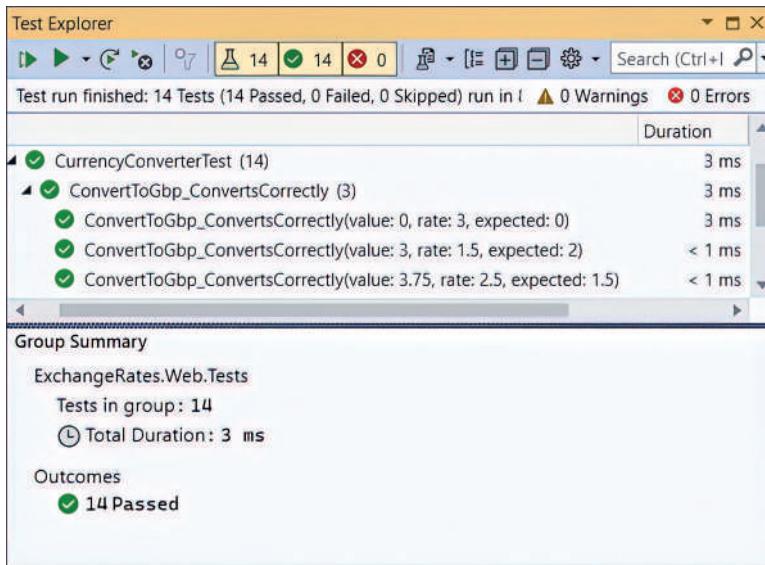


Рис. 35.7 Каждый набор параметров в атрибуте [InlineData] для теста с атрибутом [Theory] создает отдельный запуск теста. В этом примере у одного такого теста три атрибута [InlineData], поэтому он создает три теста, которые названы в соответствии с названием метода и предоставленными параметрами

[InlineData] – не единственный способ указать параметры для подобного рода тестов, но один из наиболее часто используемых.

Вы также можете использовать статическое свойство своего тестового класса с атрибутом `[MemberData]` или сам класс с помощью атрибута `[ClassData]`.

СОВЕТ Я описываю, как можно использовать атрибуты `[ClassData]` и `[MemberData]`, в статье «Создание параметризованных тестов в xUnit с помощью атрибутов `[InlineData]`, `[ClassData]` и `[MemberData]`»: <http://mng.bz/8ayP>.

Теперь у вас есть несколько тестов для успешного пути метода `ConvertToGbp()`, и я даже внедрил граничный случай в листинг 35.5, тестируя ситуацию, когда `value = 0`. Последняя концепция, о которой я расскажу, – это тестирование ошибочных случаев, когда недопустимые значения передаются в тестируемый метод.

35.6 Тестирование условий сбоя

Ключевой частью модульного тестирования является проверка того, что тестируемая система правильно обрабатывает граничные случаи и ошибки. В случае с классом `CurrencyConverter` это означало бы проверку того, как класс обрабатывает отрицательные значения, малые или нулевые обменные курсы, большие значения и курсы и т. д.

Некоторые из этих случаев могут быть редкими, но допустимыми, тогда как другие могут быть технически недопустимыми. Вызов метода `ConvertToGbp` с отрицательным значением, вероятно, допустим; преобразованный результат тоже должен быть отрицательным. Отрицательный обменный курс концептуально не имеет смысла, поэтому его следует рассматривать как недопустимое значение.

В зависимости от конструкции метода, когда в метод передаются недопустимые значения, обычно возбуждаются исключения. В листинге 35.3 было показано, что мы генерируем исключение `ArgumentException`, если параметр `exchangeRate` меньше или равен 0.

xUnit включает в себя множество вспомогательных методов в классе `Assert`, чтобы проверить, возбуждает ли метод исключение ожидаемого типа. Затем вы можете установить дополнительные утверждения в отношении исключения; например, чтобы проверить, есть ли у исключения ожидаемое сообщение.

ВНИМАНИЕ Страйтесь не связывать свои методы тестирования слишком тесно с внутренней реализацией метода. Это может сделать тесты хрупкими, когда тривиальные изменения класса будут нарушать модульные тесты.

В следующем листинге показан тест с атрибутом `[Fact]` для проверки поведения метода `ConvertToGbp()`, когда вы передаете ему параметр `exchangeRate`, равный 0. Метод `Assert.Throws` принимает лямбда-функцию, описывающую выполняемое действие, которое должно возбудить исключение при запуске.

Листинг 35.6 Использование Assert.Throws<>, чтобы проверить, генерирует ли метод исключение

```
[Fact]
public void ThrowsExceptionIfRateIsZero()
{
    var converter = new CurrencyConverter();
    const decimal value = 1;
    const decimal rate = 0;
    const int dp = 2;
    var ex = Assert.Throws<ArgumentOutOfRangeException>(
        () => converter.ConvertToGbp(value, rate, dp));
    // Дальнейшие утверждения относительно сгенерированного исключения, ex;
}
```

Недействительное значение

Вы ожидаете, что будет возбуждено исключение ArgumentOutOfRangeException

Метод для выполнения, который должен возбудить исключение

Метод `Assert.Throws` выполняет лямбда-функцию и перехватывает исключение. Если сгенерированное исключение соответствует ожидаемому типу, тест будет пройден. Если исключение не генерируется или генерируется исключение не того типа, который вы ожидали, метод `Assert.Throws` сгенерирует собственное исключение, и тест завершится неудачно.

На этом мы подошли к концу введения в модульное тестирование с помощью xUnit. Примеры в этом разделе описывают, как использовать новый .NET Test SDK, но мы не коснулись ничего, относящегося к ASP.NET Core. В главе 36 мы сосредоточимся конкретно на тестировании проектов ASP.NET Core.

Резюме

- Приложения модульного тестирования – это консольные приложения с зависимостью от .NET Test SDK, фреймворка для тестирования, такого как xUnit, MSTest или NUnit, и адаптера выполнения тестов. Вы можете запускать тесты в тестовом проекте, вызвав команду `dotnet test` из командной строки в тестовом проекте или с помощью обозревателя тестов в Visual Studio;
- многие фреймворки для тестирования совместимы с .NET Test SDK, но xUnit стал практически стандартом *де-факто* для проектов ASP.NET Core. Сама команда ASP.NET Core использует его для тестирования фреймворка;
- чтобы создать тестовый проект xUnit, выберите **xUnit Test Project (.NET Core)** в Visual Studio или используйте команду интерфейса командной строки `dotnet new xunit`. Так вы создаете тестовый проект, содержащий пакеты Microsoft.NET.Test.Sdk, xunit и xunit.runner.visualstudio;
- xUnit включает в себя два разных атрибута для идентификации методов тестирования. Методы с атрибутом `[Fact]` должны быть открытыми и не иметь параметров. Методы с атрибутом `[Theory]` могут содержать параметры, поэтому их можно использовать для

многократного выполнения аналогичного теста с разными параметрами. Вы можете предоставить данные для каждого запуска метода с атрибутом [Theory], используя атрибуты [InlineData], [ClassData] или [MemberData];

- используйте утверждения в своих методах тестирования, чтобы убедиться, что тестируемая система (SUT) вернула ожидаемое значение. Утверждения существуют для наиболее распространенных сценариев, включая проверку того, что вызов метода вызвал исключение определенного типа. Если ваш код возбуждает необработанное исключение, тест завершится неудачно.

36

Тестирование приложений ASP.NET Core

В этой главе:

- написание модульных тестов для пользовательского промежуточного программного обеспечения, контроллеров API и конечных точек минимальных API;
- использование пакета Test Host для написания интеграционных тестов;
- тестирование поведения вашего реального приложения с помощью WebApplicationFactory;
- тестирование кода, зависящего от Entity Framework Core, с помощью поставщика базы данных в памяти.

В главе 35 я описал, как тестировать приложения .NET 7 с помощью тестового проекта xUnit и комплекта разработки программного обеспечения (SDK) .NET Test. Вы узнали, как создать тестовый проект, добавить ссылку на проект в свое приложение и написать модульные тесты для сервисов в своем приложении.

В этой главе мы сосредоточимся конкретно на тестировании приложений ASP.NET Core. В разделах 36.1 и 36.2 мы рассмотрим, как тестировать общие функции ваших приложений ASP.NET Core: пользовательское промежуточное ПО, контроллеры API и конечные точки минимальных API. Я покажу вам, как писать изолированные модульные тесты, и укажу на моменты, на которые стоит обратить внимание.

Чтобы убедиться, что компоненты работают правильно, важно тестировать их изолированно. Но вам также необходимо проверить, правильно ли они работают в конвейере промежуточного программного обеспечения. ASP.NET Core предоставляет удобный пакет Test Host, который позволяет легко писать интеграционные тесты для ваших компонентов. Вы даже можете пойти еще дальше с помощью вспомогательного класса WebApplicationFactory и проверить правильность работы вашего приложения. В разделе 36.3 вы увидите, как использовать WebApplicationFactory для имитации запросов к вашему приложению и проверки того, что оно генерирует правильный ответ.

В последнем разделе этой главы я покажу, как использовать поставщика базы данных SQLite для Entity Framework Core (EF Core) с базой данных в памяти приложения. Этого поставщика можно использовать для тестирования служб, которые зависят от DbContext из EF Core, без необходимости использования реальной базы данных. Это избавляет от боли, связанной с неизвестной инфраструктурой базы данных и сбросом базы данных между тестами, когда разные люди имеют различные конфигурации базы данных.

В главе 35 я показал, как писать модульные тесты для службы калькулятора обменных курсов, например, которая может быть в модели предметной области вашего приложения. Если сервисы предметной области хорошо спроектированы, их обычно сравнительно легко протестировать. Но они составляют лишь часть вашего приложения. Также может быть полезно протестировать конструкции, специфичные для ASP.NET Core, например пользовательское промежуточное ПО, как вы увидите в следующем разделе.

36.1 Модульное тестирование пользовательского промежуточного ПО

В этом разделе вы узнаете, как тестировать пользовательское промежуточное ПО изолированно. Вы увидите, как проверить, обработало ли ваше промежуточное ПО запрос или же вызвало следующий компонент в конвейере, а также увидите, как читать поток ответов.

В главе 31 вы видели, как создать собственное промежуточное ПО и как инкапсулировать его в виде класса с функцией `Invoke`. В этом разделе вы создадите модульные тесты для проверки работоспособности простого компонента промежуточного ПО, аналогичного тому, что описан в главе 31. Это базовая реализация, но она демонстрирует подход, который можно использовать для более сложных компонентов.

Промежуточное ПО, которое вы будете тестировать, показано в листинге 36.1. При вызове оно проверяет, начинается ли путь с `/ping`, и если да, то возвращает ответ в виде простого текста "pong". Если запрос не совпадает, вызывается следующий компонент в конвейере (предоставленный `RequestDelegate`).

Листинг 36.1 Тестируемый класс StatusMiddleware, возвращающий ответ "pong"

```

public class StatusMiddleware
{
    private readonly RequestDelegate _next;
    public StatusMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    public async Task Invoke(HttpContext context)
    {
        if(context.Request.Path.StartsWithSegments("/ping"))
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("pong");
            return;
        }
        await _next(context);
    }
}

```

RequestDelegate, представляющий остальную часть конвейера промежуточного ПО

Вызывается при выполнении промежуточного ПО

Если путь начинается с "/ping", возвращается ответ "pong"

В противном случае вызывается следующий компонент промежуточного ПО в конвейере

В этом разделе мы протестируем только два простых случая:

- когда запрос выполнен с использованием пути "/ping";
- когда запрос выполнен с использованием другого пути.

ВНИМАНИЕ! По возможности я рекомендую *не* проверять пути в промежуточном ПО таким способом. Лучше использовать маршрутизацию конечных точек, как обсуждалось в главе 31. Промежуточное ПО в этом разделе предназначено лишь для демонстрационных целей.

Промежуточное ПО сложно поддается модульному тестированию, поскольку концептуально объект `HttpContext` представляет собой *большой* класс. Он содержит все сведения о запросе и ответе, а это может означать, что у вашего промежуточного ПО есть много возможностей для взаимодействия. Поэтому я считаю, что модульные тесты в основном тесно связаны с реализацией промежуточного ПО, что, как правило, нежелательно.

Для первого теста мы рассмотрим случай, когда путь входящего запроса не начинается с `/ping`. В этом случае класс `StatusMiddleware` должен оставить `HttpContext` без изменений и вызвать `RequestDelegate`, предоставленный в конструкторе, который представляет следующий компонент в конвейере.

Это поведение можно протестировать несколькими способами, но в листинге 36.2 вы проверяете, что `RequestDelegate` (по сути, функция с одним параметром) выполняется путем установки для локальной переменной значения `true`. В `Assert` в конце метода вы проверяете, что переменная была установлена и, следовательно, был вызван делегат. Чтобы вызвать `StatusMiddleware`, создайте и передайте `DefaultHttpContext`, который является реализацией `HttpContext`.

ПРИМЕЧАНИЕ DefaultHttpContext наследует от класса HttpContext и является частью базовой абстракции фреймворка ASP.NET Core. Если хотите, то можете изучить его исходный код на сайте GitHub: <http://mng.bz/q9qx>.

Листинг 36.2 Модульное тестирование класса StatusMiddleware, когда указан несовпадающий путь

```
[Fact]
public async Task ForNonMatchingRequest_CallsNextDelegate()
{
    var context = new DefaultHttpContext();
    context.Request.Path = "/somethingelse";
    var wasExecuted = false;
    RequestDelegate next = (HttpContext ctx) =>
    {
        wasExecuted = true;
        return Task.CompletedTask;
    };
    var middleware = new StatusMiddleware(next); <-- Создает экземпляр промежуточного ПО, передавая следующий RequestDelegate
    await middleware.Invoke(context); <-- Вызывает промежуточное ПО с HttpContext; должен вызывать RequestDelegate
    Assert.True(wasExecuted);
}
```

Отслеживает, был ли выполнен Request-Delegate

Проверяет, был ли вызван Request-Delegate

Создает DefaultHttpContext и задает путь для запроса

Создает экземпляр промежуточного ПО, передавая следующий RequestDelegate

В этом примере должен быть вызван RequestDelegate, представляющий следующий компонент промежуточного ПО

Когда промежуточное ПО вызывается, оно проверяет предоставленный путь и обнаруживает, что он не соответствует требуемому значению /ping. Поэтому вызывается следующий RequestDelegate и возвращается ответ.

Другой очевидный случай, который нужно протестировать, – когда путь запроса – это "/pong"; промежуточное ПО должно сгенерировать соответствующий ответ. Вы можете протестировать несколько характеристик ответа:

- в ответе должен быть код состояния 200 OK;
- ответ должен иметь заголовок Content-Type:text/plain;
- тело ответа должно содержать строку "pong".

Каждая из этих характеристик представляет собой отдельное требование, поэтому обычно каждая из них выделяется в отдельный модульный тест. Так проще определить, какое именно требование не было выполнено, если тест завершился неудачно. Чтобы было проще, в листинге 36.3 я показываю все эти утверждения в одном и том же teste.

Положительный модульный тест усложняется тем, что необходимо прочитать тело ответа, чтобы убедиться, что оно содержит слово "pong". DefaultHttpContext использует Stream.Null для объекта Response.Body, а это означает, что все, что пишется в Body, теряется. Чтобы записать ответ и прочитать его для проверки содержимого, вы должны заменить Body на MemoryStream. После выполнения промежуточного ПО вы можете использовать StreamReader для чтения содержимого MemoryStream в строку и проверить его.

Листинг 36.3 Модульное тестирование StatusMiddleware

при указании совпадающего пути

```
[Fact]
public async Task ReturnsPongBodyContent()
{
    var bodyStream = new MemoryStream();
    var context = new DefaultHttpContext();
    context.Response.Body = bodyStream;
    context.Request.Path = "/ping"; <-- Создает DefaultHttpContext и инициализирует тело с помощью MemoryStream
    RequestDelegate next = (ctx) => Task.CompletedTask; <-- Для пути задается значение, необходимое для StatusMiddleware
    var middleware = new StatusMiddleware(next: next); <-- Создает экземпляр промежуточного ПО и передает простой RequestDelegate
    middleware.Invoke(context); <-- Вызывает промежуточное ПО
    string response;
    bodyStream.Seek(0, SeekOrigin.Begin);
    using (var stringReader = new StreamReader(bodyStream))
    {
        response = await stringReader.ReadToEndAsync(); <-- Проверяет правильность значения ответа
    }
    Assert.Equal("pong", response); <-- Проверяет правильность Content-Type ответа
    Assert.Equal("text/plain", context.Response.ContentType);
    Assert.Equal(200, context.Response.StatusCode); <-- Проверяет правильность Content-Type ответа
}
} <-- Проверяет правильность кода состояния ответа
```

Перематывает MemoryStream в начало и считывает тело ответа в строку

Как видите, промежуточное ПО для модульного тестирования требует большого количества настроек, чтобы заставить его работать, но есть и положительная сторона: это позволяет тестировать промежуточное ПО изолированно. Но в некоторых случаях, особенно для простых компонентов без каких-либо зависимостей от баз данных или других служб, интеграционное тестирование может быть (что несильно удивительно) проще. В разделе 36.3 вы создадите интеграционные тесты для таких компонентов, чтобы увидеть разницу.

Пользовательское промежуточное ПО – обычное явление в проектах ASP.NET Core, но гораздо более распространены Razor Pages, контроллеры API и конечные точки минимальных API. В следующем разделе вы увидите, как тестировать их отдельно от других компонентов.

36.2 Модульное тестирование контроллеров API и конечных точек минимальных API

В этом разделе вы увидите, как проводить модульное тестирование контроллеров API, узнаете о преимуществах и трудностях изолированного тестирования этих компонентов, а также о ситуациях, когда это может быть полезно.

Модульные тесты изолируют поведение; вам нужно протестировать только логику, содержащуюся в самом компоненте, отдельно от

поведения каких-либо зависимостей. Фреймворки Razor Pages и MVC/API используют конвейер фильтров, маршрутизацию и привязку к модели, но все они являются *внешними* по отношению к контроллеру или PageModels. PageModels и сами контроллеры несут ответственность только за ограниченное количество функций:

- в случае с недействительными запросами (например, не прошедшиими валидацию) они возвращают соответствующий ActionResult (контроллеры API) или повторно отображают форму (Razor Pages);
- в случае с действительными запросами вызывают необходимые сервисы бизнес-логики и возвращают соответствующий ActionResult (контроллеры API) или показывают либо перенаправляют на страницу с сообщением об успешном результате (Razor Pages);
- при необходимости применяют авторизацию на основе ресурсов.

Контроллеры и Razor Pages сами по себе не должны содержать бизнес-логику; они должны обращаться к другим сервисам. Рассматривайте их скорее как оркестраторы, выступающие в качестве посредника между HTTP-интерфейсами, которые предоставляет ваше приложение, и службами бизнес-логики.

Если вы будете следовать этому разделению, вам будет проще писать модульные тесты для своей бизнес-логики, и вы получите больше гибкости, когда захотите изменить свои контроллеры в соответствии со своими потребностями. Учитывая это, часто возникает стремление сделать свои контроллеры и обработчики страниц как можно более легковесными до такой степени, что тестировать практически уже нечего!

СОВЕТ Одним из первых моих знакомств с этой идеей была серия сообщений Джимми Богарда. Следующая ссылка указывает на последнюю статью в этой серии, а также содержит ссылки на все предыдущие статьи. Джимми Богард также является автором библиотеки MediatR (<https://github.com/jbogard/MediatR>), которая делает создание легковесных контроллеров еще проще. См. «Посадите свои контроллеры на диету: POST и команды»: <http://mng.bz/7VNQ>.

При этом контроллеры и действия – это классы и методы, поэтому вы можете писать для них модульные тесты. Сложность состоит в том, чтобы решить, что вы хотите протестировать. В качестве примера мы рассмотрим простой контроллер API из следующего листинга, который преобразует значение с использованием заданного обменного курса и возвращает ответ.

Листинг 36.4 Тестируемый контроллер API

```
[Route("api/[controller]")]
public class CurrencyController : ControllerBase
{
    private readonly CurrencyConverter _converter
        = new CurrencyConverter();
```

CurrencyConverter обычно внедряется с использованием внедрения зависимостей. Здесь он создан таким образом для простоты

```
[HttpGet]
public ActionResult<decimal> Convert(InputModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    decimal result = _converter.ConvertToGbp(model)
    return result;
}
```

Если введенные данные не являются валидными, возвращаем результат 400 Bad Request и ModelState

Метод Convert возвращает ActionResult<T>

Если модель валидна, вычисляем результат

Возвращаем результат напрямую

Для начала рассмотрим успешный путь, когда контроллер получает валидный запрос. В следующем листинге показано, что вы можете создать экземпляр контроллера API, вызвать метод действия и получить ответ ActionResult<T>.

Листинг 36.5 Простой модульный тест контроллера API

```
public class CurrencyControllerTest
{
    [Fact]
    public void Convert_ReturnsValue()
    {
        var controller = new CurrencyController();
        var model = new InputModel
        {
            Value = 1,
            ExchangeRate = 3,
            DecimalPlaces = 2,
        };
        ActionResult<decimal> result = controller.Convert(model); <-- Создает экземпляр CurrencyController для тестирования и модель для отправки в API
        Assert.NotNull(result); <-- Утверждает, что IActionResult не является null
    }
}
```

Вызывает метод ConvertToGbp и фиксирует возвращаемое значение

Тут важно отметить, что вы тестируете *только* возвращаемое значение действия ActionResult<T>, а *не* ответ, который отправляется обратно пользователю. Процесс сериализации результата в ответ обрабатывается инфраструктурой форматирования MVC, как было показано в главе 9, а не контроллером.

При модульном тестировании контроллеров вы тестируете их *отдельно* от инфраструктуры MVC, такой как форматирование, привязка к модели, маршрутизация и аутентификация. Очевидно, что это делается намеренно, но, как и в случае тестирования промежуточного ПО в разделе 36.1, это может несколько усложнить тестирование некоторых аспектов контроллера.

Рассмотрите возможность валидации модели. Как было показано в главе 6, одна из ключевых задач методов действий и обработчиков

страниц Razor – проверять свойство `ModelState.IsValid` и выполнять соответствующие действия, если модель привязки не является валидной. Когда вы проверяете, правильно ли ваши контроллеры и `PageModel` обрабатывают ошибки валидации, это может стать хорошим кандидатом для модульного теста.

К сожалению, и здесь все непросто. Razor Page и MVC автоматически устанавливают свойство `ModelState` как часть процесса привязки модели. На практике, когда ваш метод действия или обработчик страницы вызывается в запущенном приложении, вы знаете, что `ModelState` будет соответствовать значениям модели привязки. Но в модульном teste привязка к модели отсутствует, поэтому вы должны задать свойство вручную.

Представьте, что вы хотите протестировать путь ошибок валидации для контроллера из листинга 36.4, где модель не является валидной и контроллер должен вернуть `BadRequestObjectResult`. В модульном teste нельзя полагаться на правильность свойства `ModelState` для модели привязки. Вместо этого вы должны *вручную* добавить ошибку привязки модели к свойству контроллера `ModelState` перед вызовом действия, как показано здесь.

Листинг 36.6 Проверка обработки ошибок валидации в контроллерах MVC

```
[Fact]
public void Convert_ReturnsBadRequestWhenInvalid()
{
    var controller = new CurrencyController(); ← Создает экземпляр контроллера для тестирования
    var model = new InputModel
    {
        Value = 1,
        ExchangeRate = -2, ← Создает невалидную модель привязки, используя отрицательное значение ExchangeRate
        DecimalPlaces = 2,
    };
    controller.ModelState.AddModelError(
        nameof(model.ExchangeRate),
        "Exchange rate must be greater than zero" ← Вручную добавляет ошибку модели в ModelState контроллера, что задает для ModelState.IsValid значение false
    );
    ActionResult<decimal> result = controller.Convert(model); ← Вызывает метод действия, передавая модель привязки
    Assert.IsType<BadRequestObjectResult>(result.Result);
}
} → Проверяет, что метод действия возвращает BadRequestObjectResult
```

ПРИМЕЧАНИЕ В листинге 36.6 я передал недопустимую модель, но с таким же успехом мог бы передать валидную модель или даже `null`; контроллер не использует модель привязки, если свойство `ModelState` недействительно, поэтому тест все равно будет пройден. Но если вы пишете такие модульные тесты, то рекомендую постараться, чтобы ваша модель соответствовала этому свойству; в противном случае получится, что ваши модульные тесты не тестируют ситуацию, которая возникает на практике.

Лично я стараюсь избегать модульного тестирования контроллеров API таким образом. Как уже было показано на примере привязки модели, контроллеры в некоторой степени зависят от более ранних этапов фреймворка MVC, которые вам часто нужно эмулировать. Точно так же, если ваши контроллеры обращаются к `HttpContext` (доступен в базовых классах `ControllerBase`), может потребоваться дополнительная настройка.

ПРИМЕЧАНИЕ Подробнее о том, почему я обычно не провожу модульное тестирование контроллеров, можно узнать в статье из моего блога «Нужно ли выполнять модульное тестирование контроллеров API/MVC в ASP.NET Core?»: <http://mng.bz/YqMo>.

А как насчет конечных точек минимальных API? Здесь есть как хорошие, так и плохие новости. С одной стороны, конечные точки минимальных API – это простые лямбда-функции, поэтому их можно тестировать, но эти тесты также страдают множеством недостатков:

- вы должны писать обработчики конечных точек как статические методы или методы экземпляра класса, а не как лямбда-методы или локальные функции, чтобы на них можно было ссылаться из тестового проекта;
- вы тестируете только выполнение обработчика конечной точки, вне любых фильтров, примененных к конечной точке или группе маршрутов, которые выполняются в реальном приложении;
- вы не тестируете привязку модели или сериализацию результатов – два распространенных источника ошибок на практике;
- если ваша конечная точка проста, как и должно быть, тестируть особо нечего!

Я считаю, что модульные тесты для минимальных API слишком ограничены и имеют малую ценность, поэтому я избегаю их, но вы можете увидеть пример модульного теста минимального API в исходном коде для этой главы.

ПРИМЕЧАНИЕ В этом разделе я не слишком подробно обсуждал страницы Razor Pages, поскольку по большей части они испытывают те же проблемы, в том смысле, что зависят от поддерживающей инфраструктуры фреймворка. Тем не менее если вы действительно хотите протестировать модель страницы Razor Pages, то можете прочитать об этом в документации Microsoft «Модульные тесты Razor Pages в ASP.NET Core»: <http://mng.bz/GxmM>.

Вместо использования модульного тестирования я стараюсь сделать так, чтобы мои контроллеры и Razor Pages были как можно легковеснее. Я помещаю как можно больше поведения в классы сервисов бизнес-логики, для которых можно легко выполнить модульное тестирование, или в промежуточное ПО и фильтры, которые легче тестировать независимо.

ПРИМЕЧАНИЕ Это личное предпочтение. Некоторым нравится максимально приближаться к 100 % тестового покрытия кода, но я считаю, что тестирование классов оркестрации часто доставляет больше хлопот.

Хотя я часто отказываюсь от модульного тестирования контроллеров и Razor Pages, нередко я пишу *интеграционные* тесты, которые тестируют их в контексте полноценного приложения. В следующем разделе мы рассмотрим способы написания интеграционных тестов для вашего приложения, чтобы вы могли протестировать различные его компоненты в контексте фреймворка ASP.NET Core в целом.

36.3 Интеграционное тестирование: тестирование всего приложения в памяти

В этом разделе вы узнаете, как создавать интеграционные тесты, проверяющие взаимодействие компонентов. Вы научитесь создавать `TestServer`, который отправляет HTTP-запросы в памяти, чтобы упростить тестирование пользовательских компонентов промежуточного ПО. Затем вы узнаете, как запускать интеграционные тесты для реального приложения, используя конфигурацию реального приложения, сервисы и конвейер промежуточного ПО. Наконец, вы узнаете, как использовать `WebApplicationFactory` для замены сервисов в приложении тестовыми версиями, чтобы избежать зависимости от сторонних API в своих тестах.

Если вы поищете в интернете различные типы тестирования, то найдете множество разных типов на выбор. Различие между ними иногда не так ощущимо, и люди не всегда согласны с определениями. Я решил не останавливаться на этом в данной книге – я считаю модульные тесты изолированными тестами определенного компонента, а интеграционные тесты – тестами, которые проверяют несколько компонентов одновременно.

В этом разделе я покажу, как написать интеграционные тесты для класса `StatusMiddleware` из раздела 36.1 и контроллера API из раздела 36.2. Вместо того чтобы изолировать компоненты от окружающего фреймворка и вызывать их напрямую, вы намеренно протестируете их в контексте, аналогично тому, как вы их используете на практике.

Интеграционные тесты – важная часть подтверждения правильности работы ваших компонентов, но они не устраниют необходимость в модульных тестах. Модульные тесты отлично подходят для тестирования небольших логических элементов, содержащихся в ваших компонентах, и обычно выполняются быстро. Интеграционные тесты чаще выполняются значительно медленнее, поскольку требуют гораздо большей конфигурации и могут полагаться на внешнюю инфраструктуру, такую как база данных.

Следовательно, если у вас гораздо больше модульных тестов для приложения, чем интеграционных, то это нормально. Как вы уже видели в главе 35, модульные тесты обычно проверяют поведение компонента, используя допустимые входные данные, граничные случаи и недопустимые входные данные, чтобы гарантировать, что компонент ведет себя правильно во всех случаях. Если у вас есть обширный набор модульных тестов, вам, скорее всего, понадобится пройти несколько интеграционных тестов, чтобы убедиться, что ваше приложение работает правильно.

Вы можете написать множество различных типов интеграционных тестов для приложения и проверить, что сервис может правильно писать данные в базу данных, может интегрироваться со сторонним сервисом (например, для отправки электронных писем) или может обрабатывать HTTP-запросы, которые выполняются к нему.

В этом разделе мы сосредоточимся на последнем пункте, проверив, может ли ваше приложение обрабатывать поступающие к нему запросы, как если бы вы обращались к приложению из браузера. Для этого мы будем использовать библиотеку Microsoft.AspNetCore.TestHost, предоставленную командой ASP.NET Core.

36.3.1 Создание TestServer с помощью пакета Test Host

Представьте, что вы хотите написать несколько интеграционных тестов для класса `StatusMiddleware` из раздела 36.1. Вы уже написали для него модульные тесты, но хотите иметь хотя бы один интеграционный тест, который тестирует промежуточное ПО в контексте инфраструктуры ASP.NET Core.

Можно сделать это по-разному. Возможно, наиболее полным подходом было бы создание отдельного проекта и настройка класса `StatusMiddleware` как единственного компонента в конвейере. Затем нужно будет запустить этот проект, дождаться начала его работы, отправить ему запросы и проверить ответы.

Возможно, это будет хороший тест, но он также потребует большого количества настроек, а кроме того, будет ненадежным и подвержен ошибкам. Что делать, если тестовое приложение не запускается, потому что пытаются использовать уже занятый порт или не завершает работу правильно? Сколько интеграционный тест должен ждать запуска приложения?

Пакет Test Host позволяет приблизиться к желаемой тестовой конфигурации без дополнительных сложностей, связанных с запуском отдельного приложения. Добавить Test Host в свой тестовый проект можно, используя пакет `Microsoft.AspNetCore.TestHost` либо с помощью графического интерфейса NuGet Visual Studio, консоли диспетчера пакетов или интерфейса командной строки .NET. Как вариант добавьте элемент `<PackageReference>` непосредственно в файл своего тестового проекта с расширением `.csproj`:

```
<PackageReference Include="Microsoft.AspNetCore.TestHost" Version="7.0.0" />
```

В типичном приложении ASP.NET Core вы создаете `HostBuilder` в классе `Program`, настраиваете веб-сервер (Kestrel) и определяете конфигурацию своего приложения, сервисы и конвейер промежуточного ПО (с использованием файла `Startup`). Наконец, вы вызываете метод `Build()` класса `HostBuilder` для создания экземпляра `IHost`, который может быть запущен и который будет слушать запросы по определенному URL-адресу и порту.

ПРИМЕЧАНИЕ Все это происходит за кулисами, когда вы используете минимальный хостинг `WebApplicationBuilder` и API `WebApplication`. В моем блоге по адресу <http://mng.bz/a1mj> есть подробный пост, посвященный коду `WebApplicationBuilder` и его связи с `HostBuilder`.

Пакет Test Host использует тот же HostBuilder для определения вашего тестового приложения, но вместо того, чтобы слушать запросы на уровне сети, создает экземпляр IHost, который использует объекты запроса в памяти, как показано на рис. 36.1.

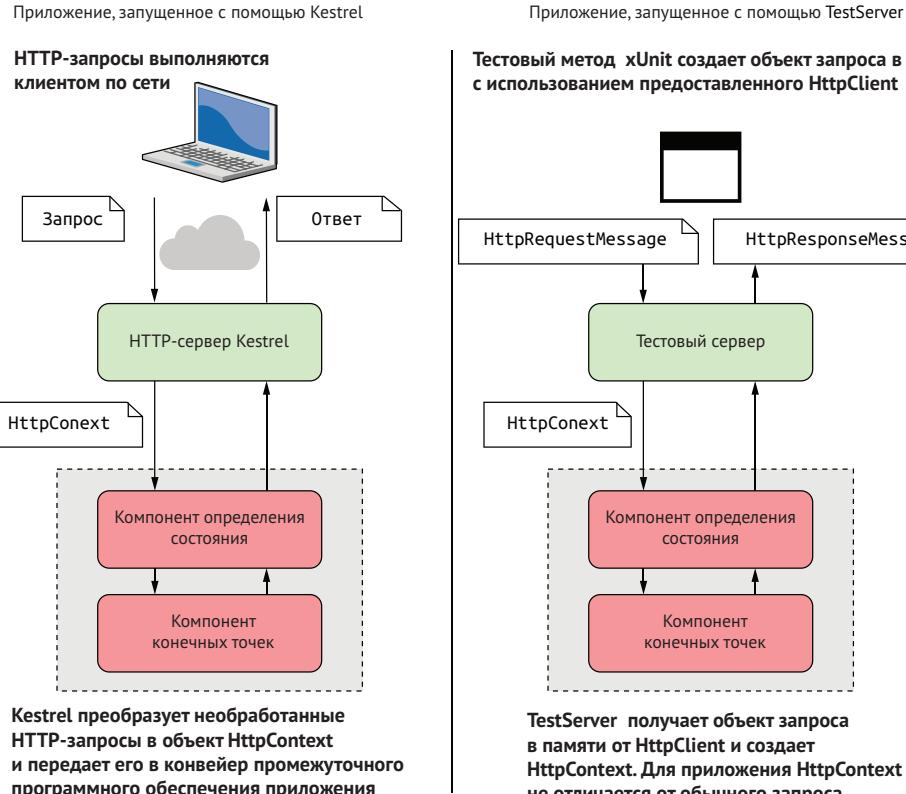


Рис. 36.1 Когда ваше приложение работает в обычном режиме, то использует сервер Kestrel. Он слушает HTTP-запросы и преобразует их в объект HttpContext, который передается в конвейер промежуточного ПО. TestServer не слушает запросы в сети. Вместо этого вы используете HttpClient для выполнения запросов в памяти. С точки зрения промежуточного ПО никакой разницы нет

Он даже предоставляет объект HttpClient, который можно использовать для отправки запросов в тестовое приложение. Вы можете взаимодействовать с HttpClient, как если бы он отправлял запросы по сети, но на самом деле запросы целиком хранятся в памяти.

В листинге 36.7 показано, как использовать пакет Test Host для создания простого интеграционного теста для класса StatusMiddleware. Сначала создайте экземпляр HostBuilder и вызовите метод ConfigureWebHost(), чтобы определить приложение, добавив промежуточное ПО в методе Configure. Это эквивалентно методу Startup.Configure(), который обычно используется для настройки приложения.

ПРИМЕЧАНИЕ Вы можете написать аналогичный тест, используя `WebApplicationBuilder`, но это устанавливает множество дополнительных настроек по умолчанию, таких как конфигурация, дополнительное внедрение зависимостей (DI) и автоматически добавляемое промежуточное программное обеспечение, которое обычно может замедлить работу и внести некоторую путаницу в простые тесты. Вы можете увидеть пример этого подхода в `StatusMiddlewareTestHostTests` в исходном коде данной книги, но я рекомендую использовать подход из листинга 36.7, используя `HostBuilder`, в большинстве случаев.

Вызовите метод расширения `UseTestServer()` в методе `ConfigureWebHost()`, который заменяет сервер Kestrel по умолчанию на `TestServer` из пакета Test Host. `TestServer` – главный компонент пакета Test Host, благодаря которому вся эта магия становится возможной. После настройки `HostBuilder` вызовите метод `StartAsync()`, чтобы построить и запустить тестовое приложение. Затем вы можете создать объект `HttpClient`, используя метод расширения `GetTestClient()`. Он возвращает `HttpClient`, настроенный для выполнения запросов в памяти к `TestServer`.

Листинг 36.7 Создание интеграционного теста с `TestServer`

```
public class StatusMiddlewareTests
{
    [Fact]
    public async Task StatusMiddlewareReturnsPong()
    {
        var hostBuilder = new HostBuilder()
            .ConfigureWebHost(webHost =>
        {
            webHost.Configure(app =>
                app.UseMiddleware<StatusMiddleware>());
            webHost.UseTestServer();
        });
    }

    IHost host = await hostBuilder.StartAsync();
    HttpClient client = host.GetTestClient();

    var response = await client.GetAsync("/ping");
    response.EnsureSuccessStatusCode();
    var content = await response.Content.ReadAsStringAsync();
    Assert.Equal("pong", content);
}
```

Этот тест гарантирует, что тестовое приложение, определенное `HostBuilder`, вернет ожидаемое значение при получении запроса по пути `/ping`. Запрос полностью находится в памяти, но с точки зрения `StatusMiddleware` это то же самое, как если бы запрос пришел из сети.

Конфигурация `HostBuilder` в этом примере проста. Хотя я и назвал этот тест интеграционным, вы тестируете именно класс `StatusMiddleware` сам по себе, а не в контексте «реального» приложения. Я думаю, что такая настройка во многих отношениях предпочтительнее для тестирования пользовательского промежуточного ПО по сравнению с «правильными» модульными тестами, которые я показывал в разделе 36.1.

Независимо от того, как вы его назовете, этот тест основан на очень простой конфигурации тестового приложения. Вы также можете протестировать промежуточное ПО в контексте реального приложения, чтобы результат соответствовал его *реальной* конфигурации.

Если вы хотите запускать интеграционные тесты на основе существующего приложения, вам не нужно настраивать тестовый `HostBuilder` вручную, как вы это делали в листинге 36.7. Вместо этого можно использовать еще один вспомогательный пакет, `Microsoft.AspNetCore.Mvc.Testing`.

36.3.2 Тестирование приложения с помощью `WebApplicationFactory`

Создание экземпляра `HostBuilder` и использование пакета `Test Host`, как вы делали в разделе 36.3.1, могут быть полезны, когда вы хотите протестировать изолированные компоненты «инфраструктуры», такие как промежуточное ПО. Также очень часто у вас может возникнуть желание протестировать «реальное» приложение с полностью настроенным конвейером промежуточного ПО и всеми необходимыми сервисами, добавленными в контейнер внедрения зависимостей. Это дает максимальную уверенность в том, что ваше приложение будет работать в промышленном окружении.

`TestServer`, который предоставляет сервер в памяти, можно использовать для тестирования реального приложения, но, в принципе, здесь требуется гораздо больше настроек. Ваше реальное приложение, вероятно, загружает файлы конфигурации или статические файлы и может использовать страницы и представления Razor. К счастью, пакет `Microsoft.AspNetCore.Mvc.Testing` и класс `WebApplicationFactory` в значительной степени решают все эти проблемы конфигурации за вас.

ПРИМЕЧАНИЕ Не дайте ввести себя в заблуждение сегментом `Mvc` в имени пакета. Вы можете использовать этот пакет для тестирования ASP.NET Core приложений, не использующих MVC или страницы Razor.

Вы можете использовать класс `WebApplicationFactory` (из пакета `Microsoft.AspNetCore.Mvc.Testing`) для запуска версии вашего реального приложения в памяти. Он использует `TestServer` за кулисами, но работает с реальной конфигурацией вашего приложения, регистрацией сервисов в контейнере внедрения зависимостей и конвейером промежуточного ПО. Например, в следующем листинге показан пример, который проверяет, что когда ваше приложение получает запрос `"/ping"`, в ответе вы получаете `"pong"`.

Листинг 36.8 Создание интеграционного теста с классом WebApplicationFactory

Тестовый класс должен реализовывать интерфейс, хотя методов для реализации нет

```
public class IntegrationTests:
    IClassFixture<WebApplicationFactory<Program>>
{
    private readonly WebApplicationFactory<Program> _fixture;
    public IntegrationTests(
        WebApplicationFactory<Startup> fixture)
    {
        _fixture = fixture;
    }
    [Fact]
    public async Task PingRequest_ReturnsPong()
    {
        HttpClient client = _fixture.CreateClient();
        Создаем HttpClient, который отправляет запросы к TestServer в памяти
        var response = await client.GetAsync("/ping");
        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();
        Assert.Equal("pong", content);
        Внедряем экземпляр WebApplicationFactory<T>, где T – это класс в вашем приложении
        Выполняем запросы и проверяем ответ, как раньше
    }
}
```

Одно из преимуществ использования класса `WebApplicationFactory`, как показано в листинге 36.8, состоит в том, что для него требуется *меньше* ручной настройки, чем при использовании `TestServer` напрямую, как показано в листинге 36.7, несмотря на выполнение *большего* количества настроек за кулисами. `WebApplicationFactory` тестирует приложение, используя конфигурацию, определенную в файлах `Program.cs` и `Startup.cs`.

ПРИМЕЧАНИЕ Обобщенный `WebApplicationFactory<T>` должен ссылаться на открытый класс в вашем приложении. Обычно используются классы `Program` или `Startup`. Если вы пользуетесь операторами верхнего уровня (top-level statements, в .NET 7 используются по умолчанию), то автоматически генерируемый класс `Program` по умолчанию `internal`. Чтобы сделать его открытым и тем самым предоставить его для тестового проекта, необходимо добавить определение частичного класса в ваше приложение: `public partial class Program {}`.

Листинги 36.8 и 36.7 *концептуально* также сильно различаются. В листинге 36.7 вы проверяете, что класс `StatusMiddleware` ведет себя должным образом в контексте фиктивного приложения ASP.NET Core; в листинге 36.8 вы проверяете, что *ваше приложение ведет себя должным образом при определенных входных данных*. Ничего конкретного о том, как это происходит, не сказано. Чтобы пройти тест из листинга 36.8, вашему приложению не обязательно использовать `StatusMiddleware`;

оно просто должно правильно отреагировать на данный запрос. Это означает, что тесту меньше известно о деталях внутренней реализации вашего приложения и его волнует только его *поведение*.

ОПРЕДЕЛЕНИЕ Тесты, которые завершаются неудачно при неизменении приложения, называются *неустойчивыми*, или *хрупкими*. Страйтесь избегать таких тестов, убедившись, что они не зависят от деталей реализации вашего приложения.

Чтобы создать тесты, использующие класс `WebApplicationFactory`:

- 1 установите пакет `Microsoft.AspNetCore.Mvc.Testing` в свой проект, выполнив команду `dotnet add package Microsoft.AspNetCore.Mvc.Testing`, используя проводник NuGet в Visual Studio или добавив элемент `<PackageReference>` в файл своего проекта:

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Testing"
    Version="7.0.0" />
```

- 2 обновите элемент `<Project>` в файле тестового проекта с расширением `.csproj`:

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

Этого требует класс `WebApplicationFactory`, чтобы он мог найти ваши конфигурационные и статические файлы;

- 3 реализуйте `IClassFixture<WebApplicationFactory<T>>` в тестовом классе `xUnit`, где `T` – это класс в проекте вашего реального приложения. По соглашению для `T` обычно используется класс `Startup` из приложения:
 - `WebApplicationFactory` использует ссылку на `T` для поиска точки входа вашего приложения, запуска приложения в памяти и замены `Kestrel` на `TestServer` для тестов;
 - если вы используете операторы верхнего уровня C# и класс `Program` в качестве `T`, то необходимо убедиться, что класс `Program` доступен для тестового проекта. Вы можете изменить видимость автоматически сгенерированного класса `Program`, добавив `public partial class Program { }` в ваше приложение;
 - `IClassFixture<TestFixture>` – это интерфейс маркера `xUnit`, который дает указание `xUnit` создать экземпляр `TestFixture` перед построением тестового класса и внедрить экземпляр в конструктор тестового класса. Подробнее можно узнать на странице <https://xunit.net/docs/shared-context>;
- 4 примите экземпляр `WebApplicationFactory<T>` в конструкторе тестового класса. Вы можете использовать его для создания объекта `HttpClient`, чтобы отправлять запросы в памяти к `TestServer`. Эти запросы имитируют поведение вашего приложения в промышленном окружении, поскольку используются реальная конфигурация, сервисы и промежуточное ПО приложения.

Существенное преимущество класса `WebApplicationFactory` состоит в том, что вы можете с легкостью протестировать поведение реального

приложения. Но не стоит забывать и об ответственности – ваше тестовое приложение будет вести себя так же, как и в реальной ситуации, поэтому оно будет осуществлять запись в базу данных и отправлять данные сторонним API! В зависимости от того, что вы тестируете, вы можете заменить некоторые из своих зависимостей, чтобы избежать этого, а также облегчить тестирование.

36.3.3 Замена зависимостей в классе `WebApplicationFactory`

Когда вы используете класс `WebApplicationFactory` для запуска интеграционных тестов в своем приложении, приложение будет запускаться в памяти, но в остальном это похоже на то, как если бы вы запускали его с помощью команды `dotnet run`. Это означает, что любые строки подключения, секреты или API-ключи, которые можно загрузить локально, также будут использоваться для запуска вашего приложения.

СОВЕТ По умолчанию класс `WebApplicationFactory` использует окружение «Development» так же, как и при локальном запуске.

С одной стороны, это означает, что у вас есть подлинный тест, позволяющий правильно запускать приложение. Например, если вы зашли зарегистрировать нужную зависимость и это обнаруживается при запуске приложения, любые тесты, использующие `WebApplicationFactory`, завершатся ошибкой.

С другой стороны, это означает, что все ваши тесты будут использовать то же соединение с базой данных и сервисы, что и при локальном запуске приложения, и часто у вас может возникнуть желание заменить их альтернативными «тестовыми» версиями своих сервисов.

Простой пример. Представим, что сервис `CurrencyConverter`, который вы тестирували в этом приложении, использует интерфейс `IHttpClientFactory` для вызова стороннего API, чтобы получить последние курсы валют. Вы не хотите повторно использовать этот API в своих интеграционных тестах, поэтому хотите заменить сервис `CurrencyConverter` на собственный сервис `StubCurrencyConverter`.

Первый шаг – убедиться, что сервис `CurrencyConverter` реализует интерфейс, например `ICurrencyConverter`, и что ваше приложение использует его, а не реализацию. В нашем простом примере интерфейс, вероятно, будет выглядеть примерно так:

```
public interface ICurrencyConverter
{
    decimal ConvertToGbp(decimal value, decimal rate, int dps);
}
```

Нужно зарегистрировать сервис в `Program.cs`:

```
builder.Services.AddScoped<ICurrencyConverter, CurrencyConverter>();
```

Теперь, когда ваше приложение только косвенно зависит от сервиса `CurrencyConverter`, можно предоставить альтернативную реализацию в своих тестах.

СОВЕТ Использование интерфейса отделяет сервисы вашего приложения от конкретной реализации, позволяя заменять альтернативные реализации. Это ключевая практика для тестирования классов.

Мы создадим простую альтернативную реализацию интерфейса `ICurrencyConverter` для наших тестов, которая всегда возвращает одно и то же значение, 3. Очевидно, что она не очень полезна в качестве реального конвертера, но дело не в этом: у вас есть полный контроль! Создайте в своем тестовом проекте следующий класс:

```
public class StubCurrencyConverter : ICurrencyConverter
{
    public decimal ConvertToGbp(decimal value, decimal rate, int dps)
    {
        return 3;
    }
}
```

Теперь у вас есть все необходимое для замены реализации в тестах. Для этого мы будем использовать функцию класса `WebApplicationFactory`, которая позволяет настроить контейнер внедрения зависимостей перед запуском тестового сервера.

СОВЕТ Важно помнить, что нужно заменять реализацию *только* при запуске в тестовом проекте. Я видел, как некоторые пытались настроить *реальные* приложения, чтобы заменить действующие службы фейковыми, например когда задано определенное значение. Как правило, в этом нет необходимости, в приложении появляется слишком много тестовых сервисов, и обычно это приводит к путанице!

Класс `WebApplicationFactory` предоставляет метод `WithWebHostBuilder`, который позволяет настраивать приложение до запуска `TestServer` в памяти. В следующем листинге показан интеграционный тест, в котором этот метод используется для замены реализации интерфейса `ICurrencyConverter`, используемой «по умолчанию», с помощью нашей заглушки.

Листинг 36.9 Замена зависимости в teste с помощью метода `WithWebHostBuilder`

Реализуем тренебуемый интерфейс и внедряем его в конструктор	<pre>public class IntegrationTests: IClassFixture<WebApplicationFactory<Startup>> { private readonly WebApplicationFactory<Startup> _fixture; public IntegrationTests(WebApplicationFactory<Startup> fixture) { _fixture = fixture; } }</pre>
---	---

```
[Fact]
public async Task ConvertReturnsExpectedValue()
{
    var customFactory = _fixture.WithWebHostBuilder(
        (IWebHostBuilder hostBuilder) =>
    {
        hostBuilder.ConfigureTestServices(services =>
        {
            services.RemoveAll<ICurrencyConverter>();
            services.AddScoped<ICurrencyConverter, StubCurrencyConverter>();
        });
    });

    HttpClient client = customFactory.CreateClient();

    var response = await client.GetAsync("/api/currency");

    response.EnsureSuccessStatusCode();
    var content = await response.Content.ReadAsStringAsync();

    Assert.Equal("3", content);
}
}

Configure-  
TestServices  
выполняется  
после того, как  
все остальные  
сервисы внед-  
рения зависи-  
мостей будут  
настроены в ва-  
шем реальном  
приложении  
  
Вызов  
CreateClient  
загружает  
приложение  
и запускает  
TestServer

```

Создаем свою фабрику с дополнительной конфигурацией

Удаляет все реализации ICurrencyConverter из контейнера внедрения зависимостей

Добавляет тестовый сервис в качестве замены

Вызываем конечную точку конвертера валют

Поскольку тестовый конвертер всегда возвращает 3, конечная точка API делает то же самое.

В этом примере следует отметить несколько важных моментов:

- метод `WithWebHostBuilder()` возвращает *новый* экземпляр класса `WebApplicationFactory`. К этому экземпляру применена ваша индивидуальная конфигурация, в то время как исходный внедренный экземпляр `_fixture` остается неизменным;
- метод `ConfigureTestServices()` вызывается после метода `ConfigureServices()` реального приложения. Это означает, что вы можете заменить ранее зарегистрированные сервисы, а также можете использовать его для переопределения значений конфигурации, как будет показано в разделе 36.4.

Метод `WithWebHostBuilder()` удобен, когда вы хотите заменить сервис для одного теста. Но что, если вы хотите заменить интерфейс `ICurrencyConverter` в каждом тесте? Весь этот шаблонный код быстро станет проблемой. Вместо этого можно создать пользовательский класс `WebApplicationFactory`.

36.3.4 Уменьшение дублирования кода за счет создания своего класса `WebApplicationFactory`

Если вы обнаружите, что часто пишете метод `WithWebHostBuilder()` в своих интеграционных тестах, возможно, стоит создать свой класс `WebApplicationFactory`. В следующем листинге показано, как централизовать службу тестирования, которую мы использовали в листинге 36.9, в пользовательский класс `WebApplicationFactory`.

Листинг 36.10 Создание класса CustomWebApplicationFactory для уменьшения дублирования кода

```
public class CustomWebApplicationFactory
    : WebApplicationFactory<Program>
{
    protected override void ConfigureWebHost(
        IWebHostBuilder builder)
    {
        builder.ConfigureTestServices(services =>
        {
            services.RemoveAll<ICurrencyConverter>();
            services.AddScoped
                <ICurrencyConverter, StubCurrencyConverter>();
        });
    }
}
```

Добавляем пользовательскую конфигурацию для своего приложения

Наследуем от класса WebApplicationFactory

Есть много функций, которые можно переопределить. Это эквивалентно вызову WithWebHostBuilder

В этом примере мы переопределяем `ConfigureWebHost` и настраиваем тестовые сервисы для фабрики¹. Вы можете использовать этот класс в любом teste, внедрив его в виде `IClassFixture`, как делали это раньше. Например, в следующем листинге показано, как можно было бы обновить листинг 36.9, чтобы использовать специальную фабрику, определенную в листинге 36.10.

Листинг 36.11. Использование пользовательского WebApplicationFactory в интеграционном teste

```
public class IntegrationTests:
    IClassFixture<CustomWebApplicationFactory>
{
    private readonly CustomWebApplicationFactory _fixture;
    public IntegrationTests(CustomWebApplicationFactory fixture)
    {
        _fixture = fixture;
    }

    [Fact]
    public async Task ConvertReturnsExpectedValue()
    {
        HttpClient client = _fixture.CreateClient(); ←

        var response = await client.GetAsync("/api/currency");

        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();

        Assert.Equal("3", content); ←
    }
}
```

Внедряет экземпляр фабрики в конструктор

Реализует интерфейс IClassFixture для пользовательской фабрики

Клиент уже содержит конфигурацию тестовой службы

Результат подтверждает, что использовался тестовый сервис

¹ У класса `WebApplicationFactory` есть много других методов, которые можно переопределить для иных сценариев. Подробнее см. <http://mng.bz/mgq8>.

Вы также можете комбинировать свой класс `WebApplicationFactory`, переопределяющий сервисы, которые вы всегда хотите заменять, с методом `WithWebHostBuilder()`, чтобы переопределить дополнительные сервисы для каждого теста. Эта комбинация дает вам лучшее из обоих миров: вы уменьшаете дублирование кода с помощью класса `CustomWebApplicationBuilder` и получаете контроль с помощью конфигурации для каждого теста.

Выполнение интеграционных тестов с использованием реальной конфигурации приложения максимально приближает вас к гарантии его правильной работы. Камнем преткновения в этой ситуации почти всегда являются внешние зависимости, такие как сторонние API и базы данных.

В последнем разделе этой главы мы рассмотрим, как использовать поставщика `SQLite` для EF Core с базой данных в памяти. Этот подход можно использовать для написания тестов для сервисов, использующих контекст базы данных EF Core, без необходимости доступа к реальной базе данных.

36.4 Изоляция базы данных с помощью поставщика EF Core в памяти

В этом разделе вы узнаете, как писать модульные тесты для кода, который полагается на экземпляр класса `DbContext` от EF Core. Вы узнаете, как создать базу данных в памяти, а также узнаете разницу между поставщиком EF в памяти и поставщиком `SQLite` в памяти. Наконец, вы увидите, как использовать поставщик `SQLite` для создания быстрых изолированных тестов для кода, который полагается на экземпляр `DbContext`.

Как вы уже видели в главе 12, EF Core – это инструмент объектно-реляционного отображения, который используется в основном с реляционными базами данных. В этом разделе мы обсудим способ тестирования службы, зависящих от экземпляра класса `DbContext`, без необходимости настраивать реальную базу данных или взаимодействовать с ней.

ПРИМЕЧАНИЕ Чтобы узнать больше о тестировании кода EF Core, см. книгу «*Entity Framework Core в действии*», 2-е изд., Джона П. Смита (Manning, 2021), <http://mng.bz/5j87>.

В следующем листинге показана сильно урезанная версия класса `RecipeService`, который вы создали в главе 12 для приложения с рецептами. Он показывает единственный способ получить подробную информацию о рецепте с использованием внедренного экземпляра класса `DbContext`.

Листинг 36.12 Класс `RecipeService` для тестирования, использующий EF Core для хранения и загрузки сущностей

```
public class RecipeService
{
    readonly ApplicationContext _context;
    public RecipeService(ApplicationContext context)
```

↓
В конструктор внедряется `DbContext` от EF Core

```

{
    _context = context;
}
public RecipeViewModel GetRecipe(int id)
{
    return _context.Recipes
        .Where(x => x.RecipeId == id)
        .Select(x => new RecipeViewModel
    {
        Id = x.RecipeId,
        Name = x.Name
    })
    .SingleOrDefault();
}

```

В конструктор внедряется DbContext от EF Core

Использует свойство DbSet<Recipes> для загрузки рецептов и создает RecipeViewModel

Написание модульных тестов для этого класса – небольшая проблема. Модульные тесты должны быть быстрыми, повторяемыми и изолированными от других зависимостей, но у вас есть зависимость от `DbContext` из вашего приложения. Вы вряд ли захотите вести запись в реальную базу данных в модульных тестах, поскольку это сделает тесты медленными, потенциально неповторяемыми и сильно зависимыми от конфигурации базы данных: сбой по всем трем требованиям!

ПРИМЕЧАНИЕ В зависимости от окружения разработки вы, возможно, захотите использовать реальную базу данных для своих интеграционных тестов, несмотря на эти недостатки. Использование базы данных, подобной той, которую вы будете применять в промышленном окружении, увеличивает вероятность обнаружения каких-либо проблем в тестах. Вы можете найти пример использования Docker, чтобы добиться этой цели, в документации Microsoft «Тестирование служб и веб-приложений ASP.NET Core»: <http://mng.bz/zxDw>.

К счастью, для этого сценария у Microsoft есть два поставщика баз данных в памяти. Напомню, что в главе 12 говорится, что при настройке `DbContext` в `Program.cs` вы настраиваете конкретного поставщика базы данных, например SQL Server:

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(connectionString));
```

Поставщики баз данных в памяти – это альтернативные поставщики, предназначенные *только для тестирования*. Microsoft включает в ASP.NET Core двух поставщиков в памяти:

- *Microsoft.EntityFrameworkCore.InMemory* – этот поставщик не моделирует базу данных. Вместо этого он хранит объекты непосредственно в памяти. Это не реляционная база данных как таковая, поэтому она не обладает всеми функциями обычной базы данных. Вы не можете выполнить к ней SQL-запрос напрямую, и она не будет обеспечивать соблюдение ограничений, но работает она быстро;

- *Microsoft.EntityFrameworkCore.Sqlite* – SQLite – это реляционная база данных. Она очень ограничена в функциях по сравнению с такой базой данных, как SQL Server, но это настоящая реляционная база данных, в отличие от поставщика базы данных в памяти. Обычно база данных SQLite пишется в файл, но у поставщика есть режим *in-memory*, при котором база данных остается в памяти. Это значительно ускоряет и упрощает его создание и использование для тестирования.

К сожалению, миграции EF Core адаптированы к конкретной базе данных. Это означает, что вы не можете запускать миграции, созданные для SQL Server или PostgreSQL, на базе данных SQLite. Можно создать несколько наборов миграций, как описано в документации (<http://mng.bz/pP15>), но это может добавить много сложностей. Поэтому всегда используйте `EnsureCreated()` в тестах с SQLite. Этот метод создает базу данных без выполнения миграции, как вы увидите в листинге 36.13.

Вместо того чтобы хранить данные в базе данных на диске, оба этих поставщика хранят данные в памяти, как показано на рис. 36.2. Это дает возможность создавать новую базу данных для каждого теста, чтобы ваши тесты оставались изолированными друг от друга.

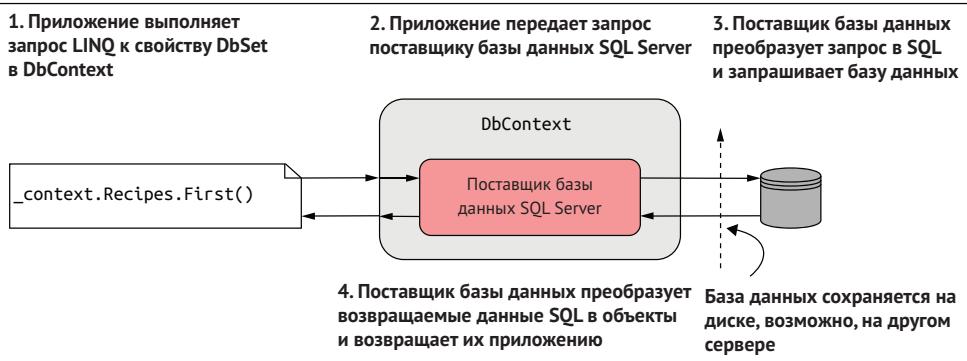
ПРИМЕЧАНИЕ В этом разделе я описываю, как использовать поставщика SQLite в качестве базы данных в памяти, поскольку она более полнофункциональна, чем поставщик в памяти. Для получения подробной информации об использовании поставщика в памяти см. <http://mng.bz/hd1q>.

Чтобы использовать поставщика SQLite в памяти, добавьте пакет `Microsoft.EntityFrameworkCore.Sqlite` в файл тестового проекта с расширением `.csproj`. Он добавляет метод расширения `UseSqlite()`, который вы будете использовать для настройки поставщика базы данных для своих модульных тестов.

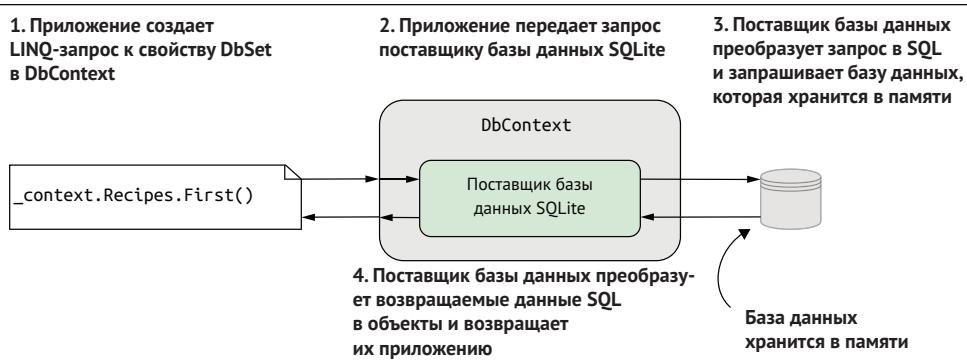
В листинге 36.13 показано, как использовать поставщика SQLite в памяти для тестирования метода `GetRecipe()` из класса `RecipeService`. Начните с создания объекта `SqliteConnection` и использования строки подключения `"DataSource=:memory:"`. Так вы сообщаете поставщику, что нужно хранить базу данных в памяти, а затем открываете соединение. Обычно это быстрее, чем использование подключения на основе файла, а также вы можете легко запускать несколько тестов параллельно, поскольку нет общей базы данных.

ВНИМАНИЕ База данных SQLite в памяти уничтожается, когда соединение закрыто. Если вы не открыли соединение самостоятельно, EF Core закроет его при удалении `DbContext`. Если вы хотите использовать базу данных в памяти между экземплярами `DbContext`, то должны явно открыть соединение самостоятельно.

Поставщик базы данных SQL Server



Поставщик базы данных SQLite (в памяти)



Поставщик базы данных в оперативной памяти

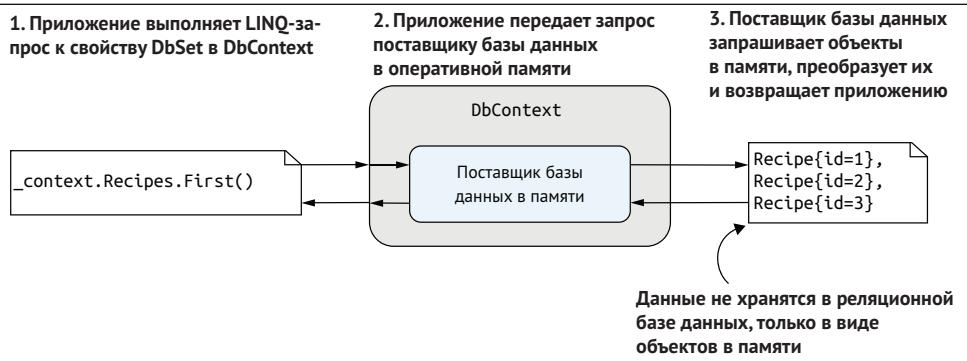


Рис. 36.2 Поставщик базы данных в памяти и поставщик SQLite (режим in-memory) по сравнению с поставщиком серверной базы данных SQL. Поставщик базы данных в памяти не моделирует базу данных как таковую. Вместо этого он сохраняет объекты в памяти и непрямую выполняет к ним LINQ-запросы

Затем передайте экземпляр `SqliteConnection` в `DbContextOptionsBuilder<>` и вызовите метод `UseSqlite()`. Так вы настраиваете результирующий объект `DbContextOptions<>` с необходимыми сервисами для поставщика SQLite и обеспечиваете подключение к базе данных в памяти. При

передаче этого объекта параметров в экземпляр `AppDbContext` все вызовы `DbContext` приводят к вызовам поставщика базы данных в памяти.

Листинг 36.13 Использование поставщика базы данных в памяти для тестирования `DbContext`

```
[Fact]
public void GetRecipeDetails_CanLoadFromContext()
{
    var connection = new SqliteConnection("DataSource=:memogu:");
    connection.Open(); ← Открывает соединение, поэтому EF
    Core не закрывает его автоматически

    Создает
    DbContext
    и передает
    параметры
    var options = new DbContextOptionsBuilder<AppDbContext>()
        .UseSqlite(connection)
        .Options; ← Создает экземпляр DbContextOptions<>
    Создает экземпляр DbContextOptions<>
    и настраивает его для использования
    соединения SQLite

    using (var context = new AppDbContext(options))
        ← Обеспечивает соответствие базы данных в памяти
        ← модели EF Core (аналогично запуску миграций)

        context.Database.EnsureCreated(); ←
        context.Recipes.AddRange(
            new Recipe { RecipeId = 1, Name = "Recipe1" },
            new Recipe { RecipeId = 2, Name = "Recipe2" },
            new Recipe { RecipeId = 3, Name = "Recipe3" });

        ← Сохраняет
        ← изменения
        ← в базе
        ← данных
        ← в памяти
        context.SaveChanges();
    }

    Добавляет
    несколько
    рецептов
    в DbContext
    using (var context = new AppDbContext(options)) ←
    {
        var service = new RecipeService(context); ←
        var recipe = service.GetRecipe(id: 2); ←
        Assert.NotNull(recipe);
        Assert.Equal(2, recipe.Id);
        Assert.Equal("Recipe2", recipe.Name);
    }

    Проверяет, правильно ли
    вы получили рецепт из
    базы данных в памяти
    } ← Выполняет функцию
    ← GetRecipe. Она выполняет за-
    ← прос к базе данных в памяти

    } ← Создает новый
    ← DbContext для
    ← проверки воз-
    ← можности из-
    ←влечения данных
    ← из DbContext

    } ← Создает
    ← RecipeService
    ← для тестирования
    ← и передает новый
    ← DbContext
}
```

Этот пример следует стандартному шаблону для каждого случая, когда вам нужно протестировать класс, зависящий от `DbContext`.

- 1 Создайте `SqliteConnection` со строкой подключения `"DataSource =:memogu:"` и откройте соединение.
- 2 Создайте `DbContextOptionsBuilder<>` и вызовите метод `UseSqlite()`, передав открытое соединение.
- 3 Извлеките объект `DbContextOptions` из свойства `Options`.
- 4 Передайте параметры экземпляру `DbContext` и убедитесь, что база данных соответствует модели EF Core, вызвав `context.Database.EnsureCreated()`. Это аналогично запуску миграций в вашей базе данных,

но это следует использовать *только* для тестовых баз данных. Создайте и добавьте все необходимые тестовые данные в базу данных в памяти и вызовите метод `SaveChanges()`, чтобы сохранить данные.

- 5 Создайте новый экземпляр `DbContext` и внедрите его в свой тестовый класс. Все запросы будут выполняться к базе данных в памяти.

Используя два отдельных экземпляра `DbContext`, можно избежать ошибок в тестах из-за кеширования данных EF Core без их записи в базу данных. При таком подходе вы можете быть уверены, что все данные, считанные во втором экземпляре `DbContext`, сохраняются в поставщике базы данных в памяти.

Это было очень краткое знакомство с использованием поставщика SQLite в качестве поставщика базы данных в памяти и тестированием с использованием EF Core в целом, но если вы последуете настройке, показанной в листинге 36.13, это займет у вас много времени. В исходном коде для этой главы показано, как объединить этот код с пользовательским `WebApplicationFactory`, чтобы использовать базу данных в памяти для своих интеграционных тестов. Дополнительные сведения о тестировании с EF Core, включая дополнительные параметры и стратегии, см. в книге «*Entity Framework Core в действии*», 2-е изд., Джона П. Смита (Manning, 2021).

Резюме

- Используйте класс `DefaultHttpContext` для модульного тестирования пользовательских компонентов промежуточного ПО. Если вам нужен доступ к телу ответа, вы должны заменить значение по умолчанию `Stream.Null` экземпляром `MemoryStream` и вручную прочитать поток после вызова промежуточного ПО;
- для контроллеров API и моделей страниц Razor можно применять модульное тестирование, как и для других классов, но они, как правило, не должны содержать бизнес-логики, поэтому усилия могут не окупиться. Например, контроллер API тестируется независимо от маршрутизации, проверки модели и фильтров, поэтому будет непросто протестировать логику, которая зависит от любого из этих аспектов;
- интеграционные тесты позволяют тестировать сразу несколько компонентов приложения, обычно в контексте самого фреймворка ASP.NET Core. Пакет `Microsoft.AspNetCore.TestHost` предоставляет объект `TestServer`, который можно использовать для создания простого веб-хоста для тестирования. Он создает сервер в памяти, к которому можно отправлять запросы и получать ответы. Вы можете использовать `TestServer` напрямую, если хотите создавать интеграционные тесты для специальных компонентов, таких как промежуточное ПО;
- для более обширных интеграционных тестов реального приложения следует использовать класс `WebApplicationFactory` из пакета `Microsoft.AspNetCore.Mvc.Testing`. Реализуйте `IClassFixture<WebApp`

- `icationFactory<Program>>` в своем тестовом классе и внедрите экземпляр `WebApplicationFactory<Program>` в конструктор. Так вы создадите версию всего своего приложения в памяти с использованием той же конфигурации, сервисов внедрения зависимостей и конвейера промежуточного ПО. Вы можете отправлять в свое приложение запросы в памяти, чтобы получить представление о том, как ваше приложение будет вести себя в промышленном окружении;
- чтобы настроить класс `WebApplicationBuilder`, вызовите методы `WithWebHostBuilder()` и `ConfigureTestServices()`. Этот метод вызывается после стандартной конфигурации внедрения зависимостей приложения, что позволяет добавлять или удалять сервисы по умолчанию для своего приложения, например заменять класс, который связывается со сторонним API, используя реализацию-заглушку;
 - если вы обнаружите, что вам необходимо настраивать сервисы для каждого теста, то можно создать специальный класс `WebApplicationFactory` путем наследования и переопределив метод `ConfigureWebHost`. Вы можете разместить всю свою конфигурацию в этом классе и реализовать `IClassFixture<CustomWebApplicationFactory>` в своих тестовых классах, вместо того чтобы вызывать метод `WithWebHostBuilder()` в каждом тестовом методе;
 - вы можете использовать поставщик EF Core SQLite в качестве базы данных в памяти для тестирования кода, который зависит от экземпляра `DbContext`. Вы настраиваете поставщика в памяти, создавая `SqliteConnection` со строкой подключения `"DataSource=:memory:"`. Создайте объект `DbContextOptionsBuilder<>` и вызовите метод `UseSqlite()`, передавая подключение. Наконец, передайте `DbContextOptions<>` в экземпляр `DbContext` своего приложения и вызовите метод `context.Database.EnsureCreated()` для подготовки базы данных в памяти для использования с EF Core;
 - сопровождение базы данных SQLite в памяти осуществляется до тех пор, пока существует открытый `SqliteConnection`. Открывая соединение вручную, базу данных можно использовать с несколькими экземплярами `DbContext`. Если вы не вызвали метод `Open()` для соединения, EF Core закроет соединение (и удалит базу данных в памяти) при удалении `DbContext`.

Приложение A.

Подготовка окружения разработки

Для разработчиков на .NET в мире, ориентированном на Windows, Visual Studio в прошлом была в значительной степени обязательным требованием. Но теперь, когда .NET и ASP.NET Core становятся кроссплатформенными, это уже не так.

Все, что относится к ASP.NET Core (создание новых проектов, сборка, тестирование и публикация), может запускаться из командной строки для любой поддерживаемой операционной системы. Все, что вам нужно, – это набор средств разработки .NET, который предоставляет интерфейс командной строки .NET. Кроме того, если вы работаете в Windows и незнакомы с командной строкой, вы все равно можете выбрать **File > New Project** в Visual Studio, чтобы сразу приступить к работе. С ASP.NET Core выбор за вами!

Аналогичным образом теперь вы можете получить отличные возможности редактирования за пределами Visual Studio благодаря проекту OmniSharp (www.omnisharp.com). Это набор библиотек и плагинов с открытым исходным кодом, которые предлагают автодополнение кода (IntelliSense) в широком диапазоне редакторов и операционных систем. То, как вы настроите свое окружение, скорее всего, будет зависеть от того, какую операционную систему вы используете и к чему привыкли.

Помните, что в случае с .NET 7 операционная система, которую вы выбираете для разработки, не имеет отношения к конечным системам, на которых вы можете запускать ваши приложения, – независимо от того, выберете ли вы Windows, macOS или Linux для разработки, вы можете развернуть приложение в любой поддерживаемой системе.

В этом приложении я покажу, как установить набор средств разработки .NET, чтобы вы могли создавать, запускать и публиковать приложения .NET, а также расскажу о некоторых интегрированных средах разработки и редакторах для создания приложений.

ПРИМЕЧАНИЕ В этой книге для большинства примеров я использую Visual Studio, но вы можете использовать любой из инструментов, которые я здесь обсуждаю. Предполагается, что вы успешно установили на свой компьютер .NET 7 и редактор кода.

A.1 Установка .NET SDK

Самое важное, что вам нужно для разработки на .NET Core и .NET 7, – это набор средств разработки .NET (.NET SDK). В этом разделе я опишу, как установить его и как проверить, какую версию вы установили.

Чтобы начать программировать на .NET, необходимо установить набор средств разработки .NET (прежнее название – набор средств разработки .NET Core). Он содержит базовые библиотеки, инструменты и компилятор, которые вам понадобятся для создания приложений на .NET.

Вы можете скачать его на странице <https://dotnet.microsoft.com/download>. Здесь есть ссылки для скачивания последней версии .NET для вашей операционной системы. Если вы используете Windows или macOS, то на этой странице вы найдете ссылки для скачивания инсталлятора; если вы используете Linux, то здесь есть инструкции по установке .NET с помощью диспетчера пакетов вашего дистрибутива в виде пакета Snap или загрузки вручную.

ПРЕДУПРЕЖДЕНИЕ Убедитесь, что вы скачиваете .NET SDK, а не среду выполнения .NET. Среда выполнения .NET используется для выполнения приложений .NET, но ее нельзя применять для их создания. .NET SDK включает в себя копию среды выполнения, поэтому он может запускать ваши приложения, но также может создавать, тестировать и публиковать их.

После установки .NET SDK вы можете запускать команды из интерфейса командной строки .NET с помощью команды `dotnet`. Выполните команду `dotnet --info`, чтобы увидеть информацию о версии .NET SDK, которая используется в настоящее время, а также установленные вами .NET SDK и среды выполнения, как показано на рис. А.1.

Как видно на рис. А.1, у меня установлено несколько версий .NET SDK. Это прекрасно, но не обязательно. Более новые версии .NET SDK могут создавать приложения, предназначенные для более старых версий .NET Core. Например, SDK для .NET 7 может создавать приложения для .NET 6, .NET 5, .NET Core 3.1 и т. д., а вот SDK для .NET 6 не может создавать приложения для .NET 7.

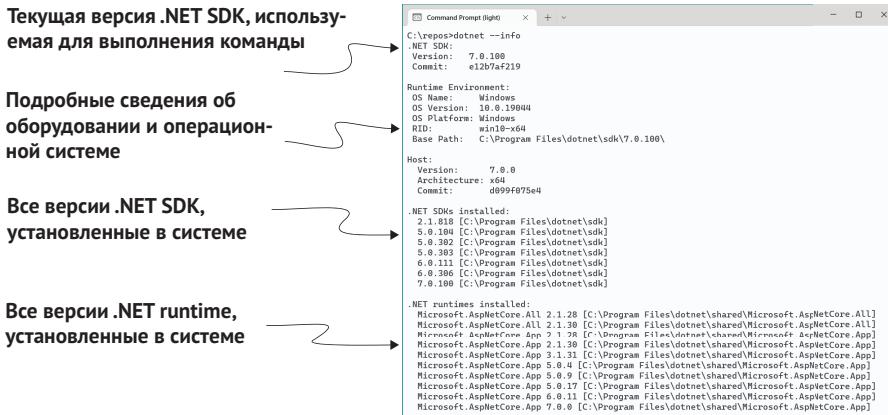


Рис. А.1 Используйте команду `dotnet --info`, чтобы проверить, какая версия .NET SDK применяется в настоящее время и какие версии доступны. На этом скриншоте показано, что в настоящее время я использую SDK для .NET 7, версию 7.0.100

СОВЕТ Некоторые IDE, такие как Visual Studio, могут автоматически устанавливать .NET 7 в процессе установки. Нет проблем с установкой нескольких версий .NET одновременно, поэтому вы всегда сможете установить .NET SDK вручную, независимо от того, установила ваша IDE другую версию или нет.

По умолчанию, когда вы запускаете команды `dotnet` из командной строки, вы будете использовать последнюю версию установленного вами .NET SDK. Вы можете контролировать это и использовать более старую версию SDK, добавив в папку файл `global.json`. Чтобы получить представление об этом файле, о том, как его использовать, и о системе управления версиями .NET, см. статью в моем блоге: <http://mng.bz/KMzP>.

СОВЕТ Если в процессе установки у вас возникнут какие-либо проблемы, команда `dotnet` не распознается или вы получите сообщение об ошибке при ее выполнении, я предлагаю проверить документацию по установке, чтобы получить советы по устранению неполадок: <https://learn.microsoft.com/dotnet/core/install>.

После установки .NET SDK пора выбрать интегрированную среду разработки или редактор. Доступные варианты будут зависеть от того, какую операционную систему вы используете, и в значительной степени будут определяться личными предпочтениями.

A.2 Выбор интегрированной среды разработки или редактора кода

В этом разделе я опишу несколько самых популярных IDE и редакторов для разработки на платформе .NET, а также способы их установки. Выбор IDE – очень личный вопрос, поэтому в этом разделе описаны только

некоторые из вариантов. Если вашей любимой IDE нет в списке, обратитесь к документации, чтобы узнать, поддерживается ли .NET.

A.2.1 Visual Studio (Windows)

Долгое время Windows была лучшей системой для создания приложений на .NET, и, учитывая доступность Visual Studio, возможно, что это по-прежнему так.

Visual Studio (рис. А.2) – это полнофункциональная интегрированная среда разработки, которая предоставляет одно из лучших универсальных средств разработки приложений ASP.NET Core. К счастью, теперь есть бесплатная версия Visual Studio Community для студентов и небольших групп разработчиков.

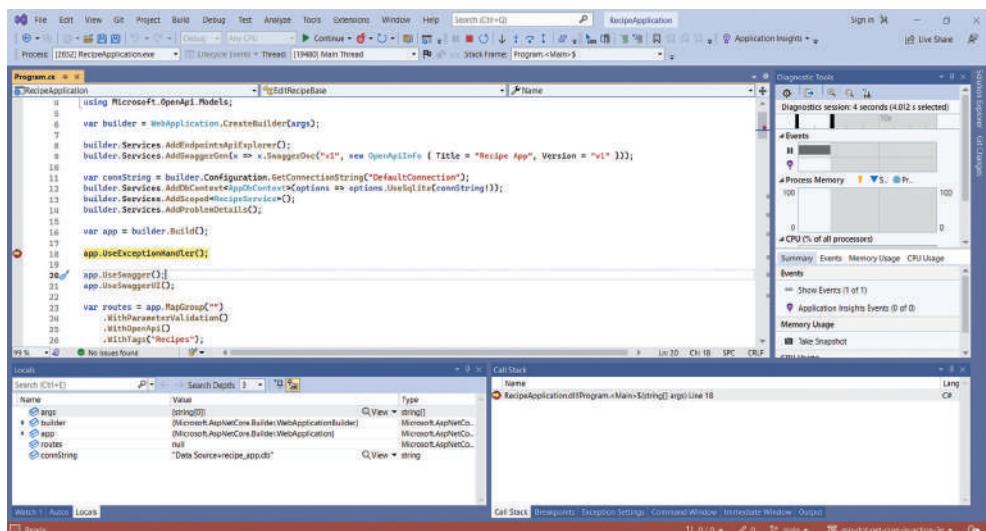


Рис. A.2 Visual Studio предоставляет одно из самых полноценных окружений для разработки ASP.NET Core для пользователей Windows

Visual Studio поставляется с целым набором шаблонов для создания новых проектов, лучшей в своем классе отладки и публикации без необходимости прикасаться к командной строке. Она особенно подходит, если вы публикуете приложения в Azure, поскольку у нее есть множество прямых подключений к функциям Azure, чтобы упростить разработку и развертывание.

Вы можете установить Visual Studio, посетив страницу <https://visualstudio.microsoft.com/vs/> и щелкнув **Download Visual Studio** (Скачать Visual Studio). Выберите **Community Edition** (если у вас нет лицензии на версию Professional или Enterprise) и следуйте инструкциям по установке.

Установщик Visual Studio – сам по себе приложение, и он попросит вас выбрать *рабочие нагрузки* для установки. Вы можете выбрать столько, сколько захотите, но для разработки на ASP.NET Core убедитесь, что вы выбрали как минимум ASP.NET и веб-разработку.

СОВЕТ В установщике доступно множество рабочих нагрузок и необязательных компонентов. Не переживайте об установке всего сразу, вы всегда сможете снова запустить установщик, чтобы добавить или удалить компоненты.

После выбора этих рабочих нагрузок нажмите **Download** (Скачать) и выберите напиток на свой вкус. Несмотря на то что размер инсталлятора недавно уменьшился, Visual Studio по-прежнему требуется несколько гигабайтов для загрузки и установки. По завершении вы будете готовы приступить к созданию приложений на ASP.NET Core.

A.2.2 Rider от компании JetBrains (Windows, Linux, macOS)

Rider (рис. А.3) от компании JetBrains представляет собой кроссплатформенную интегрированную среду разработки, альтернативу Visual Studio. Выщенная в 2017 году, Rider – еще одна полнофункциональная среда разработки, созданная на базе известного плагина ReSharper. Если вы привыкли использовать Visual Studio с плагином ReSharper и множеством функций рефакторинга, которые он предоставляет, то настоятельно рекомендую попробовать Rider. Кроме того, если вы знакомы с продуктами IntelliJ от JetBrains, с Rider вы будете чувствовать себя как дома.

Чтобы установить Rider, посетите сайт <https://www.jetbrains.com/rider/> и щелкните **Download** (Скачать). Есть 30-дневная бесплатная пробная версия, а по истечении этого срока нужно будет приобрести лицензию. Если у вас есть лицензия ReSharper, то, возможно, у вас уже есть лицензия и на Rider. Они также предлагают скидки или бесплатные лицензии для различных пользователей, например студентов и старшекурсников, так что это стоит вашего внимания.

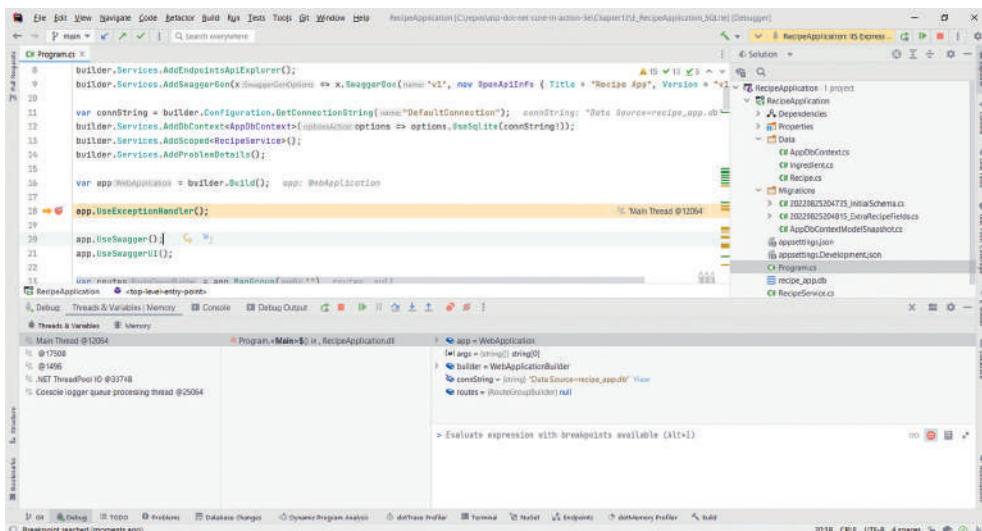


Рис. А.3 Rider – это кроссплатформенная интегрированная среда разработки от JetBrains. Она основана на плагине ReSharper для Visual Studio, поэтому включает в себя множество тех же функций рефакторинга, а также отладчик, средство запуска тестов и все остальные функции интеграции, которые вы ожидаете от полнофункциональной среды разработки

A.2.3 Visual Studio для Mac (macOS)

Несмотря на фирменный стиль, Visual Studio для Mac полностью отличается от Visual Studio. Это обновленная и расширенная версия по сравнению с предшественником, Xamarin Studio, и теперь вы можете использовать Visual Studio для Mac для создания приложений ASP.NET Core на macOS. Visual Studio для Mac обычно имеет меньше функций, чем Visual Studio или Rider, но предлагает нативную IDE и находится в активной разработке.

Чтобы установить Visual Studio для Mac, посетите страницу <https://visualstudio.microsoft.com/vs/mac>, щелкните **Download Visual Studio for Mac** (Скачать Visual Studio для Mac), скачайте и запустите установщик.

A.2.4 Visual Studio Code (Windows, Linux, macOS)

Иногда полноценная интегрированная среда разработки не нужна. Возможно, вы хотите быстро просмотреть или отредактировать файл, или вам не нравится иногда непредсказуемая производительность Visual Studio. В этих случаях простой редактор кода – возможно, все, что вам нужно, и Visual Studio Code – отличный выбор. Visual Studio Code (рис. А.4) – это легковесный редактор с открытым исходным кодом, который обеспечивает редактирование, IntelliSense и отладку для широкого диапазона языков, включая C# и ASP.NET Core.

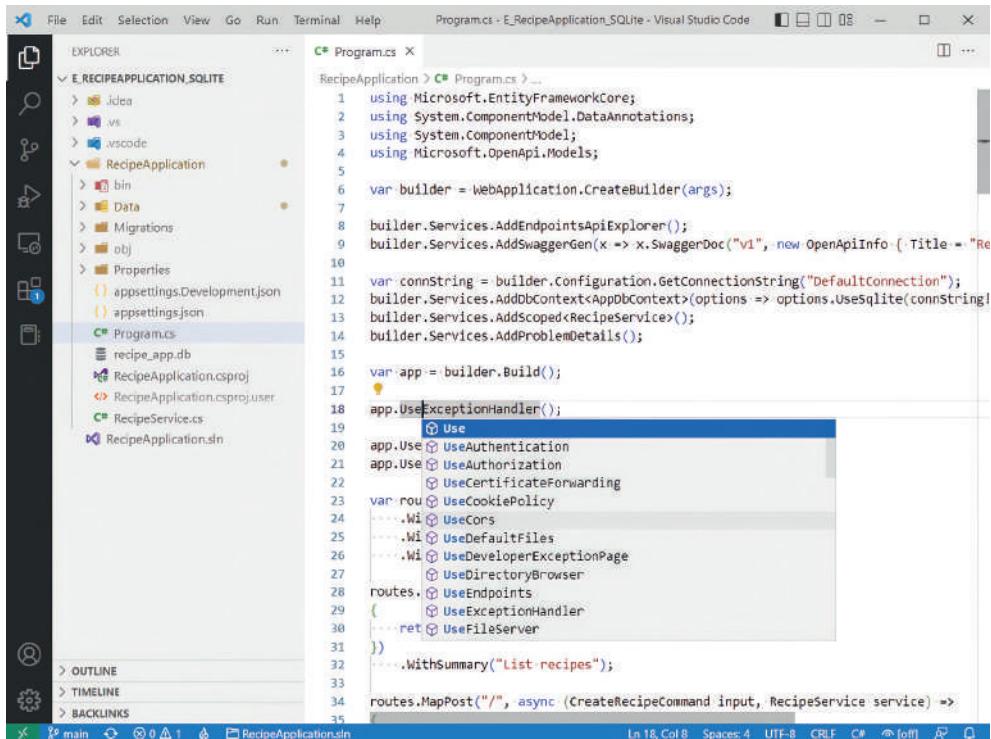


Рис. А.4 Visual Studio Code предоставляет кросс-платформенные IntelliSense и отладку

Чтобы установить Visual Studio Code, посетите сайт <https://code.visualstudio.com/>, щелкните **Download** (Скачать) и запустите скачанный установщик.

ПРИМЕЧАНИЕ Обязательно выберите загрузку, соответствующую вашей операционной системе и архитектуре. Страница загрузки пытается выбрать наиболее подходящую загрузку. Вы можете увидеть все доступные параметры по адресу <https://code.visualstudio.com/#alt-downloads>.

При первом открытии папки, содержащей проект C# или файл решения с Visual Studio Code, вам будет предложено установить расширение C#. Это дает возможность использовать IntelliSense и интеграцию между Visual Studio Code и .NET SDK.

Модель расширений VS Code – одно из самых больших его преимуществ, поскольку вы можете добавить огромное количество дополнительных функций. Независимо от того, работаете вы с Azure, AWS или любой другой технологией, обязательно посетите каталог расширений на странице <https://marketplace.visualstudio.com/vscode>, чтобы узнать, какие варианты вам доступны. Если вы ищете «.NET Core», то также найдете огромное количество расширений, которые могут приблизить VS Code к полноценной среде разработки, если вы этого хотите.

Приложение Б.

Полезные ссылки

В этом приложении я приведу ряд ссылок и справочных материалов, которые считаю полезными для изучения .NET 7 и ASP.NET Core.

Б.1 Список литературы

В данной книге мы затронули несколько тем и аспектов экосистемы .NET, которые в некоторой степени второстепенны при создании приложений на ASP.NET Core. Для более глубокого понимания этих тем я рекомендую книги, перечисленные в данном разделе. В них рассказывается о тех областях, с которыми вы неизбежно столкнетесь при создании приложений на ASP.NET Core:

- *Владимир Хориков.* Принципы юнит-тестирования (Питер, 2022), <http://mng.bz/E2go>. Научитесь совершенствовать свои модульные тесты, используя лучшие современные методы, из этой замечательной книги, содержащей примеры на C#;
- *Дастин Мецгар.* .NET в действии. 2-е изд. (Manning, 2023), <http://mng.bz/OxPK>. Приложения .NET Core создаются с использованием .NET 7. В этой книге содержится все, что вам нужно знать о работе на платформе;
- *Рой Ошеров.* Искусство автономного тестирования с примерами на C#. 3-е изд. (ДМК Пресс, 2016), <http://mng.bz/lW5o>. В книге «ASP.NET Core в действии» я обсуждаю механику модульного тестирования приложений ASP.NET Core. Для более глубокого обсуждения того, как создавать тесты, я рекомендую книгу «Искусство автономного тестирования»;
- *Крис Сэйти.* Blazor в действии (ДМК Пресс, 2023), <http://mng.bz/l1P6>. Blazor – это новая потрясающая платформа, которая использует возможности стандартного WebAssembly для запуска .NET

в браузере. С помощью Blazor вы можете создавать одностраничные приложения так же, как и с помощью платформ JavaScript, таких как Angular или React, но используя язык C# и инструменты, которые вы уже знаете;

- *Джон П. Смит.* Entity Framework Core в действии. 2-е изд. (ДМК Пресс, 2022), <http://mng.bz/BRj0>. Если вы используете EF Core в своих приложениях, я настоятельно рекомендую Entity Framework Core в действии. В ней описаны все функции и недостатки EF Core, а также способы настройки производительности вашего приложения;
- *Стивен Ван Дерсен и Марк Симан.* Внедрение зависимостей на платформе .NET (Питер, 2021), <http://mng.bz/d4lN>. Внедрение зависимостей – это основной аспект ASP.NET Core, поэтому принципы, практики и шаблоны внедрения зависимостей сейчас особенно актуальны. Книга знакомит с паттернами и антипаттернами внедрения зависимостей в контексте .NET и языка C#.

Б.2 Анонсы в блогах

Когда Microsoft выпускает новую версию ASP.NET Core или .NET Core, то обычно публикует анонс в блоге. В этих сообщениях представлен общий обзор темы со множеством примеров новых функций. Это отличная отправная точка, если вы хотите быстро ознакомиться с темой:

- *Джон Дуглас, Джереми Ликнесс и Ангелос Петропулос.* «.NET 7 доступен сегодня», блог .NET (Microsoft, 8 ноября 2022 г.), <http://mng.bz/Y1Ro>.
- анонс в блоге о .NET 7, описывающий огромное количество функций, представленных в .NET 7;
- *Ричард Лендер.* «Представляем .NET 5», блог .NET (Microsoft, 6 мая 2019 г.), <http://mng.bz/Gy9M>. Исходное объявление в блоге о .NET 5, описывающее видение платформы One .NET;
- *Иммо Ландверт.* «Будущее .NET Standard», блог .NET (Microsoft, 15 сентября 2020 г.), <http://mng.bz/zX0w>. Обсуждение того, что означает .NET 5 для будущего .NET Standard, включая рекомендации для авторов библиотек;
- *Иммо Ландверт.* «Стандарт .NET – демистификация .NET Core и .NET Standard», Microsoft Developer Network (Microsoft, сентябрь 2017 г.), <http://mng.bz/OKlp>. Длинная статья, знакомящая с .NET Core и объясняющая, какое место .NET Standard занимает в экосистеме .NET;
- документы Microsoft, «Политика поддержки .NET и .NET Core», <http://mng.bz/Ke9P>. Официальная политика поддержки Microsoft для .NET Core и .NET 7;
- *Дэниел Рот.* «Анонс ASP.NET Core в .NET 7», блог ASP.NET (Microsoft, 8 ноября 2022 г.), <http://mng.bz/gBve>. Сообщение в блоге с анонсом ASP.NET Core 7, в котором описывается, как обновить проект с .NET 6 до .NET 7, и предоставляются ссылки на многие новые функции, представленные в ASP.NET Core 7.

Б.3 Документация Microsoft

Исторически сложилось так, что документация Microsoft была скучной, но с появлением ASP.NET Core были предприняты огромные усилия, чтобы обеспечить ее полезность и актуальность. Вы можете найти пошаговые инструкции, целевую документацию по конкретным функциям и поддерживаемым API и даже компилятор C#, встроенный в браузер:

- Microsoft Docs, «.NET API Browser», <https://docs.microsoft.com/dotnet/api/>. Это браузер, который можно использовать, чтобы определить, какие API и на каких платформах .NET доступны;
- Microsoft Docs, «Документация по ASP.NET», <https://docs.microsoft.com/aspnet/core/>. Это официальная документация по ASP.NET Core;
- Microsoft Docs, «Ориентация на кросс-платформенность», <https://docs.microsoft.com/dotnet/standard/library-guidance/cross-platformtargeting>;
- Официальное руководство по выбору целевого фреймворка для своих библиотек; «Entity Framework Core», <https://learn.microsoft.com/ef/>. Это официальная документация по EF Core.

Б.4 Ссылки по темам, связанным с безопасностью

Безопасность – важный аспект современной веб-разработки. В этом разделе содержатся материалы, на которые я регулярно ссылаюсь и в которых описываются передовые практики работы веб-разработки, а также методы, которых следует избегать:

- *Дэнисе*. «Документация Duende IdentityServer v6», <https://docs.duendesoftware.com/identityserver/v6>. Документация по IdentityServer от Duende, OpenID Connect и платформе OAuth 2.0 для ASP.NET Core;
- *Доминик Байер*. «Доминик Байер об управлении идентификацией и доступом» (блог), <https://lessprivilege.com/>. Личный блог Доминика Байера, одного из авторов IdentityServer. Отличный ресурс при работе с аутентификацией и авторизацией в ASP.NET Core;
- *Скотт Хельме*, <https://scotthelme.co.uk/>. Блог Скотта Хельме с советами по стандартам безопасности, особенно с заголовками безопасности, которые можно добавлять в свое приложение;
- *Скотт Хельме*, «SecurityHeaders.io – анализируйте заголовки HTTP-ответов», <https://securityheaders.com/>. Проверьте заголовки безопасности своего сайта и получите совет относительно того, для чего и как добавить их в свое приложение;
- *Трой Хант*, <https://www.troyhunt.com>. Личный блог Троя Ханта с советами по безопасности для веб-разработчиков, в частности для разработчиков на .NET;
- Microsoft Docs, «Обзор безопасности ASP.NET Core» (Microsoft, 6 марта 2022 г.), <https://docs.microsoft.com/aspnet/core/security/>. Домашняя страница официальной документации ASP.NET Core по всем вопросам, связанным с безопасностью.

Б.5 Репозитории ASP.NET Core на GitHub

ASP.NET Core – проект с полностью открытым кодом и репозиториями на сайте GitHub. Один из лучших способов, который я нашел, чтобы изучить этот фреймворк, – просмотреть его исходный код. Этот раздел содержит основные репозитории для ASP.NET Core, .NET 7 и EF Core:

- .NET Foundation, «ASP.NET Core», <https://github.com/dotnet/aspnetcore>. Библиотеки фреймворка, образующие ASP.NET Core;
- .NET Foundation, «Entity Framework Core», <https://github.com/dotnet/efcore>. Библиотека EF Core;
- .NET Foundation, «.NET Runtime», <https://github.com/dotnet/runtime>. Среда выполнения .NET CoreCLR и библиотеки BCL, а также библиотеки расширений;
- .NET Foundation, «.NET SDK and CLI», <https://github.com/dotnet/sdk>. Интерфейс командной строки .NET (CLI), ресурсы для сборки .NET SDK и шаблоны проектов;
- .NET Foundation, «Образ Docker для .NET», <https://github.com/dotnet/dotnet-docker>. Определения Dockerfile для официальных образов .NET Docker.

Б.6 Инструменты и сервисы

В этом разделе содержатся ссылки на инструменты и сервисы, которые можно использовать для создания проектов на ASP.NET Core:

- .NET SDK – <https://dotnet.microsoft.com/download>;
- Cloudflare, глобальная сеть доставки контента, которую можно использовать, чтобы добавить кеширование и протокол HTTPS в свои приложения бесплатно, – <https://www.cloudflare.com>;
- JetBrains Rider – <https://www.jetbrains.com/rider>;
- Let's Encrypt, бесплатный, автоматизированный и открытый центр сертификации. Вы можете использовать его, чтобы получить бесплатные сертификаты SSL для защиты своего приложения, – <https://letsencrypt.org/>;
- .NET Boxed от Мухаммеда Рехана Саида, обширная коллекция шаблонов, чтобы начать работать с ASP.NET Core, предварительно настроенных с учетом множества передовых методов, – <https://github.com/Dotnet-Boxed/Templates>;
- Visual Studio, Visual Studio для Mac и Visual Studio Code – <http://www.visualstudio.com>.

Б.7 Блоги ASP.NET Core

Этот раздел содержит блоги, посвященные ASP.NET Core. Пытаетесь ли вы получить обзор какой-то общей темы или решить конкретную проблему, вам может быть полезно иметь несколько точек зрения по данной теме:

- *Халид Абухакмех.* Абухакмех, <https://khalidabuhakmeh.com/>. Большой выбор статей от Халида, посвященных .NET и разработке программного обеспечения в целом;
- *Крис Алкок.* Утренний напиток, <http://blog.cwa.me.uk/>. Коллекция статей, посвященных .NET, обновляется ежедневно;
- *Дэмиен Боден.* Разработка программного обеспечения, <https://damienvbod.com>. Отличный блог технического эксперта Microsoft Дэмиена Бодена об ASP.NET Core с большим количеством статей об ASP.NET Core и Angular;
- *Майк Бринд.* Mikesdotnetting, <https://www.mikesdotnetting.com/>. У Майка Бринда много статей об ASP.NET Core, особенно тех, что касаются ASP.NET Core Razor Pages;
- *Стив Гордон.* Стив Гордон – Пишите код со Стивом, <https://www.stevegordon.co.uk/>. Персональный блог Стива Гордона, посвященный .NET. Часто уделяется внимание написанию высокопроизводительного кода с .NET;
- *Скотт Хансельман.* Скотт Хансельман, <https://www.hanselman.com/blog>. Персональный блог известного докладчика Скотта Хансельмана. Разнообразный блог, ориентированный преимущественно на .NET;
- *Эндрю Лок.* .NET Escapades, <https://andrewlock.net>. Мой персональный блог, посвященный ASP.NET Core;
- *Microsoft .NET Team*, блог .NET, <https://blogs.msdn.microsoft.com/dotnet>. Блог команды .NET со множеством отличных ссылок;
- *Дэвид Пайн.* IEvangelist, <http://davidpine.net/>. Персональный блог технического эксперта Microsoft Дэвида Пайна с большим количеством статей об ASP.NET Core;
- *Мухаммед Рехан Сайд,* <https://rehansaeed.com>. Персональный блог Мухаммеда Рехана Саида, технического эксперта Microsoft и автора .NET Boxed;
- *Рик Штрайль*, веб-блог Рика Штраля, <https://weblog.west-wind.com>. Отличный блог, охватывающий широкий спектр тем по ASP.NET Core;
- *Филип В., StrathWeb*, <https://www.strathweb.com>. Множество статей об ASP.NET Core и ASP.NET от Филипа, технического эксперта Microsoft и активного участника проектов с открытым исходным кодом.

Б.8 Ссылки на видео

Если вы предпочитаете видео для изучения предмета, то рекомендую ознакомиться со ссылками из этого раздела. В частности, видеотрансляции от сообщества ASP.NET Core дают отличное представление об изменениях, которые вы увидите в будущих версиях ASP.NET Core, непосредственно от команды, занимающейся созданием фреймворка.

- *Microsoft .NET Conf 2022*, видеоплейлист YouTube (15 ноября 2022), <http://mng.bz/8r4Z>. Все сессии онлайн-конференции .NET Conf 2022, посвященной анонсу .NET 7;

- *.NET Foundation*, .NET Community Standup, <https://live.asp.net>. Еженедельные видеоролики, в которых команда ASP.NET Core обсуждает разработку фреймворка. Также включает видеотрансляции с командами .NET, Xamarin и EF Core;
- *Иммо Ландверт*, .NET Standard – Introduction, видео на YouTube (28 ноября 2016 г.), <http://mng.bz/VdOP>. Первое видео из отличной серии, посвященной .NET Standard;
- *Стив Гордон*. «Интеграционное тестирование приложений ASP.NET Core: передовой опыт», курс на сайте Pluralsight, 3 часа 25 минут (15 июля 2020 г.), <http://mng.bz/A09z>. Один из курсов Стива Гордона, где даются советы и рекомендации по созданию приложения на ASP.NET Core;
- *Ник Чапсас*. «Ник Чапсас», канал YouTube (14 ноября 2022 г.), <http://mng.bz/pPp5>. YouTube-канал Ника Чапсаса, на котором размещено множество видеороликов о .NET и ASP.NET Core.

Книги издательства «ДМК Пресс» можно купить оптом
в книготорговой компании «Галактика»
(представляет интересы издательств «ДМК ПРЕСС»,
«СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
Тел. +7(499) 782-38-89.
Электронная почта: books@aliants-kniga.ru.

Эндрю Лок

ASP.NET Core в действии

Третье издание

Главный редактор *Мовчан Д. А.*

dmkpress@gmail.com

Перевод *Беликов Д. А.*

Корректор *Синяева Г. И.*

Верстка *Луценко С. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура «NewBaskervilleC». Печать цифровая.

Усл. печ. л. 84,99. Тираж 100 экз.

Веб-сайт издательства: www.dmkpress.com

Фреймворк ASP.NET Core предоставляет все необходимое для создания веб-приложений профессионального качества. Благодаря повышенной производительности библиотекам для отрисовки на стороне сервера, безопасным API, простому доступу к данным и многому другому вы потратите свое время на реализацию функциональных возможностей, а не на исследование синтаксиса и отслеживание ошибок.

В третьем издании показано, как создавать веб-приложения для эксплуатации в промышленном окружении с помощью ASP.NET Core 7.0. Вы будете учиться на практических примерах, содержательных иллюстрациях и коде с подробными пояснениями. В числе новинок: создание минимальных API, обеспечение безопасности API с помощью токенов на предъявителя, WebApplicationBuilder и многое другое.

Рассматриваемые темы:

- минимальные API для обслуживания JSON;
- отрисовка на стороне сервера с использованием Razor Pages;
- доступ к данным с помощью Entity Framework Core;
- написание собственного промежуточного программного обеспечения и компонентов.

*Издание предназначено для веб-разработчиков среднего уровня.
Примеры, приведенные в книге, написаны на C#.*

Эндрю Лок – технический эксперт, имеющий статус Microsoft MVP. Работал с ASP.NET Core еще до первого выпуска фреймворка.

«ASP.NET Core — это обширная тема. К счастью, по ней есть большая книга! Это моя Библия по ASP.NET».

*Питер Моррис,
blazor-university.com*

«В этой книге мастерски разбирается веб-программирование. Ее необходимо прочитать как новичкам, так и опытным разработчикам. Она задает стандарты в пространстве .NET».

Халид Абухакмех, JetBrains

«Обширный опыт и практические советы Эндрю Лока действительно помогут любому веб-разработчику .NET превратиться из абсолютного новичка в профессионала».

*Джеймс Хики,
Savvyy Technologies*

«Содержит потрясающие сведения, которые помогут вам создавать лучшие приложения. Эта книга всегда на моем столе».

*Фостер Хейнс,
Foster's Website Company*



ISBN 978-5-93700-183-2



9 785937 001832 >